

UCLA

UCLA Electronic Theses and Dissertations

Title

Reporting Bugs in Metaprograms

Permalink

<https://escholarship.org/uc/item/0mn5w50d>

Author

Kalhauge, Christian Gram

Publication Date

2020

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Reporting Bugs in Metaprograms

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Christian Gram Kalhauge

2020

© Copyright by
Christian Gram Kalhauge
2020

ABSTRACT OF THE DISSERTATION

Reporting Bugs in Metaprograms

by

Christian Gram Kalhauge

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2020

Professor Jens Palsberg, Chair

As programs have gotten more sophisticated and integrated into our society, bugs have become a significant concern. To avoid this, we put a lot of trust in the metaprograms we use to analyze and compile them; however, these metaprograms might also contain bugs. The problem when reporting bugs in metaprograms is that they take programs as inputs: the bug might equally well be in the input program as in the metaprogram. In this dissertation, we show that modeling the input programs' validity improves our ability to verify and reduce bug reports that target metaprograms.

The dissertation of Christian Gram Kalhauge is approved.

Miryung Kim

Ravi Netravali

Harry Xu

Jens Palsberg, Committee Chair

University of California, Los Angeles

2020

Thank you. You are my first and last Line of defence.

TABLE OF CONTENTS

1	Introduction	1
1.1	Thesis Statement	3
2	Seperating Bugs from Deliberate Unsoundness in Static Analyses	6
2.1	Introduction	6
2.2	The Challenge	8
2.2.1	Examples	9
2.2.2	The Concept of a Useful Classifier	10
2.2.3	Our Solution: Easiness Analysis	11
2.3	Our Classifier	11
2.4	An Instance of the Challenge	13
2.5	Experiments	15
2.5.1	Static Analyses	15
2.5.2	Setup and Dataset	17
2.5.3	Quality of the Static Analysis	19
2.5.4	Easiness Analysis	21
2.6	Ground Truth	22
2.6.1	Static and Dynamic Analysis	22
2.6.2	Feature Detection	23
2.6.3	Bug Detection	23
2.6.4	Bug Reports	24
2.6.5	Results	26

2.7	Evaluation	28
2.8	Related Work	31
2.9	Summary	35
3	Binary Reduction of Dependency Graphs	36
3.1	Introduction	36
3.2	The Challenge	38
3.3	Reduction of Dependency Graphs	44
3.4	Binary Reduction	47
3.4.1	The Weighted Input Reduction Problem	47
3.4.2	The Binary Reduction Algorithm	49
3.5	Experimental Results	52
3.5.1	Experimental Setup	53
3.5.2	Results	55
3.5.3	Threats to Validity	57
3.5.4	Data Availability	58
3.6	Reporting bugs	59
3.7	Related Work	60
3.8	Summary	64
4	Logical Input Reduction	65
4.1	Introduction	65
4.2	Example	68
4.3	Modeling Dependencies	74
4.3.1	Featherweight Java with Interfaces	74

4.3.2	Generating the Constraints in the Example	80
4.3.3	Java Bytecode	83
4.4	Logical Reduction	86
4.4.1	Notation	87
4.4.2	Formalizing the Problem	87
4.4.3	The Generalized Binary Reduction	88
4.4.4	Our Progression	91
4.4.5	Running the Example	94
4.5	Experimental Evaluation	98
4.5.1	Experimental Setup	98
4.5.2	Analysis	100
4.6	Related Work	103
4.6.1	Input Reduction	103
4.6.2	Fuzz Testing	104
4.6.3	Input Generation and Internal Reduction	105
4.6.4	Debloating	106
4.6.5	Type-Safe Code Transformations	107
4.6.6	Search-Based Testing and Model Transformations	107
4.7	Summary	109
5	Conclusion	110

LIST OF FIGURES

2.1	A feature and a bug in WALA.	9
2.2	The first lost method	14
2.3	The code needed to turn Soot into a reachable methods analysis	18
2.4	Code to turn WALA into a reachable method analysis.	18
2.5	A histogram of the number of methods/program.	19
2.6	Metrics by easiness score.	21
2.7	Minimal examples of two bugs.	28
2.8	The probability of bug, unknown, and feature, given the easiness of the input.	29
3.1	A detailed run of the example using unmodified delta debugging	40
3.2	The dependency graph of our example	41
3.3	A run where all the invalid bytecode program inputs have been filtered out before execution (<code>verify</code>).	42
3.4	Three runs of <code>closure</code> the example in Figure 3.2.	46
3.5	Three runs of Binary Reduction (<code>binary</code>) on the example in Figure 3.2	50
3.6	Histograms on the metrics over the benchmarks	52
3.7	Cumulative frequency diagrams of different metrics.	55
4.1	The example input program	67
4.2	The variables and dependency constraints of the example	69
4.3	The dependency graph containing syntactic and referential dependencies	72
4.4	The syntax of Featherweight Java with Interfaces (FJI).	75
4.5	Our <i>reduce</i> function of FJI.	76
4.6	FJI helper rules.	78

4.7	FJI type rules.	79
4.8	The transposed graph used to generate the variable order	94
4.9	The initial run of LC_{\succeq} on our example	95
4.10	Two binary searches performed by GBR on our example	97
4.11	The distribution of benchmark over metrics	99
4.12	CFD's of metrics after reduction	101
4.13	The reduction over time	102

LIST OF TABLES

2.1	Relative precision and recall	20
2.2	Sources of features	23
2.3	The full list of bugs	25
2.4	The list of bug reports	26
2.5	The distribution of ground truth over the different intersections of static analyses	27
2.6	Distribution of ground truth on bug, feature, and unknown	29
2.7	Precision and recall	30
3.1	Aggregated results of all runs	55

VITA

2015–2015 Consultant, It-Minds, Copenhagen

2009–2015 Bachelor and Master of Science from DTU

CHAPTER 1

Introduction

Metaprograms are programs that take other programs as inputs. Many metaprograms are merely designed to interpret the input programs, like web-browsers, databases, and calculators, but others have been developed to extract knowledge from the input program. As it turns out, one of the favorite activities of developers is to make programs that help them create new programs. The most interesting question is if the program contains bugs; this happens if there exists a valid input to the program that makes the program produce unintended output. Therefore, they have designed metaprograms to prove that other programs do not contain bugs, in a process called verification, or find bugs in the programs, in a process called testing. Verification aims to prove the absence of bugs given *any* input, while testing aims to find a *single* input that produces a bug. The goal of verification is to guarantee the absence of bugs in the program. The goal of testing is to find bugs. When we find a buggy input, then together with the unintended output, we can create a bug report, which we use to debug the program.

Much like regular programs, metaprograms have bugs, but bugs in metaprograms are much more critical. For example, a bug in a compiler can affect all the programs that we compile with it. If we can reduce the number of bugs in a metaprogram, it has a ripple effect that affects all the programs they take as inputs. Metaprograms are complicated pieces of software by themselves, but since they also take programs as inputs, it becomes tough to report bugs. The critical problem is the bug might equally well be in the input program as in the metaprogram. So, even when presented with a bug report, a developer might reject it because they think the input is invalid or give up on it because the input is too

complex. Ideally, we would use verification or testing; however, these strategies work poorly for analyzing bugs reports in metaprograms, as we will now see.

Verification has been used to prove the correctness of some metaprograms. The most popular is CompCert [LBK16] that is a formally proven correct C-compiler written in Coq. However, writing the entire metaprogram in Coq is a huge endeavor, and most developers tend to use an automatic verification tool, like a static analysis instead. These tools are known to over-approximate the number of bugs found in a program. They do this so that they can soundly reject all programs that contain bugs. Recently in the Soundness Manifesto [LSS15], Benjamin Livshits et al. writes “[V]irtually all published whole-program analyses are unsound and omit conservative handling of common language features when applied to real programming languages.” In practice, any static analysis written for a real programming language is neither sound nor complete. When we want to analyze bug reports, this is not ideal: not all bugs will be reported, and those that are, are not necessarily real. Because metaprograms need to handle the full semantics of the input programs, they are often as complex as their inputs, and tend to use the language features that the static analyses under-approximate. We, therefore, tend to use testing when we want to find bugs in metaprograms.

We can manually sit down and test every input program which we can think of, and see if the metaprogram produces an intended output. This process is laborious and is often limited by the imagination of the developer. Instead, we use automatic techniques to generate inputs in a process called fuzzing [ZGB19]. Much work in testing has been focused on automatically figuring out if the output is intended, referred to as the *oracle validity problem*. However, there exist another problem. When analyzing metaprograms, fuzzing is at a disadvantage because the inputs which are programs are complicated. The input space is vast, and many configurations are invalid. For example, only a few sequences of bytes are valid Java Bytecode programs. Trying inputs at random increases the risk of choosing invalid inputs that produce undefined behavior in the metaprogram, and the undefined behavior might not be easy to detect. Yang et al. [YCE11] coined the term *Input Validity Problem*, when they used their fuzzer, C-Smith, to generate test programs for a C-compiler. They noticed that many of

the inputs programs they generated contained undefined behavior. Because programs with undefined behavior are unsuitable for bug-reports, they concluded they either had to *detect* or *avoid* these inputs. Therefore, the problem is not finding inputs that produce unintended behavior in metaprograms but to verify that they are bugs and ultimately have the bugs fixed.

Both automatic verification and testing has significant drawbacks when reporting bugs in metaprograms. In this dissertation, I turn the problem on its head and use verification techniques on the *input* instead of the *program*, to tackle the input validity problem straight on.

1.1 Thesis Statement

I propose a solution between general verification and testing. Instead of modeling the semantics of the metaprogram or trying all inputs, I suggest that we focus on bug reports. A bug report contains input that produce unintended output in the metaprogram. They can have been generated by a user of the metaprogram or by any of the existing fuzzing techniques. We can then statically model the validity of the inputs that have been found to produce unintended behavior in the metaprogram. By modeling the inputs' validity, we can check whether the inputs we have are valid, and we can reduce the input to generate a smaller valid input that produces the same unintended behavior in the program. The benefit of this approach is that we focus our statical analysis power on inferring the validity of the input, which is often more straightforward than modeling the correctness of the metaprogram.

Building custom analyses for the input of a single program is only useful to the developer. But the stakes are different for metaprograms. First, metaprograms often play a crucial role in the development chain. For example, a bug in a proof assistant brings all the proofs proved by it into question. So finding bugs in metaprograms have a more significant impact on the programming world at large. This can justify using more time designing custom made solutions for that set of metaprograms. Second, metaprograms often share input; a model of

the validity of a Java program can be used to find bugs in a Java-compiler and a Java static analysis.

There exist a vast literature on fuzz testing [YCE11, GKL08, GLM08, HHZ12, PLS19, ZGB19], which seeks generate new valid inputs to find unintended outputs in the program. In contrast, our approach is not to find new bugs but to verify existing bugs, and to help the developers understand them better by giving smaller examples of the same bug. Furthermore, previous validity models of inputs have mostly been focused on grammar-based validity [GKL08, HHZ12, SLZ18]. They did this because grammars can be used to generate new inputs, potentially finding all possible bugs. Because our goal is ultimately focused on a single possible bug, we do not have to generate entirely new inputs. Therefore, we can go beyond grammar-based validity and use more complex validity models with the power to model programs that do not have a grammar, like bytecode.

In summary, my thesis is:

Modeling the validity of inputs to a metaprogram using easiness, graphs, and logic improves the effectiveness of verifying and reducing bug reports.

To support my thesis, we will present three different, progressively more complicated models of the validity of inputs to metaprograms, which goes beyond grammars. All three examples focus on Java bytecode. We have chosen Java bytecode because it is a highly complex and widely used input, which input validity cannot be modeled using a grammar alone. Essentially, if the techniques work here, there is a good chance they will work everywhere.

The first part of the dissertation is about verifying bugs in static analyses. Ideally, if a static analysis fails to over-approximate the program's behavior, it would be considered a bug. The problem is that most static analyses are deliberately unsound if the input program contains hard to model features, like reflection. Thus, an unsoundness is only a bug if the static analysis did it by accident. In this case, we would say the input is valid if it does not contain hard to model features, namely that the input is easy to handle. The problem is that what features are hard to model is often not defined, and even those that are can be

hard to detect. To get around this, we model the validity, or easiness, using a consensus of static analyses. Contrary to previous work on differential testing [McK98], which focuses on differences in the outputs. We focus on differences in the inputs.

The second and third part is about input reduction. The goal of input reduction is that given a failure-inducing input to a program, produce the smallest sub-input such that it still produces the failure, which makes it easier to debug [ZH02]. The biggest problem in reduction is to avoid invalid inputs. When reducing inputs to metaprograms, avoiding invalid inputs becomes even harder. In metaprograms, there is much more structure required by the inputs. They have to pass parsing, type-checking, and sometimes substantial analysis. If the bug is buried in the metaprogram, building a valid input requires an exact model of the input.

The rest of the dissertation is structured like this:

- In Chapter 2, we model the easiness of inputs using a consensus of four static analyses. We define easiness as a continuous measure of the validity of an input to a static analysis. This enables us to verify or reject bug reports in four deliberately unsound static analyses.
- In Chapter 3, we model the internal dependencies between class-files. This way, we avoid invalid inputs and reduce failure-inducing inputs to a smaller final size, faster than `ddmin` [ZH02].
- In Chapter 4, we extend our granularity to also include methods and fields. To do this we have to model the internal dependencies using propositional logic. In total, we can generate more valid sub-inputs, which allows us to reduce programs even better.

Together, these three examples show that by modeling the input programs' validity we can verify and reduce bug reports for metaprograms, which we conclude in Chapter 5.

CHAPTER 2

Separating Bugs from Deliberate Unsoundness in Static Analyses

A static analysis may be unsound on an input program either because it has a bug or because its designer deliberately underapproximated a hard-to-model case like reflection. In other words, the input program is invalid if it contains hard-to-model features. This raises the question of how to distinguish bugs from deliberate unsoundness. If we have found a case of unsoundness, how do we know if the input is valid and that we can report it as a bug? We might compare the output with that of other analyses, but they are allowed to overapproximate differently. Thus, differences do not imply bugs. In this chapter, we present the first automated solution to this problem. Our approach uses an easiness analysis to define a classifier that separates bugs from deliberate unsoundness. In particular, if a static analysis is unsound in a case where the input is easy, the classifier will say that input is valid, and the static analysis has a bug. We have implemented our technique for four static analyses of Java, namely, Doop, Petablox, Soot, and WALA. Our experiments with 6,044 Java programs identified bugs with 90 percent precision and 79 percent recall. It leads to 12 bug reports that we have confirmed to be real bugs.

2.1 Introduction

How can we find bugs in a static analysis tool? A standard approach is to do *soundness testing*, which compares the results of the static analysis with the results of a dynamic analysis. The idea is that any run-time behavior noted by the dynamic analysis but missed

by the static analysis is evidence of *unsoundness*. Once we have an input program that induces such behavior, we may want to send it to the static analysis tool developers as part of a bug report. However, for Java, our experiments show that accuracy is poor: in many cases, the unsoundness is due to reflection. Reflection is a known challenge for static analyses and many tools deliberate underapproximate program behavior in its presence. The Soundness Manifesto [LSS15] from 2015 used the term *soundy* for such underapproximation-by-design and noted that every realistic whole-program analysis tool uses it. Such soundness is a *feature* because it can improve speed and precision. Thus, when faced with a report about deliberate unsound behavior, a tool developer is unlikely to change the tool.

In general, when a static analysis tool is unsound, it may be a feature or it may be a bug. Specifically, a tool may underapproximate some language constructs by design and underapproximate others by mistake. Either way, the tool’s output looks similar and the cases are hard to tease apart. In this chapter, we present the first approach that classifies the bugs and the features correctly with high probability. Our technique is accurate, general, and automatic. Specifically, for unsound outputs in our experiment, our classifier identifies the bugs with 90 percent precision. Ultimately, this approach can help developers of static analysis tools find bugs in their tools easily.

Akin to *differential testing* [McK98], we run multiple static analyses on the same input, but in contrast to differential testing, we don’t compare the outputs. The reason is that static analyses may produce different output for a wide variety of reasons, including that they underapproximate differently. Thus, *differences do not imply bugs*. Instead, we additionally do soundness testing and use the soundness information to build a classifier. Our classifier is based on a novel notion of *easy inputs*, which are inputs that most analyses get right in our experiments. Based on this idea, our classifier says that a static analysis has a bug if it *maps an easy input to an unsound output*.

Rest of the Chapter In Section 2.2 we use examples to illustrate the difficulty in differentiating bugs from features in soundy analyses. The following sections make the following

contributions.

- In Section 2.3 we show how to define a classifier of bugs and features for any kind of static analysis. Our approach defines a notion of *easiness* of an input, which is found by an *easiness analysis*, that requires no human intervention.
- In Section 2.4 we instantiate our technique for method reachability with four static analyses (Dooop [BS09], Petablox [MZN15], Soot [VGH00], and WALA [DFS15]) which we will use in our experiments.
- In Section 2.5, we use the easiness analysis we assign an easiness score to 6,044 Java programs from NJR [PL18]. We find that 82% of programs are considered easy, and for the remaining 1,031 programs for which at least one static analysis is unsound, our classifier predicts that 756 of those cases reveal bugs in at least one of the static analyses.
- In Section 2.6, we report on a our effort to establish the ground truth about bugs and deliberate unsoundness for our dataset. This effort resulted in 12 bug reports, which have been confirmed by the developers, and 6 different sources of features.
- In Section 2.7, we compare our results from Section 2.5 with the ground truth from Section 2.6. Our results indicate that unsound behavior on easy inputs implies bugs, and the resulting classifier identifies real bugs with 90% precision and 79% recall.

We compare with related work in Section 2.8 and we summarise our findings in Section 2.9.

2.2 The Challenge

Many applications of static analyses do not rely on soundness [LSS15], yet more sound results are welcome. Finding better ways to handle reflection and other edge cases is a research topic

```

public class Ex1 {
    public void main() throws Exception {
        Ex1.class.getMethod("target")
            .invoke(null);
    }
    public static void target() {
        System.out.println("Reached");
    }
    // ... 98 methods that are never called
}

public class Ex2 implements Runnable {
    public void main() throws Exception {
        java.awt.EventQueue
            .invokeLater(new Ex2());
    }
    public void run() {
        System.out.println("Reached");
    }
}

```

(a) A feature

(b) A bug

Figure 2.1: A feature and a bug in WALA.

in itself, so finding bugs can be seen as lower hanging fruit that can improve the soundness of a static analysis. Reporting bugs to static analysis developers is a straightforward way to help improve all tools that rely on them.

However, when we encounter a unsound behavior, how do know if it is a unintended bug, or an indented feature of the static analysis. In the rest of the chapter, we are going to denote deliberate unsoundness as *features*, while we use *bugs* to denote unintended unsoundness.

2.2.1 Examples

The programs in Figures 2.1a and 2.1b illustrate the challenge. Both programs use reflection, which most static analyses underapproximate.

First, we consider an example of a feature. In Figure 2.1a, class “Ex1” uses reflection to invoke the method “target” and print “Reached”. We used WALA [DFS15] to analyze class “Ex1” and got the result that “target” is unreachable. This might seem like a bug, but it turns out to be a feature. Intuitively, WALA’s design choice is to underapproximate the effect of “invoke” and effectively ignore that “invoke” will call a method. An alternative and sound approach would be to overapproximate the effect of “invoke” and say that “invoke” can call *any* method. In a program with 100 methods, of which 98 are never called, the result of the alternative approach is uninformative reachability information. Intuitively, WALA prefers to be unsound instead of useless.

Next, we consider an example of a bug. In Figure 2.1b, class “Ex2” calls “invokeLater”, which, at some point in the future, will call “run”. Method “invokeLater” uses reflection internally, and we can see that our two examples look somewhat alike. We used WALA to analyze class “Ex2” and got the result that “run” is unreachable. We contacted one of the WALA authors who confirmed that this is an unintended unsoundness. The problem stems from the fact that “invokeLater” is a method in a standard library. For many such methods, WALA represents their behavior, but it missed “invokeLater”. Our experiments found many such unintended unsoundnesses; they totaled over half of 710 bug-inducing programs.

The two examples suggest that distinguishing the features from the bugs is hard, especially if the static analysis is a black box. While our examples used reflection (Java’s reflection interface has 181 public methods [LSV17]), Java has several other hard-to-model cases, including dynamic class loading, proxies, serialization, runtime compilation, native methods, unsafe, invokedynamic, and instrumentation that modifies the bytecodes after static analysis [DSR17]. All of these constructs are candidates for deliberate underapproximation.

2.2.2 The Concept of a Useful Classifier

Our challenge is to design and implement a classifier that divides input programs that produce unsoundness in a static analysis into bugs and features. We want the classifier to be *useful*, which means that it is accurate, general, and automated.

Accurate We want a classifier to accurately label the potentially many unsoundness inducing programs as features or bugs. If the classifier is inaccurate, we will have little confidence in the bug reports.

General We want a classifier to work with each static analysis as a black box. Thus, the classifier should make no assumptions about the design and implementation of the static analysis. Rather, the classifier should apply to different kinds of static analysis, to different static analyses of the same kind, and to different versions of a single static analysis.

Automatic If we do automated testing on many input programs to reveal unsoundness, we will want the classifier to be automated too, to save time.

2.2.3 Our Solution: Easiness Analysis

Several existing approaches find bugs by comparing the output of tools with the same purpose, namely differential testing [McK98] and N-version programming [CA76, Avi85]. These approaches rely on an assumption that bugs will deviate from the typical behavior of a collection of programs. Finding bug is as easy as comparing the outputs of many programs.

Static analyses can produce different outputs even on simple inputs; For example one static analysis could predict that any of the 98 methods in Figure 2.1a is reachable, and it would still be working as it should be. Differences do not imply bugs.

We turn differential testing inside out. We collect the results of many static analysis on a given input and use it to determine how easy the inputs is. We call this *easiness analysis*. The more analyses that are able to be sound on the input, the “easier” the input must be.

Using this knowledge we can classify bugs as unsoundness on easy inputs and features as unsoundness on hard inputs. Intuitively, if every other analysis is able to be sound, then the unsound analysis should have been too. We will describe how this works in the next section.

2.3 Our Classifier

In this section we present the design of our classifier. Our starting point is a set F of static analyses for which each $f \in F$ has a function UN SOUND_f . We will examine each of those in turn and then define our classifier.

We model a static analysis as a function that maps a program to information about the program. The static analyses in F must be of the same kind, that is, they must have the same functionality. For example, an analysis may map each C program to per-statement information about live variables, map each JavaScript program to per-method information

about side effects, or map each Java program to per-class information about whether its objects are confined to the enclosing package. Given a set of analyses of the same kind, we can run the analyses on the same input programs.

The function UNSOUND_f must satisfy the following property:

$$\begin{aligned} \text{UNSOUND}_f & : \text{Program} \rightarrow \{0, 1\} \\ \text{UNSOUND}_f(x) = 1 & \Rightarrow f(x) \text{ is unsound information.} \end{aligned}$$

Intuitively, UNSOUND_f records that a static analysis f misses behavior that a program could exhibit when run with x . Typically, an implementation of UNSOUND_f uses testing to demonstrate that the program x has a behavior that is missed by f . Such testing tends to be incomplete, so above we use a one-directional implication, which is sufficient for our purposes.

The idea behind our classifier is that if a static analysis maps an *easy* input to an unsound output, then it has a bug. What is an easy input? We address this by defining a function easiness_F that maps an input to a score that ranges from hard (0) to easy (1). Notice that we use the entire set F of static analyses to define easiness. Akin to differential testing, we run each analysis in F on the same input, but in contrast to differential testing, we don't compare the outputs.

Our definition of the easiness counts the analyses that map an input to unsound output. Specifically, given a set F of static analyses, we define the easiness of an input to be a value in the interval $[0, 1]$, where higher means easier.

$$\begin{aligned} \text{easiness}_F & : \text{Program} \rightarrow [0, 1] \\ \text{easiness}_F(x) & = 1 - \frac{1}{|F|} \sum_{f \in F} \text{UNSOUND}_f(x). \end{aligned}$$

Intuitively, the more static analyses produce unsound information, the harder the input is. For example, if all the analyses all produce unsound information for x , then the sum above

is 1, so $easiness_F(x) = 0$, and the input is hard. Dually, if none of the analysis produces unsound information for x , then the sum above is 0, so $easiness_F(x) = 1$, and the input is easy. Note that in case we want to differentiate the static analyses in F , we can generalize the formula for $easiness_F$ to use a weighted sum.

Now we can formalize the idea that if a static analysis maps an easy input to an unsound output, then it has a bug:

$$\begin{aligned} \text{bug}_F & : (F \times \text{Program}) \rightarrow \text{Boolean} \\ \text{bug}_F(f, x) & = \text{UN SOUND}_f(x) \wedge \text{easiness}_F(x) \geq \varphi \end{aligned}$$

Intuitively, if f maps x to unsound output and the easiness of x is above a threshold φ , then f has a bug. As an example of how to pick the threshold φ , consider a set F of size n as well as $f \in F$ where $\text{UN SOUND}_f(x)$. If we want $\text{bug}_F(f, x)$ to mean that all the other analyses in F map x to sound output, then we can pick $\varphi = \frac{n-1}{n}$.

The design of our classifier is general and can be applied to any set F of static analyses of the same kind and any function UN SOUND_f that satisfies our requirement.

2.4 An Instance of the Challenge

In this section we present an instance of the challenge to separate bugs from deliberate unsoundness. Our experiments focus on this instance.

Our choice of F Our set F consists of four static analyses that answer the question of *which methods in the application are reachable from main?* We picked this problem because it is fundamental: the reachable methods are the nodes of the call graph. So, we can view the problem of finding the reachable methods as a subtask of constructing the call graph. Bugs in the call graph analysis affect all other analyses that rely on the call graph.

Specifically, our set F of static analyses consists of Doop [BS09], Petablox [MZN15], Soot

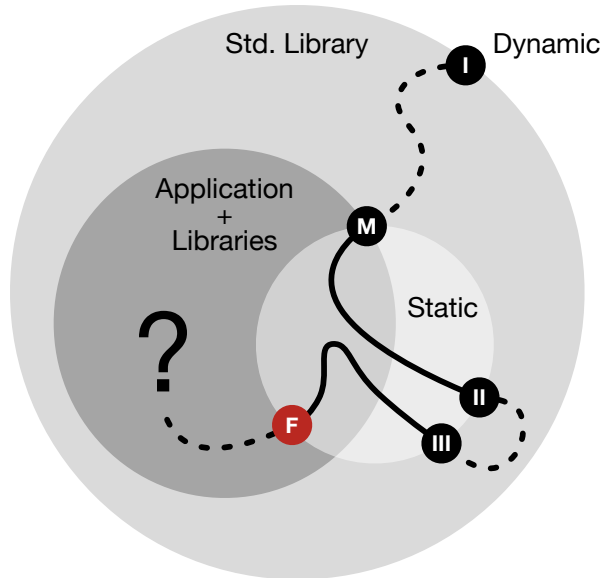


Figure 2.2: The first lost method (F) is the first executed method in the application that was missed by the static analysis.

[VGH00], and WALA [DFS15]. Among those, Doop, Soot, and WALA are widely used tools. We picked settings of the tools such that they all use a variant of 0-CFA and do rather little to analyze reflection.

Our choice of UN SOUND_f Our implementation of UN SOUND_f is a modification of the dynamic analysis Wiretap [KP18]. Wiretap is a good fit for our purposes because it instruments methods dynamically (using a Java agent). This means that even in the presence of reflection, it will log events for every method. Additionally, Wiretap avoids instrumenting its code, and java and sun classes. We modified Wiretap to take static information about reachable methods as input and monitor whether the methods that it enters are in that static information.

The first lost method In Section 2.3 we defined an easiness score for the entire input program. For our particular instance of the challenge, which focuses on methods, we refine the easiness score to apply to individual methods in the input. For simplicity we will report on the easiness score of a single method in each program: the first executed method that was

missed in the static information. We call this method *the first lost method*. This method is of particular interest because it marks the first point in the execution for which the static analysis and the dynamic analysis disagree.

Figure 2.2 illustrates how execution begins in the standard library (I), proceeds to the main method (M), executes other code in the standard library (II and III), and eventually executes the first lost method (F). Notice that (II and III) are not considered lost because they are part of the standard library. Notice also that other methods, executed after (F), is maybe missed by the static analyses. This can indeed be *because* the first-lost method was missed.

Example Consider again example “Ex2” from Figure 2.1b. WALA reports that the reachable methods are main, the empty constructor method, and 1,275 standard library methods, However, a dynamic analysis will note that the `run` method is also executed, so `run` becomes the first lost method. Now the challenge is: was the `run` method missed because of a bug or a feature in WALA? Our experiments show that Doop, Petablox, and Soot all agree that `run` is reachable. Thus, with 3 against 1, the easiness score is 0.75, which is high, so we conclude that WALA has a bug, which turns out to be correct.

2.5 Experiments

In this section we introduce our dataset of Java programs and we show our results from doing easiness analysis based on static and dynamic analysis.

2.5.1 Static Analyses

Doop. Doop is a static analysis tool that uses Datalog as a basic engine to run the static analysis. It runs datalog using Souffle. We are using doop version “4.10.11” downloaded

from the git repository¹ at commit “cdc59ce7”. We made a small modification such that the analysis was pure and did not write to its build directory. Doop uses library definitions, called platforms, that are stored in another repository², we used commit “5e0b6a87”.

We use their “context-insensitive” analysis and no reflection handling:

```
doop --platform java_8 -a context-insensitive -id 0 --main <mainclass> -i <classpath>
```

Petablox. Petablox also uses datalog as its basic engine, but runs it using the bddbldb tool. We downloaded Petablox from their git repository³ at commit “b95fd275”. We ran petablox like this:

```
java -Dpetablox.work.dir='pwd' petablox.project.Boot
```

With a “petablox.properties” in the working directory similar to:

```
petablox.datalog.engine=bddbldb
petablox.main.class=<mainclass>
petablox.run.analyses=reachable-methods
petablox.jvmargs=-Xmx4096m
petablox.class.path=<classpath>
petablox.reflect.kind=none
```

Soot. While Petablox and Doop are built on top of Soot, also Soot itself comes with a built-in call graph construction. Soot is an analysis framework so we had to write our own extension (see Figure 2.3) to make Soot print the reachable methods.

We used version “3.1.0”, which we downloaded from the official repository⁴. We ran soot using this command:

¹<https://bitbucket.org/yanniss/doop.git>

²<https://bitbucket.org/yanniss/doop-benchmarks.git>

³<https://github.com/petablox/petablox.git>

⁴<http://soot-build.cs.uni-paderborn.de/nexus/repository/soot-release/ca/mcgill/sable/soot/3.1.0/soot-3.1.0-jar-with-dependencies.jar>

```
java -cp=soot.jar:ext SootReachableMethod \  
-p cg.spark on -pp -w -f n -app -cp <classpath> <mainclass>
```

WALA. We used WALA version “1.5.0” and we downloaded `shrike-1.5.0.jar`, `core-1.5.0.jar`, and `util-1.5.0.jar` from the maven repository⁵.

WALA is a static analysis framework, so we have to build our own code to make WALA print the reachable methods. (see Figure 2.4).

After communication with the authors of WALA, we used the following exclusion file. Any class that matches any of the regexes in the following file is excluded:

```
com\sun\.*  
sun\.*  
org\netbeans\.*  
org\openide\.*  
com\ibm\crypto\.*  
com\ibm\security\.*  
org\apache\xerces\.*
```

Using this code we can now execute WALA:

```
java -cp core.jar:util.jar:shrike.jar:ext WalaReachableMethod \  
-exclude "excludes.txt" -classpath <classpath> -mainclass <mainclass>
```

2.5.2 Setup and Dataset

Setup We ran experiments on 4 servers with 24 cores at 2.10GHz each and around 188Gb of ram. The servers ran NixOS (<http://nixos.org/nix>). All experiments are with Java 8 and written as nix scripts to be reproducible. The experiments were distributed over the different servers with one of the servers dedicated as the master. We ran the experiments in batches of 15 (10 on the master server). We ran each static analysis for maximum of 1,800

⁵<http://central.maven.org/maven2/com/ibm/wala/>

```

import java.util.*; import java.io.*; import soot.*;
public class SootReachableMethod {
    public static void main(String[] args) {
        PackManager.v().getPack("wjtp").add(
            new Transform("wjtp.reachable-methods",
                new SceneTransformer() {
                    protected void internalTransform(String phase, Map<String, String> options) {
                        try {
                            Iterator x = soot.Scene.v().getReachableMethods().listener();
                            BufferedWriter writer = new BufferedWriter(
                                new FileWriter("reachable-methods.txt"));
                            while (x.hasNext()) {
                                writer.write(x.next().toString() + "\n");
                            }
                            writer.close();
                        } catch (IOException e) { e.printStackTrace(); }
                    }
                }
            ));
        soot.Main.main(args);
    }
}

```

Figure 2.3: The code needed to turn Soot into a reachable methods analysis

```

// import wala files
public class WalaReachableMethod {
    public static void main(String[] args)
        throws WalaException, IllegalArgumentException, CancelException, IOException {
        Properties p = CommandLine.parse(args);
        String classpath = p.getProperty("classpath"),
            mainclass = p.getProperty("mainclass"),
            exclude = p.getProperty("exclude");
        ClassHierarchy cha = ClassHierarchyFactory.make(
            AnalysisScopeReader.makeJavaBinaryAnalysisScope(classpath, new File(exclude)));
        Iterable<Entrypoint> entrypoints = Util.makeMainEntrypoints(scope, cha,
            "L" + mainclass.replaceAll("\\.", "/"));
        AnalysisOptions options = new AnalysisOptions(scope, entrypoints);
        options.setReflectionOptions(AnalysisOptions.ReflectionOptions.NONE);
        CallGraphBuilder builder = Util.makeZeroCFABuilder(Language.JAVA, options,
            new AnalysisCacheImpl(), cha, scope);
        Iterator<CGNode> it = builder.makeCallGraph(options, null).iterator();
        FileWriter fw = new FileWriter("reachable-methods.txt");
        while (it.hasNext()) {
            IMethod m = it.next().getMethod();
            /* print method to fw */
        }
        fw.close();
    }
}
}

```

Figure 2.4: Code to turn WALA into a reachable method analysis.

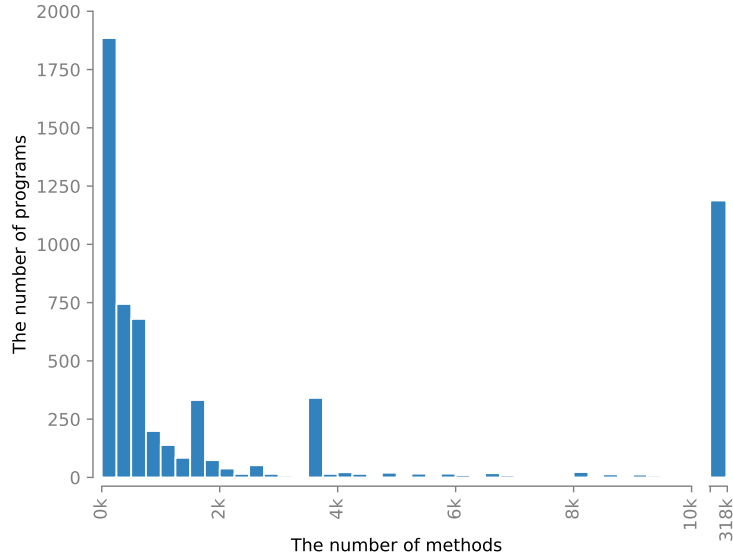


Figure 2.5: A histogram of the number of methods/program.

seconds on each program, and we converted the analysis outputs to a common format that enables easy comparison. We ran Wiretap for no more than 7 minutes on each program.

The Dataset Our dataset is 6,044 Java programs that we got from the NJR project [PL18]. NJR is a project that contains 100,000 Java executable programs scraped from GitHub. We selected 10,000 programs at random from the set and then filtered on the criteria that each program executes at least two application methods and that every static analysis returns at least one reachable method. The selected programs are diverse. Figure 2.5 illustrates the number of methods in the application and in third-party libraries, but excludes methods in the standard library. Notice that many of the programs are small, yet we see a long tail of large projects. The median number of methods is 662. The smallest program has 9 methods, while the biggest has 317,868 methods. In total those programs have 71,971,612 methods.

2.5.3 Quality of the Static Analysis

In the dataset of 6,044 Java programs, we found 1,031 programs for which at least one static analysis produces unsound results, compared to the output of Wiretap.

analysis	$ SA $	$ SA \cap W $	relative	
			precision	recall
WALA	523 696	83 266	15.9%	77.3%
Doop	1 016 708	86 137	8.5%	80.0%
Petablox	765 621	80 727	10.5%	75.0%
Soot	1 807 013	86 460	4.8%	80.3%
union	1 823 903	87 198	4.8%	81.0%
intersection	442 051	78 077	17.7%	72.5%

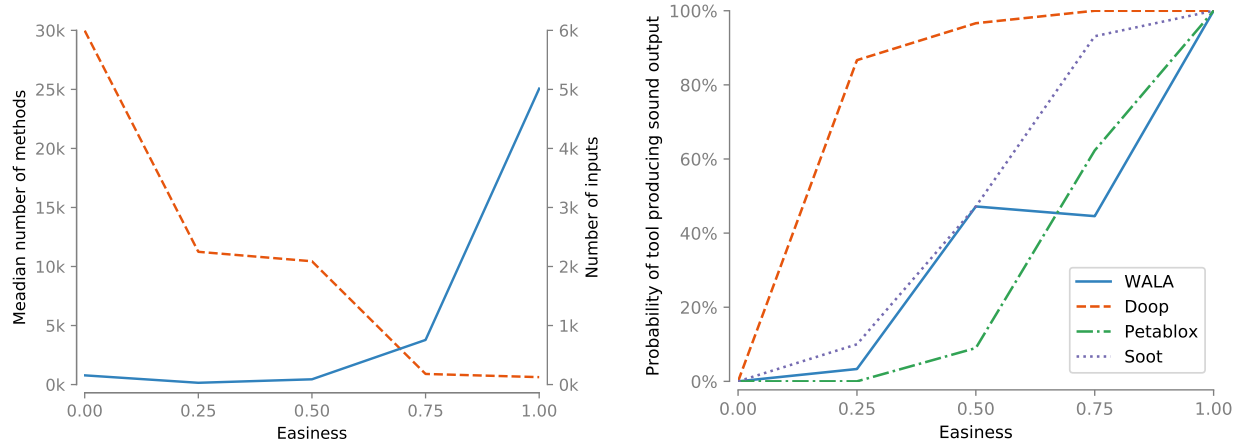
Table 2.1: Relative precision and relative recall for four static analyses and their combinations. The size of W (the output of the dynamic analysis) is 107,697.

To determine the quality of the static analysis we use *relative precision and recall*. Precision and recall is normally computed based on ground truth, while in our case we use the output of Wiretap, which underapproximates the ground truth about reachable methods.

Table 2.1 gives the relative precision and relative recall for the four static analyses. If we order the analyses by relative precision, then WALA is best, then Petablox, followed by Doop, and finally Soot. In contrast, if we order the analyses by relative recall, then Soot is best, then Doop, followed by WALA, and finally Petablox. Table 2.1 also shows results for the union and for the intersection of the outputs of the four static analysis.

From the low relative precision (4.8–15.9%) we conclude that, relative to the runs done by Wiretap, every static analysis produces a large number of false positives. We believe that those false positives are due, in part, to the poor code-coverage of the dynamic analysis (0.15%). In contrast, from the much higher relative recall (75.0–80.3%), we conclude that the analyses find high percentages of the executed methods. Additionally, if we combine the static analyses, we get even better relative recall (81.0%), and if we take the intersection we get an even better precision (17.7%).

For our purposes, the four static analyses produce sufficiently many unsound outputs for us to experiment with our classifier.



(a) The number of input programs (blue, solid) and the median number of methods per input (orange, dashed).

(b) The probability that a tool is sound.

Figure 2.6: Metrics by easiness score.

2.5.4 Easiness Analysis

We are now ready to do our easiness analysis. For each input we calculate the easiness by the formula described in Section 2.3. This produced an easiness score for all 6,044 programs. The results can be summed up in two plots. For each input we can plot the occurrences of inputs and the median number of methods in those inputs over easiness score, as seen in Figure 2.6a. We can see that most of the inputs are easy (1.00: all the static analyses were sound), some are medium hard (0.25–0.75) (at least one static analysis was sound), and then finally a small amount of inputs were hard (0.00: no static analysis was sound). The graph speaks to the maturity of the static analyses we are checking, as most of the programs are easy. We can also see that the median number of methods per input is decreasing as the inputs gets easier. This makes sense because bigger programs contain more code and the chance that they will use the hard-to-model cases of the language is greater.

The second plot (Figure 2.6b) shows the probability that any of the given static analysis will be sound given the easiness of the inputs. In general, the static analyses shows a trend toward being more sound on easier inputs, which is what we would expect. We think the dip in WALA’s performance from 0.5 to 0.75 is an indicator of some kind of bug which is

especially easy to fix.

The easiest inputs for which at least one single static analysis is unsound is a group of 756 programs that have an easiness score of 0.75. We expect those programs to have the highest ratio of bugs to features. The hardest group of programs (with an easiness score of 0) contains 156 inputs. We expect those programs to have the highest ratio of features to bugs. In the next section we will establish ground truth about bugs and features, and then in Section 2.7 we will evaluate our classifier based on that ground truth.

2.6 Ground Truth

In this section explain how we established a ground truth about bugs and features. Our approach is domain-specific and uses both static and dynamic information.

2.6.1 Static and Dynamic Analysis

Our static analysis scans Java bytecode to identify the use of certain language constructs and get information about the class hierarchy. We used static information to categorize bugs.

Our dynamic analysis records two aspects of the execution. First, it records the stack trace at the point of entry to the first lost method. Second, it records the returned objects and methods of calls outside the context of the application. The stack trace is the best way to find the context in which the method was missed. For example, if a method is called using reflection, then a reflection method will appear on the stack frame. The log of objects returned from methods outside the scope can be used to detect indirect loss of context. Specifically, some static analyses might lose track of objects sent to the standard library, and then later, these objects might be called, and the static analysis would miss it.

feature	hard / all
Excluded by WALA	27 / 45
Indirect reflection	2 / 2
java/lang/Class.forName	20 / 22
java/lang/ClassLoader.loadLibrary	0 / 2
java/lang/reflect/Constructor.newInstance	51 / 51
java/lang/reflect/Method.invoke	40 / 40
<i>sum</i>	140 / 162

Table 2.2: Known sources of features, and the number of unsoundnesses from the hard category (easiness = 0) and everything, that were attributed to them.

2.6.2 Feature Detection

We detected features (deliberate unsoundness) mainly based on the stack trace at entry to the first lost method. If the stack map contained reflection or known excluded classes, we classified the case as a feature. While direct use of reflection was the common case, we also found two cases of indirect reflection. Indirect reflection happens when a program calls “Class.newInstance”, and then precedes to call a method on the created object. If that method was lost by a static analysis, then we mark it as indirect reflection. Table 2.2 describe the features we found.

2.6.3 Bug Detection

Bug detection was an intense manual effort with some tool support. First we used the easiness score from the previous section to identify a list of programs that are likely to reveal bugs. Then our effort to detect bugs proceeded in three stages:

- *Inspect.* We take the program with the fewest number of methods and manually inspect it.
- *Minimize.* If we detect a bug, then we produce a minimal program that reproduces the bug and we report it to the developers of the tool.

- *Categorize.* Then we use our knowledge of the bug to build a recognizer that categories if the bug exist by inspecting the static and dynamic information available. We then mark all matches as bugs and remove them from the list.

Some bugs have complex origins and were beyond our capability to detect precisely, so our recognizer is imperfect. In 4 cases, both feature detection and bug detection kicked in, and in those cases, we classified the cases as bugs.

2.6.4 Bug Reports

We found a total of 12 bugs which we have reported to the developers of the tools. Table 2.3 presents a complete list of all the bugs, and Table 2.4. All bugs were confirmed as bugs by the developers or fixed in later versions. For each bug we give a short description of the cause of the bug and a high level description of the recognizer we built.

Bugs in WALA In the easiest unsound category for WALA, there exists 419 programs. Of those we were able to categorize 402 as bugs and 17 as features which was due to the excluded classes of WALA. We found 4 distinct bugs. Three of the bugs were about missing approximations of commonly used methods in the standard library.

Bugs in Petablox Of the 285 easy unsound programs for Petablox we were able to confirm 229 of them as bugs. We found 6 distinct bugs, but were unable to classify the remaining 56 programs.

Petablox has a range of bugs: Some of them are about the standard library and some of them are not bugs of omission, but bugs in the code of Petablox. A good example is the indirect implements bug (10): Petablox misses methods that implements a method in an indirect interface. This is best illustrated in the example in Figure 2.7b. Petablox missed that an interface can extend other interfaces. This meant that when the static analysis saw the call to “I”s “notfound”, it concluded no classes implemented “I”, even though “Main”

	ID	easy / all	description	recognizer (The bug is recognized if..)
WALA	1	2 / 2	A class extends a class that is excluded, the class itself is also excluded.	The first lost method is contained in a class that extended an excluded class.
	2	2 / 2	“addShutdownHook” from “Runtime” can call the “run:()V” method of the argument when Java is shutting down.	The first element on the stack is a run method and there is a “addShutdownHook” in the reachable methods of WALA.
	3	396 / 400	The bug shown in Figure 2.1b. “invokeLater” from “EventQueue” is not modeled.	The bottom element in the stack is an “EventDispatchThread” (which “EventQueue” wraps all its threads in) [†] .
	4	2 / 11	“setDefaultUncaughtExceptionHandler” special case. The uncaught exception handler is called when the program is ended by an uncaught exception.	The first lost method is “uncaughtException” from “DefaultExceptionHandler”.
Petablox	5	99 / 99	It does not find “toString” methods to be reachable when called from “print” and “println”.	The first lost methods is a “toString” method and it had a print method on the stack.
	6	55 / 55	Some methods from the Executors library.	The first lost method is a “run” or “call” method, and “runWorker” from “ThreadPoolExecutor” is on the stack.
	7	24 / 27	It lose the context of the object if put into a multi-array.	The object of the first lost method that can be placed in a multi-array somewhere in the code [†] .
	8	39 / 41	Methods called on objects that are returned by an iterator.	The object of the first lost method at some point is returned by “Iterator.next()”.
	9	2 / 2	Methods called on objects that are returned by “Queue.peek”.	Same as above
	10	10 / 10	Does not handle transitive interface implementations.	The first lost method implements an indirect interface [†] .
Soot	11	6 / 15	Does not handle the new “default” keyword In Java 8.	The first lost method is implemented in an interface.
	12	46 / 46	Missed class instantiation when a child class initializer does not exist.	The first lost method is a class initializer and there exist a child class that has no class initializer [†] .

Table 2.3: The full list of all the bugs found and what we did to categorize them. The easy / all column indicate the number of bugs we recognized in the set of inputs with easiness = 0.75 and in all inputs respectively. † means that our recognizer slightly overapproximates.

ID	Bug Report	Response
1	https://github.com/wala/WALA/issues/364	Fixed
2	https://github.com/wala/WALA/issues/368	Fixed
3	https://github.com/wala/WALA/issues/372	Unintended unsoundness [†]
4	https://github.com/wala/WALA/issues/369	Fixed
5	https://github.com/petablox-project/petablox/issues/20	Categorized as bug
6	https://github.com/petablox-project/petablox/issues/25	Categorized as bug
7	https://github.com/petablox-project/petablox/issues/26	Categorized as bug
8	https://github.com/petablox-project/petablox/issues/27	Categorized as bug
9	https://github.com/petablox-project/petablox/issues/28	Categorized as bug
10	https://github.com/petablox-project/petablox/issues/22	Fixed
11	https://github.com/Sable/soot/issues/1054	Fixed
12	https://github.com/Sable/soot/issues/1055	Categorized as bug

Table 2.4: The full list of bug reports. IDs correspond to the bugs in Table 2.3. † indicates that the bug was confirmed in a personal correspondence with the author of the tool.

indirectly does.

Bugs in Soot We were able to confirm all the 52 easy unsound programs for Soot as bugs. We found 2 distinct bugs.

The bugs were interesting because they were very different from the bugs of Petablox and WALA. An example of Bug 12 can be seen in Figure 2.7a. Here, Soot does not find the class initializer of “SuperClass” because it is only ever called when “Main” is initialized. Soot did not model that even if the class initializer of a child does not exist, the class initializer of the parent might still be called. If the commented line is uncommented in the example then Soot finds that the class initializer of “SuperClass” is reachable.

Bugs in Doop We found no easy unsound programs for Doop.

2.6.5 Results

Table 2.5 sums up our data about ground truth. Here, we can see the distribution of the ground truth over the intersection of the different static analyses. In summery, out of the

WALA	Doop	Petablox	Soot	easiness	ground truth			total
					bugs	features	unknown	
•	•	•		0.75	52	0	0	52
•	•		•	0.75	229	0	56	285
•	•			0.50	0	0	39	39
•		•	•	0.75	0	0	0	0
•		•		0.50	0	0	0	0
•			•	0.50	0	0	3	3
•				0.25	0	1	0	1
	•	•	•	0.75	402	17	0	419
	•	•		0.50	8	0	0	8
	•		•	0.50	2	2	35	39
	•			0.25	4	0	22	26
		•	•	0.50	0	0	0	0
		•		0.25	0	0	0	0
			•	0.25	1	2	0	3
				0.00	12	140	4	156
<i>sum</i>					710	162	159	1031

Table 2.5: The distribution of the ground truth of the unsound programs over the different intersections of static analyses. “•” indicates where the static analysis did not miss the first lost method.


```

public class Main implements J {
    public static void main(String args[]) {
        I x = new Main();
        x.notfound();
    }
    public void notfound() { System.out.println("notfound"); }
}
interface I { public void notfound(); }
interface J extends I { }

```

(a) Bug 12.

```

public class Main extends SuperClass {
    // static int x = 1;
    public static void main(String[] args) {}
}
class SuperClass {
    static { System.out.println("Hello World"); }
}

```

(b) Bug 10.

Figure 2.7: Minimal examples of two bugs.

1,031 programs that induces unsoundness in the static analyses we found 710 programs that induce real bugs (68.9%), 162 programs that induce features (15.7%), while the reasons for unsoundness induced by the remaining 159 programs (15.4%) remain unknown.

2.7 Evaluation

Now we are ready to evaluate our classifier. Our main claim is that, within the scope of separating bugs from deliberate unsoundness in static analysis:

a classifier based on easiness analysis is useful.

In particular, we find that using our easiness analysis as the basis of a bug classifier gives a 90.3% precision and 78.6% recall.

Returning to Section 2.2.2, we asserted that a useful analysis has to be accurate, general, and automatic. We will now evaluate our classifier in those dimensions.

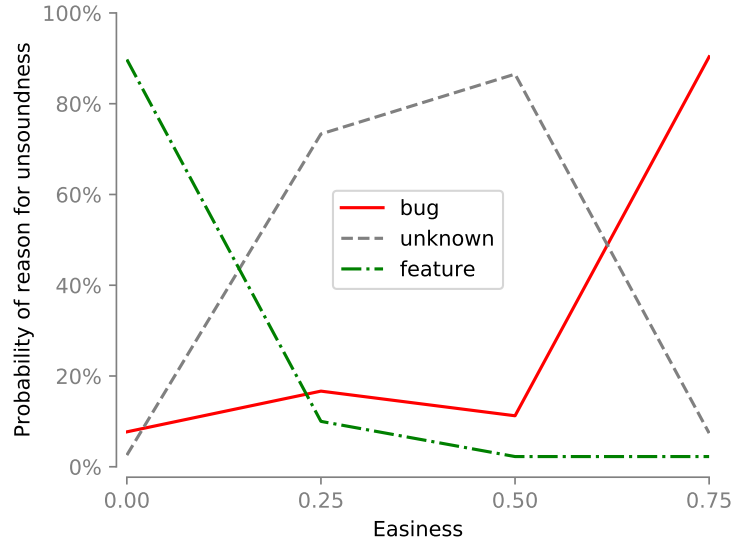


Figure 2.8: The probability of bug, unknown, and feature, given the easiness of the input.

		ground truth				
		cut-off	bug	feature	unknown	total
bug	≥ 0.75		683	17	56	756
feature	< 0.25		12	140	4	156
unknown			15	5	99	119

Table 2.6: This table is like Table 2.5, but is grouped by the classifications made after easiness analysis.

Accurate First, we will investigate accuracy. By accumulating Table 2.5, we can plot the chance to find a bug, feature, and unknown of our ground truth as a function of the easiness of the inputs in Figure 2.8. We can see that most of the features are grouped around the hard inputs, and most of the bugs are grouped around the easy inputs. The unknown inputs are most likely in the medium hard inputs. Thus, easiness analysis gives a much better chance at identifying bugs and features than picking at random. If we picked at random, our ground truth says that we should expect to find a distribution of 15.7% features and 68.9% bugs at every easiness level.

As we discussed in Section 2.3, we can exploit easiness analysis as a classifier by defining

	precision	recall
bug	90.3%	78.6%
feature	89.7%	43.6%

Table 2.7: Precision and recall. The precision is calculated in relation to the ground truth. Recall is calculation is including the programs with unknown unsoundness.

a threshold. By choosing the threshold (≥ 0.75) for bugs and (< 0.25) for features, we can construct Table 2.6, and then use our ground truth to calculate the precision and recall for bug and feature detection. In Table 2.7, we can see that our classifier finds bugs with 90.3% precision and 78.6% recall, and features with 89.7% precision and 43.6% recall. We would get even higher recall if we only count the categorized bugs and features, but because we used the easy inputs for categorizing the bugs, this would unfairly skew the results.

Let us compare the above accuracy results against a simpler approach that detects reflection, like before, and classifies everything else as bugs. This simpler classifier says that 117 cases of reflection are features and that the remaining 914 case of unsoundness are bugs. Now the precision of detecting bugs is only 77.7%. In summary, our approach is accurate.

General If we inspect the hard cases only, the easiness analysis was able to categorize 96.5% (Table 2.2) of the reflection as hard problems without having any knowledge of reflection. This indicates that we can extract knowledge about hard inputs without knowing anything the language and its constructs. The only known features that were categorized as bugs were due to explicit excluded classes of WALA.

Our definitions of easiness and of bugs based on easiness are language independent and independent of the kind of static analysis. In summary, our approach is general.

Automatic We ran our easiness analysis on 6,044 programs with no human intervention. In summary, our approach is automatic.

2.8 Related Work

In this section, we will go over the most recent related work in differential testing and classification of unsoundness in static analyses.

Input classification using differential testing Recently, Zhu et al. used differential testing in a novel way [ZHQ18] by testing two browsers with a *known* difference. Specifically, one of the browsers has an adblocker and the other has no adblocker. Their goal was to find “input” that reveals the difference, namely a website with an anti-adblocker. Their approach analyzes the two execution traces and gets a higher rate of detecting such websites than previous work.

Bug finding by voting Engler et al. [ECH01] presented an approach to bug finding that looks for contradictions in how a program is written. Specifically, if the program access particular data in one way in the majority of cases but different in other cases, likely the deviant case is a bug.

Differential testing has been used before with programs as inputs in the tool CSmith by Yang et al. [YCE11]. Like our work, they work with soundy tools, namely compilers. In their case, the compilers are expected to produce the same code as long as the input is well defined. Contrary to our work, the hard cases are found using static analysis, and the outputs are unknown. In our approach, the hard cases are unknown, but the expected output is known.

Differential testing of tools with programs as inputs were also used in a paper by Paleari et al. [PMF10], where they used a differential testing technique, called N-version Disassembly. The focus of the technique is to compare the output of n disassemblers. The input can be invalid, but the disassemblers are well defined for all inputs, so they are always supposed to produce the same answer. They use a partial oracle to weed out decompilers that are wrong so that they do not affect the voting.

Differential testing is closely related to N-version programming [CA76, Avi85]. The concept is that “Bugs are deviant behavior”, which means that we can recover the correct behavior by majority vote. Criticism of this statement was made by Knight and Leveson [KL86], who showed that humans often make the same mistakes when coding. Our technique is not immune to this effect, yet the consequences are less severe. If multiple static analyses make the same mistake, we will assign that input a lower easiness score. This will not affect the quality of our results but might reduce the number of bugs we find.

Analysis of reflection For Java, many static analyses approximate the targets of reflection calls, often via string analysis [LWL05, LTS14, LTX15, SKB15].

Bodden and his co-authors [BSS11] presented the TamiFlex tool that enables a static analysis of Java to be sound with respect to a set of recorded program runs. TamiFlex enforces soundness dynamically: it checks at run time whether the program calls a method that is unknown to the static analysis.

For JavaScript, static analysis is more difficult than for Java [FSS13]. Richards, Hammer, Burg, and Vitek [RHB11] presented statistics on the strings that were passed to calls of `eval` in a large number of JavaScript applications. Later, Meawad, Richards, Morandat, and Vitek [MRM12] presented tool support for removing uses of `eval` from a program. Such studies could be useful also for Java reflection.

Contrary to these techniques, our tool does not care if the program reflection, only if the input is hard to do static analysis on. While using a reflection detector can predict if the input program contains reflection, which is hard to model, it does not help with native calls and other dynamic features of Java.

Classification with a known outcome Reif et al. [RKE18] did soundness testing of Soot and WALA. Specifically, they designed a test suite that enabled them to test the soundness of Soot and WALA. They identified reflection and missing support for Java 8 as sources of unsoundness. Each program in their test suite was designed to reveal either a bug (missing

support for Java 8) or a feature (underapproximation of reflection). Thus, their approach classifies inputs as revealing bugs or features up front.

Sui et al. [SDE18] presented a carefully designed microbenchmark for dynamic language features of Java. They used Doop, Soot, and WALA to analyze their microbenchmark and found that all three analyses produce unsound results. Their microbenchmark reveals many cases of deliberate unsoundness in a static analysis while it makes no attempt to reveal bugs.

In contrast to both techniques, our approach takes any input program for which a static analysis is unsound and classifies the unsoundness as a bug or a feature.

Manual classification Lhotak [Lho07] presented a tool for Java that compares a static call graph produced by Soot (in the Spark configuration) to a dynamic call graph. For a single benchmark, his tool found 143 methods that were executed but missed by the static analysis. He did a manual inspection of the calls to those 143 methods and produced a list of classes that the execution had loaded using reflection. He made no attempt to classify the unsound results into bugs and features, while this is what our approach can do.

Andreasen et al. [AMN] used soundness testing for JavaScript along with a methodology for how to identify the root cause of unsound output from a static analysis. Their approach uses delta debugging [ZH02] on the unsoundness-inducing input to produce a smaller program with the same property. While delta debugging makes classification easier, their approach leaves a manual effort to understand whether the root cause is a feature of a bug. In contrast, our approach will do this automatically.

Classification via program instrumentation Christakis, Müller, and Wüstholz [CMW15] compared the output from the Clousot static analyzer of CIL bytecodes [ECM06] with the output from a dynamic analysis. They also presented a tool for instrumenting the input program such that it tracks all deliberate unsoundness in Clousot. The instrumentation enabled the authors to evaluate whether their six benchmarks violate Clousot’s unsound assumptions. They found that the assumptions of the static analysis were violated in 2–26

percent of the methods during execution. However, those violations led to no mistakes by the client of the static analysis, which was an assertion checker. The authors focused entirely on known features of the static analysis and made no effort to find bugs in the static analysis. In contrast, our approach can classify unsound results into bugs and features.

Unsoundness due to incomplete input Xue and Ngyuen [XN05] presented a static analysis of incomplete Java programs. Their analysis produces both static information and a list of call sites for which the static information may be unsound. They focused entirely on unsoundness that is due to an incomplete input program rather than due to bugs or features in the static analysis.

2.9 Summary

We have shown that easiness is a continuous indicator of the validity of inputs to static analysis. And we have used it to verify the correctness of 683 bug reports. Our easiness-based classifier can, with 90 percent precision and 79 percent recall, determine if an input that produces unsoundness is valid. Our approach is an accurate, general, and automatic way of classifying bugs and deliberate unsoundness in static analyses. One could imagine that a static analysis developer amended their bug tracker with an automatic easiness analysis. This would automatically assign an easiness score to each reported bug, which indicates if the bug should be taken seriously or rejected.

Our approach has great potential because it assumes no domain-specific knowledge. Future work may evaluate the approach on other approximative metaprograms and types of input programs. Finally, more work is needed to significantly reduce the size of bug-revealing inputs before using them in bug reports. Input reduction is precisely the topic of the next two chapters.

CHAPTER 3

Binary Reduction of Dependency Graphs

Delta debugging is a technique for reducing a failure-inducing input to a small input that reveals the cause of the failure. This has been successful for a wide variety of inputs, including C programs, XML data, and thread schedules. However, for input with many internal dependencies, delta debugging produces many invalid inputs and scales poorly. Such input includes programming languages like C#, Java, and Java bytecode, and they have presented a significant challenge for input reduction until now. This means that input reduction for metaprograms for these kinds of input programs have been under-explored.

In this chapter, we show that the core challenge is a reduction problem for dependency graphs, and we present a general strategy for reducing such graphs. We combine this with a novel algorithm for reduction called Binary Reduction in a tool, called J-Reduce, for Java bytecode. Our experiments show that our tool is 12x faster and achieves more reduction than delta debugging on average. This enabled us to reduce bug reports significantly for three Java bytecode decompilers, which we reported to the developers.

3.1 Introduction

Delta debugging automates a process that programmers otherwise do by hand. When a program crashes on an input, the programmer tries to understand the cause of the crash by *reducing* the input. Intuitively, the programmer can cut the input in half and see if one of the two halves causes the crash as well. After some repetitions of this step, the input may be small enough for the programmer to spot the cause of the problem. Delta debugging

executes a more advanced version of this, automatically. For example, delta debugging can map the original input to a nonconsecutive subsequence. Thus, delta debugging relieves programmers from the tedium of reducing and executing, and lets them focus on improving their programs.

In their seminal paper on delta debugging, Zeller and Hildebrandt [ZH02] showed successful experiments in which the inputs were C programs, Mozilla user actions, and UNIX commands. Other papers have reported on experiments with XML data [MS06a], thread schedules [CZ02], and event sequences [HBB15]. The problem of reducing failure-inducing input to a minimal size is NP-complete [MS06a], and for an input with n characters, trying all 2^n substrings may be futile. Instead, the delta debugging algorithm `ddmin` [ZH02] tries $O(n^2)$ substrings. This led to massive success but when most natural subsets of the input are invalid, most iterations of `ddmin` fail and are of no help towards reduction. As a step towards scalability, Zeller and Hildebrandt showed how `ddmin` does better when applied to a list of *lines*. This is better than a character-oriented approach because often a line of code represents a syntactic element such as a statement. Mishserghi and Su [MS06a] went further and introduced hierarchical delta debugging (HDD) that works with the syntactic structure of the data. For example, for reduction of a method body, HDD represents the body as a list of *statements* and runs `ddmin` on the list. This is better than a line-oriented approach because a statement can span multiple lines. The use of the syntactic structure increase the chance that each input is syntactically valid and increases the chance that each run produces useful information.

In this chapter, we consider the next level of difficulty, which arises when elements of the syntactic structure have many *internal dependencies*. Such input includes C#, Java, and Java bytecode, where a class may depend on other classes and where compilation and bytecode verification require all dependencies to be present. We can represent such a program as a list of classes and run `ddmin` on the list, yet most runs will fail because the input is invalid. We solve this by modeling the internal dependencies in the input as a *dependency graph* and then running reduction on a list of transitive closures in the dependency graph.

We will show experiments with reduction by both `ddmin` and a novel algorithm called Binary Reduction.

In the remainder this chapter, Section 3.2 introduces the challenge in detail, after which Sections 3–6 present our contributions:

- We show that dependency graphs are a convenient data structure for reduction, particularly by `ddmin` (Section 3.3).
- We present a new reduction algorithm, called Binary Reduction that runs only $O(n \log n)$ iterations (Section 3.4).
- We evaluate on 238 Java bytecode programs that induce failures in three decompilers. Binary Reduction on graphs is 12x faster and reduces more than `ddmin` (Section 3.5).
- We submitted bug reports for the decompilers (Section 3.6).

Finally, Section 3.7 discusses related work, and we summarize the chapter in Section 3.8.

3.2 The Challenge

We will explain the challenge of reducing input with dependencies with an example. The example concerns the Java bytecode decompiler called CFR (<http://www.benf.org/other/cfr/>). CFR takes as input a valid Java bytecode program and decompiles it to a Java source program. This is useful for programmers who want to inspect and reason about libraries that have been shipped as bytecode. Ideally, a decompiler produces source code that can be compiled to bytecode such that the input bytecode and output bytecode are behaviorally equivalent. When we look for bugs, we will use a more modest quality measure: a decompiler should produce source code that compiles. If CFR maps a valid bytecode program to a source program that doesn't compile, we say that *CFR fails*.

We define a valid bytecode program as a set of class-files that each individually verifies and depends only on classes in the program itself or in the standard library. A class A

depends on another class B if A mentions B anywhere in its bytecode. This can happen in many places, such as in an extends-clause, in a type annotation, in a new-expression, or in a type cast.

Our example begins with the discovery of a bug in CFR. We ran CFR on a valid Java bytecode program with 17 classes and then we ran `javac` on the produced source program, which led to this error message from `javac`:

```
... error: illegal start of expression
if (var2_3.hasNext()) ** break;
```

Now we would like to send a bug report to CFR, but it can be hard to locate the bug in 17 classes. In this chapter, we focus on reducing the bytecode program to one with *a subset* of the classes that still induces CFR to fail with the same bug report. Thus, the reducer *picks* classes without *changing* them.

The task of reducing a set of classes to a smaller set of classes is of the kind for which delta debugging usually excels. We implemented the delta debugging algorithm called `ddmin` by Zeller and Hildebrandt [ZH02] such that it works on a list of classes. However, the result of reducing our Java bytecode program with 17 classes was disappointing: the result was a program with 14 classes.

Figure 3.1 illustrates our run of `ddmin`. The boxes and \times 's show which classes were input to an iteration of `ddmin`, while the column labeled fail shows whether CFR failed (marked with *yes*), succeeded (marked with *no*), or whether the bytecode program was invalid (marked with *?*). In most cases, the input bytecode program is invalid so to highlight the few steps with valid inputs, we use boxes in those steps. Specifically, when CFR reproduces the bug we use \square , and in all other cases with valid inputs we use \blacksquare .

The many iterations with invalid bytecode programs inputs are of no help towards reduction. Additionally, each invocation of CFR and `javac` can take between a couple of seconds and multiple minutes, which decreases scalability.

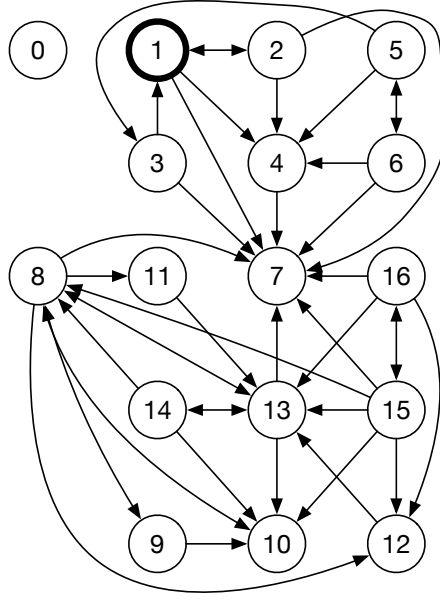


Figure 3.2: The dependency graph of our example program. The nodes are classes in the program and the edges represent references to other classes. The class marked 1 induces the bug in the decompiler.

Regehr et al. [RCC12a] identified this kind of problem in 2012 and called it the *test-case validity problem*. They also identified two kinds of solutions, namely:

1. detect invalid inputs or
2. avoid invalid inputs.

In the context of C, Regehr et al. [RCC12a] used two tools to *detect* invalid code, which led to an excellent reducer. However, they left *avoiding* invalid code as an open problem.

Inspired by the success of Regehr et al. [RCC12a], our first attempt to improve the situation was to *detect* invalid bytecode programs. Specifically, we enhanced `ddmin` to a version called `verify` that checks, in every iteration, that the bytecode program is valid before running CFR and `javac`. Given that the original bytecode program is valid and that each class stays unchanged, a check of whether a bytecode program is valid boils down to checking that all dependencies are present. We do this by going through each class to find its dependencies, after which we assemble the dependencies into a graph.

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	fail
□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	.	.	<i>yes</i>
■	<i>no</i>
.	■	<i>no</i>
.	□	□	□	□	□	□	□	□	□	□	□	□	□	□	.	.	<i>yes</i>
.	■	<i>no</i>
.	□	□	□	□	□	□	□	□	□	□	□	□	□	□	.	.	

Figure 3.3: A run where all the invalid bytecode program inputs have been filtered out before execution (`verify`).

Figure 3.2 shows the dependency graph for our example; each node represents a class and each edge represents a dependency. The classes are numbered from 0 to 16 (corresponding to numbers in Figure 3.1), for simplicity. The edge $1 \rightarrow 4$ means that class 1 depends on class 4. Sometimes classes are tightly coupled, in that case bidirectional edges are possible. Using this graph, `verify` can check for each iteration that all the dependencies are present before running CFR. Figure 3.3 shows that `verify` invokes CFR and `javac` just five times, yet still produces a program with 14 classes. In Section 3.5, our experiments show that `verify` is 3x faster than `ddmin` on a list of classes, on average.

We have found that the actual error is induced by class 1, marked in bold in Figure 3.2. However, for a bytecode program with class 1 to be valid, classes 2, 4, and 7 also have to be present. Thus, the smallest valid input that induces an error in CFR is $\{1, 2, 4, 7\}$, which is 3.5x smaller than the result given by `ddmin` and `verify`. This raises the question: why do `ddmin` and `verify` do poorly and what can we do about it?

The problem has to do with a lack of *monotonicity* that we explain now. In our example, consider the two valid, failure-inducing inputs $\{1, 2, 4, 7\}$ and $\{0, 1, \dots, 16\}$. Figure 3.3 shows many sets S where

$$\{1, 2, 4, 7\} \subseteq S \subseteq \{0, 1, \dots, 16\}$$

and in every case, S is invalid bytecode. Thus, we don't have the property that as inputs get bigger, failure is preserved. Equivalently, we don't have the property that as inputs get smaller, nonfailure is preserved. In other words, when we run CFR followed by `javac` on possibly invalid bytecode, this combined operation fails to be monotonic.

The lack of monotonicity has a big effect on the reduction process. Specifically, the process can move from a failure-inducing input such as $\{0, 1, \dots, 16\}$ to a smaller input such as $\{0, 1, \dots, 8\}$ that induces no failure, and still miss the even smaller, failure-inducing input $\{1, 2, 4, 7\}$. For example, if we from $\{0, 1, \dots, 8\}$ remove some classes that had missing dependencies, the removal may make the input valid again, hence make the failure reappear.

We note that the original paper on `ddmin` assumes that “*failure is monotone*” [ZH02, Section VIII]. However, delta debugging has been successful even when monotonicity fails, including when the input is a C program, so what is different about our case? The answer is that

for input with many internal dependencies, monotonicity can fail spectacularly.

Indeed, Figure 3.1 shows that almost every subset is invalid bytecode so trying $O(n^2)$ subsets among the (2^n) possible subsets has little chance of success.

Notice that in Figure 3.1, `ddmin` managed to remove the interdependent classes 15 and 16 in a single step. This was mostly due to a lucky ordering of the classes. We can see from this example that for `ddmin` to remove interdependent classes, it must remove them at the same time. An attempt to remove either one in a single step would run into invalid bytecode.

For another example, notice that if we remove class 11 from $\{0, 1, \dots, 16\}$, we get an invalid bytecode program. By inspecting Figure 3.2 we can see that in addition to removing class 11, we also have to remove class 8, thus also class 13, and so on. For `ddmin` to have a chance to remove such a long dependency chain in a single step, we would need the classes to be ordered in a particularly fortunate way. However, given that the reduction problem is NP-complete, finding a good listing is a hard problem.

The above analysis has led us to abandon the idea of *detecting* invalid input and instead pursue how to *avoid* it. We will present a new approach that avoids invalid bytecode programs entirely by putting dependencies front and center. The key idea is to do reduction of dependency graphs, as we explain next.

3.3 Reduction of Dependency Graphs

In this section, we will distill the essence of reducing an input with internal dependencies in a way that avoids invalid inputs. Thus, we will run CFR followed by `javac` only on valid bytecode. Hence, all remaining violations of monotonicity, in the sense of Section 3.2, come from CFR and `javacc`, and those tend to be insignificant.

Let us assume that the validity of an input can be modeled with a dependency graph. If all elements in the input have no missing dependencies, the input is valid. Intuitively, if we group all elements with their dependencies, and with the dependencies of their dependencies, and so on, then picking such a group would be a valid input. For the case of a set of verified classes, no missing dependencies mean a valid bytecode program.

We avoid invalid inputs by changing the reduction problem from working with a list of elements to working with a list of *sets of elements*. We ensure that each such set of elements is a valid input by requiring that it is a self-contained subset without missing dependencies. Those subsets are the *transitively closed* subsets of nodes in the dependency graph of the input. Recall that the transitive closure (or simply *closure*) of a set of nodes is the smallest superset that is transitively closed.

Our reduction strategy is based on the idea that *a set of closures represents the union of those closures*, which makes sense because the union of two closures is itself a closure. This means that no matter what subset of the list of closures a reduction algorithm picks, the union of that subset would be a closure. And since every closure is valid input we have a strategy that *avoids* invalid inputs.

The dependency graph reduction strategy Here is our strategy for reduction of an input with internal dependencies:

1. Map the input to its dependency graph.
2. Compute the closure of each node.

3. Form a list of the closures.
4. Run a reduction algorithm on the list of closures.
5. Output the union of the reduced list of closures.

For our dependency graph in Figure 3.2, Step 2 maps the 17 nodes to the following 8 different closures: $S_1 = \{7, 8, \dots, 16\}$, $S_2 = \{7, 8, \dots, 14\}$, $S_3 = \{1, 2, \dots, 7\}$, $S_4 = \{1, 2, 3, 4, 7\}$, $S_5 = \{1, 2, 4, 7\}$, $S_6 = \{4, 7\}$, $S_7 = \{7\}$, and $S_8 = \{0\}$. We have fewer closures than nodes because of cycles in the graph.

The above strategy leaves two aspects to be refined. First, in Step 3 we must decide how to order the closures. This turns out to be important, as we will discuss below. Second, in Step 4 we must decide how to reduce the list of closures. We have some freedom here because when we remove a closure from the list, the result is again a list of closures. Thus, a reduction algorithm can remove closures and avoid invalid subsets entirely.

Using `ddmin` on a list of closures In this section we use `ddmin` in Step 4 and refer to this algorithm as `closure`. In each iteration of `ddmin`, the union of the closures represents a valid bytecode program so all we need to do is to check whether CFR fails.

Figure 3.4a shows two runs of `closure` on the closures of the nodes in Figure 3.2. The difference lies in how we ordered the closures up front; any ordering is possible. In both cases, the number of iterations is much smaller than in the run of `ddmin` shown in Figure 3.1. The reason is that `closure` encounters no invalid subsets so it homes in on a solution in just four and five iterations, respectively. In this example `closure` run as fast as `verify`, in Figure 3.3, but the first run returns $S_3 = \{1, 2, \dots, 7\}$, which is much better than the output in Figure 3.3, which is $\{1, 2, \dots, 14\}$. The second run returns $S_5 = \{1, 2, 4, 7\}$, which is the best possible subset.

For the second run in Figure 3.4a, we sorted the closures by size, from smallest to largest. The reason this works well has to do with a quirk in running `ddmin` on a list of sets. Specifically, `ddmin` views S_3 , S_4 , S_5 as equally good reduced sets, because each one is a

S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	fail
□	□	□	□	<i>yes</i>
■	■	<i>no</i>
.	.	□	□	<i>yes</i>
.	.	□	<i>yes</i>
.	.	□	

S_7	S_8	S_6	S_5	S_4	S_3	S_2	S_1	fail
■	■	■	■	<i>no</i>
.	.	.	.	□	□	□	□	<i>yes</i>
.	.	.	.	■	■	.	.	<i>no</i>
.	■	■	<i>no</i>
.	.	.	.	■	.	.	.	<i>no</i>
.	■	.	.	<i>no</i>
.	■	.	<i>no</i>
.	■	<i>no</i>
.	□	□	□	<i>yes</i>
.	■	.	.	<i>no</i>
.	■	<i>no</i>
.	□	.	□	<i>yes</i>
.	■	.	.	<i>no</i>
.	■	<i>no</i>
.	□	.	□	

(a) Two runs. The first run has the closures in an arbitrary order; the second has them sorted after size.

(b) Failure induced by $\{1, 12\}$.

Figure 3.4: Three runs of `closure` the example in Figure 3.2.

single closure. Whether `ddmin` produces one of S_3 , S_4 , and S_5 depends on the order of the closures, and, like in section 3.2, some orders are more lucky than others. In each of the two runs in Figure 3.4a, `closure` produced a single closure and behaved like binary search. We know that `ddmin` tends to process input from left to right, as we can see in Figure 3.1, so when we sort the closures by size we have a good chance to get the smallest closure.

The algorithm `closure` leaves room for improvement. For example, suppose the failure is induced by the combination of class 1 and class 12 (rather than only class 1). The smallest reduced set is $S_5 \cup S_2$, which has size 11. However, Figure 3.4b shows a run of `closure` that produces the larger set $S_3 \cup S_1$, which has size 16. Like before, this happens because `ddmin` looks for the smallest possible set of sets without considering the sizes of those sets. Here, ordering the closures by size is insufficient to get the best result. In the next section, we present an algorithm that matches `closure` in simple cases and is better and faster in general.

3.4 Binary Reduction

We will extend the classical input reduction problem with a notion of *cost* that can model sizes of closures, and we will present an algorithm called Binary Reduction.

3.4.1 The Weighted Input Reduction Problem

For all sets we can refer to the elements using indices: e.g. if A is a set, then $A_1, \dots, A_{|A|}$ refer to the elements of A . If I is a set, then 2^I is the powerset of I . $\mathbf{1}$ is true and $\mathbf{0}$ is false. We say that $P : 2^I \rightarrow \text{Bool}$, a predicate on subsets of I , is *monotonic* if $S_1 \subseteq S_2$ implies that $P(S_1) \Rightarrow P(S_2)$.

We recast reduction as a decision problem:

Definition 1 (Weighted Input Reduction Problem). *Given (I, P, C, k) , where I is failure inducing input, $P : 2^I \rightarrow \text{Bool}$ is a polynomial-time monotonic predicate which is true for all failures ($P(I) = \mathbf{1}$), $C : 2^I \rightarrow \mathbb{N}$ is a polynomial-time cost function, and $k \in \mathbb{N}$ is a natural number, decide $\exists S \subseteq I : P(S) \wedge (C(S) < k)$.*

Intuitively, P represents buggy software and I represents failure-inducing input. We follow the convention of Mishserghi and Su [MS06a] that P returns $\mathbf{0}$ both in the case of invalid input and in the case of no failure. The novelty in the above definition is the cost function C . With a cost function we are allowed to use more complicated inputs. Later, in Chapter 4 (Section 4.4), we will define a version of the Input Reduction Problem which uses logic to define the valid inputs. In order to define the problem as a decision problem, we use the standard technique of asking whether the cost of S exceeds a threshold k .

Many instantiations are possible, including the following three.

Original Problem In the seminal delta debugging paper [ZH02], I is the index set of the input list and $C(S) = |S|$. The problem is to find the smallest subset of the index set that induces a failure.

Set of Sets If we want to minimize the *union* of a set of sets, we can pick $I = 2^E$, where E is a set and $C(S) = |\cup S|$. Additionally, we can lift any predicate $Q : 2^E \rightarrow Bool$ on subsets of E to a predicate $P : 2^I \rightarrow Bool$ by defining $P(S) = Q(\cup S)$.

Dependency Graphs The previous section explains how to do reduction of a dependency graph by mapping it to the *Set of Sets* problem above. The idea is to compute the closures of the nodes in the dependency graph and then to find the smallest union of the closures that satisfies the predicate.

Misherghi and Su [MS06a] proved that the hierarchical delta debugging problem is NP-complete. In a similar manner, we prove that the Weighted Input Reduction Problem is NP-complete.

Theorem 1. *The Weighted Input Reduction Problem is NP-complete.*

Proof. The Weighted Input Reduction Problem is in NP because, given a witness $S \subseteq I$, we can check in polynomial time that $P(S) \wedge (C(S) < k)$ since P and C runs in polynomial time.

We show that the Weighted Input Reduction Problem is NP-hard by reducing from the Hitting Set Problem, which is NP-complete [Kar72]. The Hitting Set Problem is: given (Σ, Z, k) , where $Z \subseteq 2^I$ is a set of sets, decide $\exists S \subseteq \cup_i Z_i : (\forall i : S \cap Z_i \neq \emptyset) \wedge (|S| < k)$. The reduction works as follows. Define $I = \cup_i Z_i$ and $P(S) = (\forall i : S \cap Z_i \neq \emptyset)$ and $C(S) = |S|$. Notice that P is monotonic. Notice also that if (I, P, C, k) has a solution S , then $S \subseteq I$, $P(S)$ and $C(S) < k$, which means that $S \subseteq \cup_I Z_i$, $\forall i : S \cap Z_i \neq \emptyset$, and $|S| < k$. This means that S is also a solution to (Σ, Z, k) . □

Our problem is NP-complete and, unless $P = NP$, the best we can do in polynomial time is an approximation. In the next section, we will present a polynomial-time approximation algorithm for solving the input reduction problem.

Algorithm 1: Binary Reduction

Input: (I, P, C, k) where $P(I)$
Output: If there exist an S st. $P(S)$ and $C(S) < k$
Define: $A \preceq_{S'} B := C(S' \cup \{A\}) \leq C(S' \cup \{B\})$
Data: A current solution $S \leftarrow \emptyset$ and sorted search space $D \leftarrow \text{sort}_{\preceq_S}(I)$
while $\neg P(S)$ **do**
 $r \leftarrow \min r$ st. $r > 0 \wedge P(S \cup \{D_j : j \leq r\})$
 $S \leftarrow S \cup \{D_r\}$
 $D \leftarrow \text{sort}_{\preceq_S}(\{D_j : j < r\})$
end
return $C(S) < k$

3.4.2 The Binary Reduction Algorithm

Recall that in Figure 3.4b, `closure` makes a bad choice and rejects the left half of the input. The closures on the left are insufficient to induce a failure. Instead, `closure` finds a much worse solution among the bigger closures on the right. The reason is that `closure` takes no advantage of getting a sorted input list.

We introduce Algorithm 1, an algorithm called Binary Reduction. The algorithm is inspired by the second run of `closure` in fig. 3.4a, where `ddmin` operates like a binary search and quickly finds a single closure. Binary Reduction extends this idea to work in cases where multiple closures are required. We use $(\text{sort}_{\preceq_S}(X))$ to denote the sorting of X according to the total order \preceq_S and $(\min r \text{ st. } p)$ to denote finding the smallest r such that p is satisfied.

The idea is to maintain two sets S and a D . Here, S is the set of elements that we know are in the final set, and D is a sorted set of elements still to be searched. We initialize S to be empty, indicating we know nothing, and we initialize D to be the sorted input, as we want to search the entire space.

The algorithm starts by testing if the current solution satisfies $P(S)$. If not then we search for the minimal prefix of a sorted listing of D that together with S satisfies P . Since we know that $S \cup \{D_j : j \leq r\}$ is the smallest prefix that satisfies P , we also know that removing D_r from the sets would make P false, therefore D_r must be part of the final set.

S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	fail
.	<i>no</i>
□	□	□	□	<i>yes</i>
■	<i>no</i>
■	■	<i>no</i>
□	□	□	<i>yes</i>
<hr/>								
.	.	⊗	<i>yes</i>
<hr/>								
.	.	⊗	

S_7	S_8	S_6	S_5	S_4	S_3	S_2	S_1	fail
.	<i>no</i>
□	□	□	□	<i>yes</i>
■	<i>no</i>
■	■	<i>no</i>
■	■	■	<i>no</i>
<hr/>								
.	.	.	⊗	<i>yes</i>
<hr/>								
.	.	.	⊗	

(a) The first run has $C(X) = |X|$; the second has $C(X) = |\cup X|$.

S_7	S_8	S_6	S_5	S_4	S_3	S_2	S_1	fail
.	<i>no</i>
■	■	■	■	<i>no</i>
■	■	■	■	■	■	.	.	<i>no</i>
□	□	□	□	□	□	□	.	<i>yes</i>
<hr/>								
.	⊗	.	<i>no</i>
■	■	■	.	.	.	⊗	.	<i>no</i>
□	□	□	□	□	.	⊗	.	<i>yes</i>
□	□	□	□	.	.	⊗	.	<i>yes</i>
<hr/>								
.	.	.	⊗	.	.	⊗	.	<i>yes</i>
<hr/>								
.	.	.	⊗	.	.	⊗	.	

(b) Failure induced by $\{1, 12\}$ and with $C(X) = |\cup X|$.

Figure 3.5: Three runs of Binary Reduction (**binary**) on the example in Figure 3.2

We can therefore add D_r to S and reduce the search space to the smaller prefix without D_r , $\{D_j : j < r\}$. We continue to reduce the search space until the $P(S)$. Since we only added elements to the solution that were required and since P is monotonic, the solution is a *local minimum*: if any elements are removed from S , then P is no longer satisfied.

The core of the algorithm is the search for the smallest prefix of D that satisfies P . In general, this takes $O(n)$ time, where n is the size of the search space. However, we have a monotonic P so

$$P(\emptyset) \Rightarrow P(\{D_1\}) \Rightarrow P(\{D_1, D_2\}) \Rightarrow \dots \Rightarrow P(\{D_1, D_2, \dots, D_n\})$$

and thus we can use binary search.

The final touch is to keep the set D sorted, using the cost function C . Our idea is to use the cost function to sort the search space such that low-cost elements are chosen early.

This is a greedy algorithm that makes the best pick possible in each iteration. As with other greedy algorithms, this may fail to produce the best global solution, yet our experiments show that the results are good in practice. Notice that every iteration sorts D according to the cost of the union of the currently selected set S and the individual inputs. This is an advantage because sometimes the cost of the union of two input sets does not equal the sum of the cost of each of the sets.

We will use Binary Reduction in Step 4 of the strategy in Section 3.3; we refer to this algorithm as **binary**. The first diagram in Figure 3.5a shows a run of **binary** on the example in Figure 3.2, with a natural cost function $C(X) = |X|$. We use \boxtimes to mark the D_r that is added to the solution in the line $S \leftarrow S \cup \{D_r\}$. Like in the first run of delta debugging over the closures of the graph in Figure 3.4a, we get S_3 , which is suboptimal. However, in contrast to **closure**, we can easily modify **binary** to use a more interesting cost function, like the number of elements in the union of the sets $C(X) = |\cup X|$. Figure 3.5a shows that run. Like in the second run of **closure** in Figure 3.4a, we get S_5 , which is the best solution.

Figure 3.5b shows a run of **binary** on the example in Figure 3.2, but this time the failure is induced by $\{1, 12\}$. In contrast to the run of **closure** in Figure 3.4b, we get the best solution $S_5 \cup S_2$. Notice that the run took only 3 binary searches and 9 invocations of P .

Even though the choice of $C(X) = |\cup X|$ solves our problem, we could imagine more interesting cost functions like the total size of classes. Binary Reduction greedily choose a local minimum regardless of cost function, but we expect that it performs best if the cost function is monotone in the size of X .

Complexity analysis The complexity of Binary Reduction depends on the complexity of the cost function C ($\$C$), the predicate P ($\P), the size $n = |I|$ of the input, and the final size s of the reduction. We do at most s binary searches, with $O(\log n)$ invocations of P and worst-case n calculations of $C(S)$ as part of sorting (assuming caching) which takes

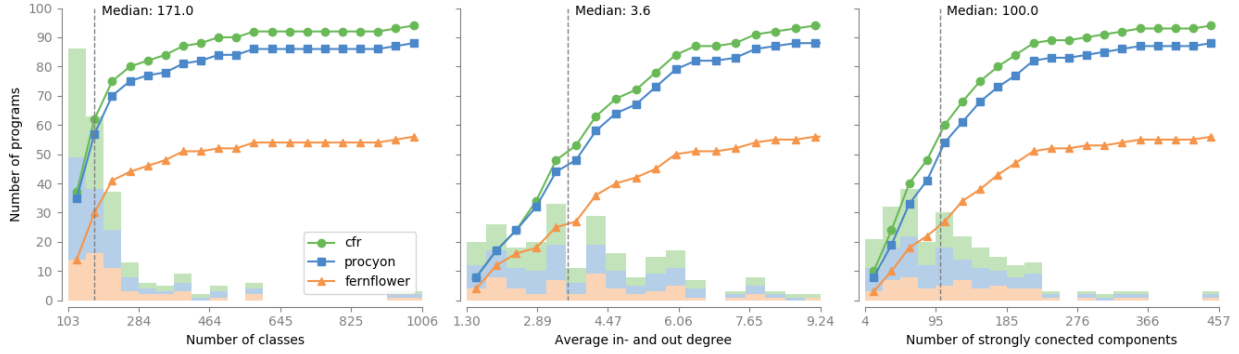


Figure 3.6: These three histograms show the distribution of the failure inducing inputs over three metrics; number of classes, the average in- and out degree of the underlying dependency graph, and the number of strongly connected components.

$O(n \log n)$ time. So in total we have

$$O(s (\log n \cdot \mathbf{\$P}(n) + n \cdot \mathbf{\$C}(s) + n \log n)).$$

Inspecting the time complexity of the algorithm we can see that we will make at most $O(s \log n)$ invocations of P . Since s is bound by n , the complexity of the algorithm is $O(n \log n)$ iterations.

3.5 Experimental Results

In this section, we present an empirical evaluation of using dependency graphs and Binary Reduction for reduction. We have implemented those techniques in a tool for Java bytecode programs called J-Reduce. J-Reduce is a general tool for reducing bytecode programs while preserving errors. We will use three decompilers as part of the evaluation, yet any tool that takes Java bytecode as input could have been used. The evaluation supports the two main claims of the chapter:

- (1) **Reduction based on a list of closures is faster and better than reduction based on a list of classes.** When we run `ddmin` on a list of classes, we time out 75% of the

runs after an hour. In contrast, when we run `ddmin` on a list of closures, we time out only 9% of the runs after an hour. Including the timeouts, the list-of-closures approach gives 7x speedup and 1.07x smaller results, on average.

(2) Binary Reduction is faster and better than `ddmin`. Only 1% of the runs of Binary Reduction on a list of closures time out after an hour. Including the timeouts, Binary Reduction gives 1.7x speedup and 1.15x smaller results, on average, compared to running `ddmin` on the same input. Overall, we get 12x speedup and 1.24x smaller results compared to running `ddmin` on a list of classes.

3.5.1 Experimental Setup

Implementation J-Reduce has a single frontend that extracts a dependency graph from binary Java class files. Specifically, J-Reduce scans through each class-file to search for references to other classes and assembles them into a dependency graph. The common frontend and backend enable easy comparison of the algorithms. J-Reduce implements four different reduction algorithms:

- `ddmin`: Classical delta debugging on a list of classes.
- `verify`: Uses `ddmin` plus detection of invalid bytecode.
- `closure`: Uses `ddmin` on a list of closures, sorted after size.
- `binary`: Uses Binary Reduction on a list of closures.

We implemented J-Reduce in 7,929 lines of Haskell code that passed FSE’s artifact evaluation [KP19a] and is open source¹.

¹<https://github.com/ucla-pls/jreduce>

Choice of Predicate For testing of the decompilers, we use the property that a decompiler should produce source code that compiles with `javac`; otherwise it has a bug. We use the predicate that `javac` produces the same bug as the original bug.

To get a monotonic predicate we took special care to keep all inputs except the reduced class files exactly the same. For example, the internal ordering in the file system may play a role in the output of the decompilers and in `javac`. Specifically, `javac` produces only a subset of the bugs in the source code, depending on which files it reads first. So, we kept a sorted lists of files and only wrote to the file system and jars in that order.

Choice of Decompilers We choose three decompilers as the basis of our predicates: CFR [Ben, version 0.132], Fernflower [Sc, commit 8be977e76], and the decompiler from the Procyon project [Str, version 0.5.30]. We set up each decompiler according to the instructions on its webpage. We ran Fernflower with the `-dgs=1` flag to enable handling of generics. We ran CFR with `--caseinsensitivefs true`. We ran Procyon with no special arguments.

Benchmarks Our benchmarks are 100 large Java programs that we obtained from the NJR project [PL18]. We selected programs that each has at least 100 classes and for which we have source code. We focus on bytecode files that we have produced from source code ourselves to ensure that we start each reduction with a valid bytecode program. Some of the projects are dependent on large-scale libraries, but our reduction leaves those libraries unchanged and we exclude them from the dependency graph and our measurements.

We found that CFR fails on 94% of the programs, Fernflower fails on 56%, and Procyon fails on 88%. Thus, in total we have 238 failure-inducing inputs.

Figure 3.6 shows how the distribution of the inputs over three metrics: number of classes, average in-degree and out-degree in the underlying dependency graph (excluding self-loops), and number of strongly connected components. The inputs contain a median of 171 classes, and between 103 and 1006 classes. The benchmarks are diverse both in terms of the in-degree and out-degree, with a median of 3.6, and in terms of the number of strongly connected

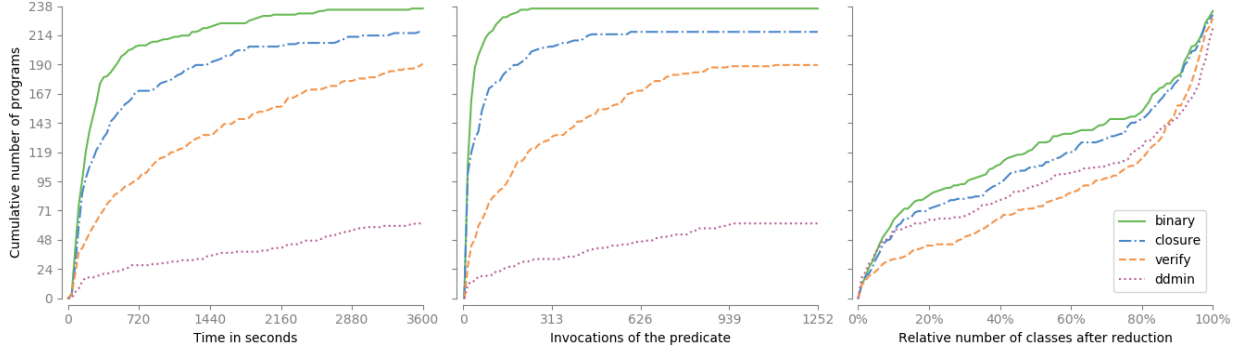


Figure 3.7: Cumulative frequency diagrams of different metrics. The first two charts show the number of cases that terminate within x seconds and x iterations, and the third chart shows the final number of classes relative to the original size. Higher is better.

	timeout	final size	time [s]
binary	0.8%	25.7%	203
closure	8.8%	29.8%	336
verify	19.8%	42.6%	750
ddmin	74.8%	31.9%	2339

Table 3.1: Aggregated results of all the runs. The first column indicates the percentage of runs that we had to timeout. The second column is the average (GM) final relative size after reduction. The third columns are the average (GM) running times in seconds. Smaller is better.

components, with a median of 100.

Platform We performed the experiments on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 Gb RAM. We executed the experiments using OpenJDK (1.8.0_172-02). We ran the experiments in parallel in batches of 8. We ran each reduction for no more than an hour (3600s).

3.5.2 Results

For each reduction algorithm (binary, closure, verify, and dadmin) we measured the total time in seconds, the number of invocations of the predicate made by the algorithm, and the

fraction of classes left in the output after reduction. The results are shown in Figure 3.7. If a tool was timed out after an hour (3,600 seconds), we report the smallest set of classes that had been found to preserve the bug. This reflects that a user can use the best result available at time out.

The times include the generation of the graph (median 0.5s, max 7.1s), the initial run that tests if the predicate is true, and for the tools that used closures, the calculation of the closures (median 5.0ms, max 96.0ms).

Table 3.1 shows the aggregated results of all the runs: the percentage of the runs that we time out, the geometric mean of the relative final size, and the mean time used (including timeouts). The geometric mean allows us to talk about how many times a tool is better than another, based on how much is left after reduction.

We have also plotted all the results in cumulative charts. Figure 3.7 shows the results of the four configurations in three charts. The first chart shows how many programs that each reducer has finished after some seconds. The second chart shows how many programs that each reducer has either finished or timed out on after some invocations of the predicate. The third and final chart represents the relative size after reduction for an hour.

First, let us evaluate how `ddmin` (`ddmin`) performs against `ddmin` plus detection of invalid bytecode (`verify`). Unsurprisingly, `verify` is much faster than `ddmin`, because it does not have to run the predicate for the cases where not all the dependencies are present. Also, `verify` timeouts on 19.8% of the programs where `ddmin` timeouts on 74.8%. The resulting size of `verify`, however, is worse, with 42.6% average final size against `ddmin`'s average of 31.9%. There are two factors that affect this. One, adding the verifier makes the predicate more non-monotonic: missing dependencies in classes not visited by `javac` emerge as a problem. Two, our dependency graph may be an over-approximation of the actual dependencies used by the decompilers and compiler. This means that our verifier can reject a program that might decompile and make `javac` produce an error.

Second, let us evaluate runs of `ddmin` on a list of closures (named `closure`). This is

affected by the overapproximation of the dependency graph, but the number of items is now both smaller (median 100 vs median 171), and also the predicate is now monotonic (disregarding non-determinism). This has a dramatic effect on speed and leads to a smaller output. `closure` only timeouts in 8.8% of all the inputs, and produces on average 29.8% final size. `closure` performs better on most of the inputs, but `ddmin` outperforms `closure` in a few cases. We think this happens because the dependency graph is an overapproximation of the actual dependencies used by the compiler. This means that `ddmin` in some cases can remove an extra class, because it is in reality not needed by the compiler.

Third, let us evaluate runs of Binary Reduction on a list of closures (`binary`). `binary` performs better than `closure`, with a timeout rate of only 0.8%, and is on average 1.7x faster. The final size of the reduction is also better, with 25.7% final size on average. The better reduction can be attributed to two factors: fewer timeouts and Binary Reduction's ability to pick the smallest closure. When controlling for the fewer timeouts, by not counting the benchmarks where either of the algorithms timed out, Binary Reduction is able to produce 1.11x smaller results than delta debugging. On a few cases `ddmin` outperforms `binary`, this is partly due to the overapproximation, but also that some of the benchmarks could yield different results when run twice. `ddmin` sometimes runs the same reduction candidate twice, which means that it has a higher chance of getting lucky and accepting the reduction.

In conclusion, using Binary Reduction on the dependency graph of Java bytecode programs are 12x faster and 1.24x smaller results on average than delta debugging directly on the list of classes.

3.5.3 Threats to Validity

External Validity. The primary threats to external validity is the choice of domain. We chose the domain of decompilers, because bugs were plentiful and easy to find. We do, however, believe that the results extend to all domains with inputs with internal dependencies, and especially to domains that expect valid bytecode programs as inputs.

Internal Validity. We chose 100 fairly large benchmarks at random from the NJR repository; we deem them to be representative of real life programs. We chose programs with over 100 classes to go beyond what people may be willing to reduce by hand. The timeout time was set at one hour, which might skew the study; however, we think that an hour is a reasonable cutoff for most simple applications. Our definition of a bug in a decompiler (*produces code that does not compile*) is not the strongest definition. We could expect that we could find even more bugs if we had used a stronger requirement. This does however not affect our study as we find plenty bugs.

In our experiment, we reduced using a cost function that tries to minimize the number of classes; however, the classes' total size would also be an interesting metric, as two small classes might be better than one big class. Our technique is sufficiently general to reduce using any cost function, though we do expect `ddmin` to perform even worse in this case, and it would be an unfair comparison as `ddmin` can only reduce based on counts. While we believe that `ddmin` is an adequate baseline, we could calculate the reduction approximation ratio of both algorithms if we had an ideal reduction, which we could find by doing an exhaustive search. We leave this to future work, perhaps for a smaller benchmark suite. Finally, the decompilers that we have chosen are not completely deterministic, which means that the predicates, even over the closures, are not completely monotonic. We see this as a strength of the study, since real-life programs are often not deterministic, and predicates are often not completely monotonic.

3.5.4 Data Availability

We have made the raw data used for the analysis available [KP19c]. It includes two files: a “benchmarks.csv” file, which contains the data for histogram in Figure 3.6, and a “deliverable.csv” file, which contains the data for the cumulative diagrams in Figure 3.7 and averages in Table 3.1.

3.6 Reporting bugs

In Section 3.5, we listed results from reducing input in a general manner that preserves the output from `javac` in its entirety. The median reduced bytecode program has 84 classes, which is too many to include in a succinct bug report. This observation led us to consider how domain-specific knowledge about `javac` can lead to additional reduction. We found that the output from `javac` may list multiple problems so a straightforward idea is to preserve *less* than the entire output. As a radical step towards more aggressive reduction, we ran an experiment in which we preserve only `javac`'s exit code. Thus, we preserve that `javac` returns an error, but not which one(s). Indeed, the final list of problems may have no overlap with the initial list.

Our experiment with running Binary Reduction on 238 programs took 34 minutes in wall clock time, or 13 hours in processing time, for an average of 3 minutes per program. The median reduced bytecode program had 2 classes, excluding libraries. Indeed, in 133 cases out of 238 cases, the reduced program had 1 or 2 classes: 57 cases for CFR, 46 for Procyon, and 30 for Fernflower. The output from `javacc` included many distinct error messages (disregarding line number and class): 67 distinct error messages for CFR, 83 for Procyon, and 27 for Fernflower.

We used the results of the experiment to report 2 bugs to CFR, 1 bug to Procyon, and 2 to Fernflower. We choose the benchmarks of size no more than 2 classes, which induced errors that looked like a fixable bugs and were significantly different from each other. The developers of CFR have confirmed and fixed one of the bugs, the developers of Procyon have triaged the bug, but not yet fixed it, and the developers Fernflower have triaged the bugs but not had time to fix them.

We stopped short of filing additional bug reports because we are aware that two failure-inducing inputs may be about the same bug. We want to avoid reporting the same bug twice and we leave it to future work to find an effective way to categorize bug reports.

3.7 Related Work

The literature on program reduction and delta debugging is rich and diverse. We will cover some of the most closely related papers from that literature and we will focus on three aspects. The first aspect is how previous work has dealt with the test-case validity problem, the second aspect is how our approach compares to various approaches to input reduction, and the final aspect looks at program reduction as slicing or debloating.

The Test-Case Validity Problem Our technique is the first to *avoid* invalid input. We will discuss how some prominent papers have dealt with invalid input.

Zeller and Hildebrandt [ZH02] introduced delta debugging. They wrote [ZH02, Section VIII] that “*Delta Debugging assumes that failure is monotone*”. However, their paper showed how to apply Delta Debugging to a variety of input for which failure *isn't* monotone, namely C programs, Mozilla user actions, and UNIX commands. For each kind, we can remove a few characters from a failure-inducing input and thereby change it into an invalid input. In some cases, we can remove additional characters and get another failure-inducing input. Delta debugging works well for those kinds of inputs because most *natural* subsets are valid. In contrast, for Java bytecode, most natural subsets are invalid. Our experiments show that for Java bytecode, delta debugging of a list of classes times out often and gives a disappointing factor of reduction.

Delta debugging has also been implemented in the Delta tool [MWG15]. This tool uses a line-based algorithm that suppresses newlines below a particular depth in the syntax tree. This decreases the risk of removing half of a subtree and thereby producing invalid inputs.

Misherghi and Su [MS06a], in their paper on hierarchical delta debugging, avoided invalid subsets by structuring the input as a syntax tree and by removing entire subtrees at a time. Their insight is that the elements of a subtree can be a natural subset of the input, such as a statement in a statement list. They found that they can remove a single statement from a statement list and preserve that the syntax tree is valid. Compared the classical delta

debugging algorithm, the hierarchical approach gave a decrease in the number tests needed for C-program input by a factor of 11.5 on average. However, while each Java bytecode class is a natural subset of a bytecode program, most subsets of the classes are invalid. For Java bytecode, the top level of hierarchical delta debugging is delta debugging of a list of classes, which we have shown works poorly.

Regehr et al. [RCC12a] identified the problem with invalid input and pursued an approach, for C, that detects invalid input. The core of their approach is akin to the algorithm we called `verify`. They went further and built in detailed knowledge of C that enabled their tool to reduce C-program 25 times more than language-independent tools. In contrast, our tool avoids invalid code and uses a general reducer. Our experiments show that for Java bytecode, detection of invalid input is slow and this gives a small factor of reduction.

Sun et al. [SLZ18] showed, with their tool `Perses`, how to avoid invalid inputs in a language-independent manner. They did this by transforming an input grammar into a convenient form that can guide reduction. They showed that this approach is competitive with less general approaches. However, the approach relies on that once the grammar has reached the convenient form, two specific transformations preserve validity. While indeed those transformations do preserve validity for many kinds of input, they often produce invalid subsets in the case where the input is a list of Java bytecode classes. The problem is that a grammar has no model of the many internal dependencies. Thus, while the generality of the approach is attractive, the approach is ineffective for inputs such as Java bytecode programs.

The recent tool `Chisel` [HLP18] uses reinforcement learning to do fast debloating of C programs. The approach is 3.7–7.1x faster than competing approaches. The approach *detects* invalid input, as illustrated by the following quote from the paper: “Chisel simply rejects nonsensical programs without invoking the test script by using a simple dependency analysis, such as programs that do not contain the main function, variable declarations, variable initializations, or return statements.”. We speculate that one can combine their idea of reinforcement learning with our idea of using a dependency graph to avoid invalid input. We

leave this to future work.

In Cleve and Zeller’s work on STRIPE [CZ00], they tried to use different clustering techniques to increase the speed of delta debugging on an execution trace. Specifically, they wrote “[O]ur future work will concentrate on introducing domain knowledge into delta debugging. In the domain of code changes, we have seen significant improvements by grouping changes according to files, functions, or static program slices, and rejecting infeasible configurations[.]” We believe that we have solved this problem by giving the user a simple interface, graph-based dependencies, with which they can encode many different kinds of domain specific dependency information.

Approaches to Input Reduction BiSect [Dev, Cza18] is a tool for use with git that does a binary search to find the commit that introduced a bug. This is akin to the binary search that we use to implement the min function in Binary Reduction. The BiSect technique does no reduction of the input.

The papers by Artho [Art11], by Li et al. [LZR16], and by Yu et al. [YLC12] all have the goal to isolate failure-inducing changes in a revision history. They use clever representations of revision histories and use variations of delta debugging to achieve the goal. In all three papers, dependencies among changes and validity of history slices play major roles. Artho [Art11] notes that, in the context of interdependent changes, the approach “cannot deal with certain changes affecting multiple files”. Li et al. [LZR16] *detects* invalid history slices, while Yu et al. [YLC12] uses classical delta debugging with no optimization for invalid input. We speculate that those approaches can be enhanced with ways to avoid invalid history slices, in a way that is akin how we avoid invalid input. We leave this to future work.

Delta debugging has a wide range of applications. In particular, researchers have shown how to use delta debugging to help normalize, generalize, and improve test cases [GHK17, GAZ16, LOZ07]. For test cases with many internal dependencies, our approach can be used to avoid giving invalid inputs to the reducer.

Program Slicing and De-bloating Delta debugging can be used to slice a program [Wei81], that is, reduce the program while preserving its behavior. When the slices are intended to be used as runnable program, such reduction is called de-bloating.

Delta debugging is a simple approach to slicing as it requires little knowledge about the program: we can reduce the program while preserving the observable properties like those given test cases. Binkley et al. [BGH14] uses delta debugging to reduce a set of files using a technique they call observational slicing (ORBS). Our technique is sufficiently general that it can augment ORBS by allowing the user to define dependency edges between lines in different files.

J-Reduce functions perfectly as a program slicer. Since we are using a static analysis to detect the edges in the dependency graph we likely under-approximate edges that are the result of reflection. Under-approximating the dependency graph is acceptable, as it will only result in more strongly connected components in our algorithm. In the worst case we have exactly one strongly connected component for each input node, which means that we are just doing regular reduction. This might take longer, but will always output correct bytecode.

Our technique is akin to Agrawal and Horgan [AH90], which uses a over-approximating static analysis to collect a dependency graph between statements in a program. It then uses a dynamic analysis to traverse the program and reduce the graph to see which nodes are actually executed. In contrast our technique starts with a possible under-approximating static analysis and does not need to run the program. This means that it can be used on other properties like finding bugs in decompilers. Since our technique tolerates under-approximation, we might use a dynamic analysis to generate the dependency graph. We leave this for future work.

3.8 Summary

We have presented a new approach to reducing failure-inducing input programs with many internal dependencies. Our approach uses a dependency graph to avoid invalid inputs, and it uses a new algorithm called Binary Reduction, that we showed works better than `ddmin`. We have implemented an open-source tool `J-Reduce` that reduces Java class-files. We evaluated our tool on decompilers, yet our tool works for any program that takes class files as input. Examples include static and dynamic analyses, code coverage tools, and code visualizers. Our tool is 12x faster and achieves more reduction than delta debugging. This enabled us to create and submit short bug reports for three Java bytecode decompilers.

We have only evaluated our approach on Java bytecode and decompilers; however, we see no reason it would not extend to other input programs and metaprograms. For example, our technique can be used for languages with module systems, such as `C#` or `Python`. We can also consider using the dependency graph in a graph where the nodes are methods and fields. However, it turns out to be harder than expected. This is precisely the topic of the next chapter.

CHAPTER 4

Logical Input Reduction

Reducing a failure-inducing input to a smaller one is challenging for input with internal dependencies because most sub-inputs are invalid. In the previous chapter, we made progress on this problem by mapping the task to a reduction problem for dependency graphs that entirely avoided invalid inputs.

In this chapter, we allow the removal of elements within the classes, like fields and methods. Removing these elements introduce complicated dependencies, which we show cannot be model precisely using graphs. Our approach uses propositional logic as an expressive framework for specifying dependencies that generalizes dependency graphs. As a demonstration, we show how to specify dependencies among program elements in Java bytecode, using the full power of propositional logic. The work in the previous chapter using graphs does not extend to logic. To that end, we generalize the Binary Reduction algorithm to reduce a progression of increasingly bigger valid sub-inputs and present a simple algorithm for calculating such progressions given a logical dependency model. We use this to improve J-Reduce so that it reduces Java bytecode to 4.6% of its original size, which is 5.3 times better than the previous chapter approach.

4.1 Introduction

We have an input to a program that makes the program produce a bug. The input is well-formed, so the program should handle it; however, the input is so massive that we can't determine the nature of the bug. The process of input reduction, first introduced by

Zeller and Hildebrandt [ZH02], finds a small input to the program that reproduces the bug. Their algorithm, `ddmin`, produces a *sub-input*, the original input with pieces removed, that reproduces the bug in $O(n^2)$ runs of the program. They found that a sub-input can result in three outcomes when given to a program: the bug is still there, the bug is gone, and don't know. The “don't know” outcome happens when it has run an invalid input to the program.

An invalid sub-input is of no help with finding the bug. Since the original input was valid, the reduction must have violated the internal dependencies in the input. For example, in Java, if a method constructs an object from a class, then without the class, the method would no longer type-check. The method *depends* on the class.

To deal with this problem, we can either detect or avoid invalid inputs. If we can detect an invalid input, we don't have to run the program with that input, which is good because executing the program can take a long time. Avoiding invalid inputs is, however, much more efficient, as we saw in the previous chapter. The reason is that avoiding invalid inputs makes the existence of the bug *monotone* over sub-inputs. That is, monotonicity allows for fast and precise reduction because if the bug is not present in the input, we do not have to test any of the sub-inputs contained in the input.

In this chapter, we build on a long tradition of gradually modeling more and more of the internal dependencies of the input to avoid invalid inputs. In Mishherghi and Su's paper on hierarchical delta debugging (HDD) [MS06b], they concluded that they were able to avoid many invalid inputs by exploiting the tree-like nature of the inputs. Sun et al. [SLZ18] took this a step further and used a syntax tree as their model in their tool `Perses`, which enabled them to avoid all inputs that would not syntax-check. These techniques only work for modeling syntax. To address this, we designed a more expressive model that uses a dependency graph, in the previous chapter and accompanying paper [KP19b], which can model strictly more internal dependencies than HDD and `Perses`.

In this chapter, we introduce a new model of dependencies that goes beyond dependency graphs. Consider, for example, the Java program in Figure 4.1a, which is the input to a tool and which makes the tool crash. While a programmer quickly can reduce Figure 4.1a

<pre> class A implements I { String m() { /* bug */ } B n() { ... } } class B implements I { String m() { ... } B n() { ... } } interface I { String m(); B n(); } </pre>	→	<pre> class A implements I { String m() { /* bug */ } } interface I { String m(); } </pre>
---	---	--

(a) The input that we want to reduce.

(b) The optimal reduction.

```

class M {
  String x(I a) { return a.m(); /* bug */ }
  String main() { return new M().x(new A()); /* bug */ }
}

```

Figure 4.1: The example input program that produces an bug in a tool when the body of `M.x()`, `M.main()`, and `A.m()` are present at the same time. The second sub-figure is the optimal reduction that preserves the bug. We exclude the code in `...` for brevity.

to Figure 4.1b, automatic reduction based on a dependency graph will produce a bigger program or be able to produce invalid inputs. The reason is that dependency edges cannot express, for example, that if we want to preserve that `A` implements `I` *and* that `I` has a signature `m`, then we must also preserve that `A` implements `m`. We will see this is more details in the example in the next section.

In this chapter, we solve the underlying problem of modeling dependencies by using the full power of propositional Boolean logic. We use the model to search through valid sub-inputs efficiently while avoiding invalid sub-inputs, and to produce the result in Figure 4.1b. The claim of this chapter is:

The use of propositional Boolean logic for modeling internal dependencies leads to an effective and efficient reduction of complex inputs.

We build this claim on that we find dependencies in Java Bytecode, which can only be represented using propositional Boolean logic. Furthermore, we have shown that we can

effectively and efficiently reduce Java Bytecode programs with dependencies described using this model with our new reduction algorithm, Generalized Binary Reduction.

After a dive into the example in Figure 4.1 that illustrates why previous techniques are unable to model all dependencies (Section 4.2), we have structured the rest of this chapter after our contributions.

- We have built a model of internal dependencies for a modest extension of Featherweight Java, which we call Featherweight Java with Interfaces (FJI). We prove that if a program type checks, then every sub-input that satisfies the dependencies also type checks. This is a sound dependency model of a complex input that previous techniques are unable to model. We then discuss the changes needed to extend this model to Java bytecode (Section 4.3).
- We introduce the Generalized Binary Reduction algorithm, which given a model of the internal dependencies of a bug-inducing input to a program, can find a sub-input in polynomial time while preserving the bug. Furthermore, we have proved the correctness, polynomial time complexity, and an optimality property of the algorithm (Section 4.4).
- We have implemented our approach and evaluated it on the benchmarks from the previous chapter. Our improved tool reduces to 4.6% while J-Reduce only reduces to 24.3%. This is 5.3x more reduction than J-Reduce (Section 4.5).

Finally, we go over related work in Section 4.6 and summarize our findings in Section 4.7.

4.2 Example

This section illustrates that the previous graph-based approach does not extend to a more fine-grained reduction of Java bytecode and what we have done to solve it. Consider the example in Figure 4.1a. It contains a Java source program, which, when compiled, functions

Variables:

In A: $[A], [A \triangleleft I], [A.m()], [A.m()!code], [A.n()], [A.n()!code]$
In B: $[B], [B \triangleleft I], [B.m()], [B.m()!code], [B.n()], [B.n()!code]$
In I: $[I], [I.m()], [I.n()]$
In M: $[M], [M.x()], [M.x()!code], [M.main()], [M.main()!code],$

Syntactic Dependencies:

$[A.n()!code] \Rightarrow [A.n()]$	$[A.n()] \Rightarrow [A]$	$[A.m()!code] \Rightarrow [A.m()]$	$[A.m()] \Rightarrow [A]$
$[B.n()!code] \Rightarrow [B.n()]$	$[B.n()] \Rightarrow [B]$	$[B.m()!code] \Rightarrow [B.m()]$	$[B.m()] \Rightarrow [B]$
$[A \triangleleft I] \Rightarrow [A]$	$[B \triangleleft I] \Rightarrow [B]$	$[I.m()] \Rightarrow [I]$	$[I.n()] \Rightarrow [I]$
$[M.x()!code] \Rightarrow [M.x()]$	$[M.x()] \Rightarrow [M]$	$[M.main()!code] \Rightarrow [M.main()]$	$[M.main()] \Rightarrow [M]$

Referential Semantic Dependencies:

$[A \triangleleft I] \Rightarrow [I]$	$[B \triangleleft I] \Rightarrow [I]$	$[A.n()] \Rightarrow [B]$	$[B.n()] \Rightarrow [B]$
$[I.n()] \Rightarrow [B]$	$[M.x()] \Rightarrow [I]$	$[M.x()!code] \Rightarrow [I.m()]$	$[M.x()!code] \Rightarrow [I]$
$[M.main()!code] \Rightarrow [M.x()]$	$[M.main()!code] \Rightarrow [A]$	$[M.main()!code] \Rightarrow [M]$	

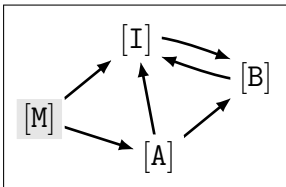
Non-Referential Semantic Dependencies:

$[A \triangleleft I] \wedge [I.m()] \Rightarrow [A.m()]$	$[A \triangleleft I] \wedge [I.n()] \Rightarrow [A.n()]$
$[B \triangleleft I] \wedge [I.m()] \Rightarrow [B.m()]$	$[B \triangleleft I] \wedge [I.n()] \Rightarrow [B.n()]$
$[M.main()!code] \Rightarrow [A \triangleleft I]$	$[M.main()!code]$

Figure 4.2: The variables (20) and dependency constraints (32 + 1 duplicate (gray)) of the example. All the constraints should be conjoined.

as an input to a tool. When we run the tool, we get an error. The error is produced by a combination of the code in the body of `A.m()` and `M.x()`, but we don't know that. We do know, from the tool, that it always requires `M.main()` to run at all. We want to reduce the input program while preserving the error.

In the previous chapter, we described an approach to reduce Java bytecode. J-Reduce models the dependencies between classes using a graph, which allows us to produce smaller results an order-of-magnitude faster than `ddmin` [ZH02]. The modeling language is a conjunction of required classes `[A]` and dependencies between classes `[A] ⇒ [B]`. If we create the graph correctly, all closures in the graph are a valid sub-inputs. In this case, the class dependencies are:

$$\begin{aligned}
 & [M] \wedge ([M] \Rightarrow [A]) \wedge ([M] \Rightarrow [I]) \\
 & \wedge ([A] \Rightarrow [I]) \wedge ([A] \Rightarrow [B]) \quad \text{or} \\
 & \wedge ([B] \Rightarrow [I]) \wedge ([I] \Rightarrow [B])
 \end{aligned}$$


Since we want to preserve the code of `M.main()` we require `[M]`. We include a dependency between every pair of classes if the first mentions the other. However, the resulting dependency graph is disappointing: the graph only contains a single closure, which contains `[M]`. That closure contains all classes, so we cannot reduce the input any further using this technique.

Going Beyond Classes But all is not lost. We can inspect the program and see that if we are allowed to remove items within the classes, we can reduce the program. If we reduce the program by hand, we could get the program, which we show in Figure 4.1b. We can remove four different kinds of items: classes (`[A]`), implementations (`[A < I]`), methods (`[A.m()]`), and the code associated with the methods (`[A.m()!code]`). In this example, we have a total of 20 separate items, which we have listed in Figure 4.2, under the heading Variables.

When we generate constraints beyond the class level, we can reuse some of the ideas from previous work, but not all. In the previous chapter [KP19b], we exclusively modeled

referential dependencies: one item depends on another if the item refers to it. We can transfer this idea directly to items within classes, and we have added a list of *Referential Semantic Dependencies* to fig. 4.2. In summary, both of the `implements` statements mentions the interface `I`, the code of `main` mentions `I`, `A`, and the methods `I.m()`, and all the `n` methods mentions `B`. The `m` methods mention `String`, but since we do not try to remove this class, there is no reason to model dependencies to it.

Contrary to the previous chapter, items are now nested. The nested structure of the items means that we cannot remove a parent item before we have removed all its children. Otherwise, we might find us in a situation where we want to keep a method, but we have removed its enclosing class. We can fix this by adding dependencies from children to their parents. We call these *Syntactic Dependencies*, and we have listed them in Figure 4.2.

Additional Dependencies The syntactic and referential dependencies by themselves are not enough to correctly model valid inputs. We can see this by inspecting Figure 4.3, which is the dependency graph created from the syntactic and referential dependencies. This graph contains closures that are not valid inputs. For example, the closure of variables in `M` (shaded gray) is not a valid input! In `[M.main()!code]` we cast `A` to `I` before we call `I.m()` on `A`. We are simply not allowed to cast `A` to `I`, unless that `A` is a subtype of `I`. In our case, we can see that we needed to preserve `[A < I]`. So we know that there exist dependencies that we have not encoded. Referential dependencies alone are not enough to define all dependencies. Also, there exist references that does not correspond to dependencies. For example, in Java, we can refer to methods that are defined in a superclass. Assume we have a class `C` which extends `A`, then we are allowed to call `(new C()).m()`, because `C` inherits `A`'s methods. The bytecode would refer to `C.m()`, even though there is no method `m()` in `C`. We need a more general concept for defining dependencies.

Our First Contribution In the example, the input has to type-check before we can run the tool on it. The problem is that referential semantic dependencies are not the only kind

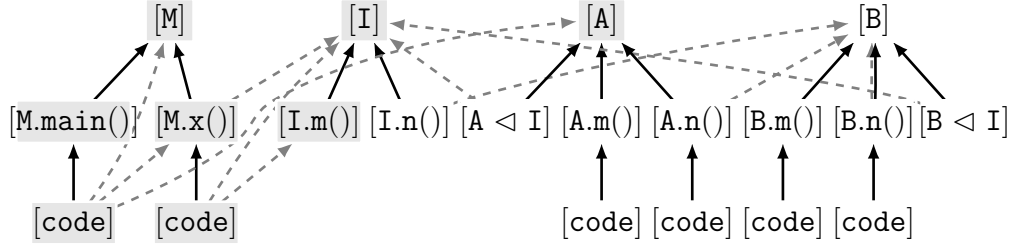


Figure 4.3: The dependency graph containing syntactic (solid, black) and referential (dashed, gray) dependencies. We have abbreviated the code variables to `[code]`. We have shaded the variables part of the minimal closure from all the variables in `M`.

of semantic dependencies. By inspecting the type-checking rules, we can see that the code of `M.main()` casts `A` to `I` and therefore depends on that `A` implements `I`. We can model this like this:

$$[\text{M.main()}!\text{code}] \Rightarrow [\text{A} \triangleleft \text{I}].$$

We also have to model the inheritance laws. If `[I.m()]` should be preserved then `A` has to implement `[A.m()]`. This is true because `A` implements `I` and `I.m()` is an abstract method. However, this constraint depends on `[A < I]`, because if `[A < I]` has been removed we can safely remove `[A.m()]` without removing `[I.m()]`. In other words, if we preserve that `A` implements `I` and `I.m()` we must also preserve `A.m()`:

$$[\text{I.m()}] \wedge [\text{A} \triangleleft \text{I}] \Rightarrow [\text{A.m()}].$$

Finally, we also include `[M.main()!code]`, because, as described earlier, we know the tool does not work without it. We add the last six dependencies in Figure 4.2. The dependencies, now, precisely model the semantics of both the class hierarchy and the type system. A key part of our first contribution is to model the internal dependencies of the type-system of Java using propositional Boolean logic. We will describe a full dependency model of Featherweight Java with Interfaces, which we prove only can produce sub-inputs that type-check. We present this and how we got these exact constraints in Section 4.3.

Our Second Contribution We have shown that we must go beyond dependency graphs to get better reduction than in previous work. Instead we use propositional Boolean logic:

$$F ::= [x] \mid \neg F \mid F \wedge F$$

In our formulation, $[x]$ is a variable that indicates if a construct x remains in the sub-input or is removed. In a sound model, a valid truth assignment corresponds to a valid sub-input of the original input. The graph-based constraints convert directly to the new modeling language, as we can model each edge $[x] \Rightarrow [y]$ as an implication: $\neg([x] \wedge \neg [y])$.

We can iterate through all satisfying truth assignments to the constraints in Figure 4.2 and see that not only are they all valid sub-inputs that type-check but they also contain the truth assignment that constitutes the minimal sub-input in Figure 4.1b:

$$[A \triangleleft I], [A.m()], [A.m()!code], [A], [I.m()], [I], \\ [M.x()!code], [M.x()], [M.main()!code], [M.main()], [M]$$

The original input, with no knowledge of the internal dependencies, has $2^{20} = 1,048,576$ sub-inputs. Far from all of these inputs are valid. Using our new constraints, we can count the number of valid truth assignments with a tool like sharpSAT [Thu06]. Since a satisfying truth assignment corresponds to a valid input, we can see that there are 6,766 valid programs left. While it is possible to run 6,766 different sub-inputs to find the smallest one, this number scales exponentially with the input's size. The previous chapter, we used that we could do reduction on a list of closures of the graph. Since each closure is a valid sub-input and the union of closures is a closure, we could reduce a list of closures quickly by doing binary searches over the union of the prefixes. Sadly in this case, the union of two satisfying truth assignments is not always a satisfying truth assignment. Therefore, we introduce the Generalized Binary Reduction algorithm, which is a new technique to reduce inputs given internal dependencies constraints. With this new algorithm, we can find the

optimal solution by checking only 11 inputs. We present the algorithm and a run on the example in Section 4.4.

4.3 Modeling Dependencies

We will now formalize how we model dependencies and we will discuss aspect of our implementation that go beyond the formal model. We first formalize a core language called Featherweight Java with Interfaces (FJI) and prove as a theorem that reduced programs type check. Then we explain additional aspects of our Java bytecode reducer that go beyond FJI.

4.3.1 Featherweight Java with Interfaces

Featherweight Java with Interfaces (FJI) is a modest extension of Featherweight Java [IPW99]: each class implements a single interface. While we can model the dependencies of Featherweight Java with “dependency edges”, we need the full power of propositional logic for FJI.

FJI is a convenient setting in which to show that reduced programs type check. We will define the syntax and type system for FJI, along with a reducer. From a program, we generate constraints that model the internal dependencies, then we solve the constraints, and finally we feed the solution to a reducer. The idea is that for any solution, the reduced program type checks (Theorem 2).

For examples in FJI, our formalization generates the same constraints as our implementation. In particular, the core of the example in Section 4.2 is an FJI program, and our formalization generates the constraints listed in Section 4.2, we will show this in Section 4.3.2.

Syntax The only novelty compared to Featherweight Java is that a class implements a single interface. An interface consists of a collection of signatures.

P	$::= \bar{R} e$	<i>programs</i>
R	$::= L \mid Q$	<i>type declarations</i>
T, U	$::= C \mid I$	<i>type names</i>
L	$::= \text{class } C \text{ extends } D \text{ implements } I \{ \bar{T} \bar{f}; K \bar{M} \}$	<i>classes</i>
Q	$::= \text{interface } I \{ \bar{S} \}$	<i>interfaces</i>
K	$::= C(\bar{T} \bar{f}) \{ \text{super}(\bar{f}); \text{this}.\bar{f} = \bar{f}; \}$	<i>constructors</i>
M	$::= T m(\bar{T} \bar{x}) \{ \text{return } e; \}$	<i>methods</i>
S	$::= T m(\bar{T} \bar{x});$	<i>signatures</i>
e	$::= x \mid e.f \mid e.m(\bar{e}) \mid \text{new } C(\bar{e}) \mid (T) e$	<i>expressions</i>

Figure 4.4: The syntax of Featherweight Java with Interfaces (FJI).

Our metanotation for Featherweight Java is similar to the one used in the original paper on Featherweight Java [IPW99]. The grammar in fig. 4.4 uses the following metanotation:

- Nonterminal symbols are words written in *this font*.
- Terminal symbols are written in **this font**.
- A production is of the form $lhs ::= rhs$, where lhs is a nonterminal symbol and rhs is a sequence of nonterminal and terminal symbols, with choices separated by \mid .

We use A, B, C, D to range over class names, we use m, p to range over method names, we use f to range over field names, and we use x to range over formal-parameter names. We write \bar{f} as a shorthand for a possible empty sequence f_1, \dots, f_n (and similarly for $\bar{C}, \bar{x}, \bar{e}$, etc) and write \bar{M} as a shorthand for $M_1 \dots M_n$ (with no commas). We write the empty sequence as \bullet and denote concatenation of sequences using a comma. We abbreviate operations on pairs of sequences in the obvious way, writing $\bar{C} \bar{f}$ for $C_1 f_1, \dots, C_n f_n$, where n is the length of \bar{C} and \bar{f} , and similarly $\bar{C} \bar{f};$ as shorthand for the sequence of declarations $C_1 f_1; \dots; C_n f_n;$ and $\text{this}.\bar{f} = \bar{f};$ as shorthand for $\text{this}.f_1 = f_1; \dots; \text{this}.f_n = f_n;$. Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names. We will use subscripts to distinguish metavariables.

Figures 4.6 and 4.7 show the helper rules and type rules of Featherweight Java with Interfaces. In the type rules, we use Γ to range over type environments, that is, mappings

$$\begin{array}{lcl}
\text{reduce}(\bar{R} \ e, \varphi) & = & \text{reduceR}(\bar{R}, \varphi) \ e \\
\text{reduceR} & & \\
(\text{class } C \text{ extends } D & = & \begin{cases} \text{class } C \text{ extends } D & \text{if } \varphi([C]) = \mathbf{1} \\ \text{implements } \text{reduceI}(C, I, \varphi) & \\ \text{implements } I \{ \bar{T} \ \bar{f}; K \ \bar{M} \} & \\ , \varphi & \end{cases} \\
, \varphi & & \left\{ \begin{array}{l} \bullet \\ \bullet \end{array} \right. \quad \begin{array}{l} o/w \\ o/w \end{array} \\
\text{reduceI}(C, I, \varphi) & = & \begin{cases} I & \text{if } \varphi([C \triangleleft I]) = \mathbf{1} \\ \text{EmptyInterface} & o/w \end{cases} \\
\text{reduceR} & & \\
(\text{interface } I \{ \bar{S} \} & = & \begin{cases} \text{interface } I \{ \text{reduceS}(I, \bar{S}, \varphi) \} & \text{if } \varphi([I]) = \mathbf{1} \\ \bullet & o/w \end{cases} \\
, \varphi & & \\
\text{reduceM} & & \\
(C & = & \begin{cases} T \ m(\bar{T} \ \bar{x}) \{ \text{return } e; \} & \text{if } \varphi([C.m()!code]) = \mathbf{1} \\ , T \ m(\bar{T} \ \bar{x}) \{ \text{return } e; \} & \text{if } \varphi([C.m()]) = \mathbf{1} \\ , \varphi & \wedge \varphi([C.m()!code]) = \mathbf{0} \\ \bullet & o/w \end{cases} \\
, \varphi & & \\
\text{reduceS}(I, T \ m(\bar{T} \ \bar{x}), \varphi) & = & \begin{cases} T \ m(\bar{T} \ \bar{x}) & \text{if } \varphi([I.m()]) = \mathbf{1} \\ \bullet & o/w \end{cases}
\end{array}$$

Figure 4.5: Our *reduce* function of FJI.

from identifiers to types. We use the abbreviation $\bar{\pi} = \pi_1 \wedge \dots \wedge \pi_n$. We use $P(C)$ to denote the class in P with name C , and we use $P(I)$ to denote the interface in P with name I . For every P we assume we have

$$P(\text{EmptyInterface}) = \text{interface EmptyInterface } \{ \}.$$

The type rules specify the conditions under which a program P type checks. When P satisfies those conditions, we write $\vdash P \mid \pi$. We explain the role of π below.

Boolean Variables and a Program Reducer For a given program, we define a set of Boolean variables that will be used by the constraints. Then we define a reducer that given a solution to the constraints will map a program to a reduced program.

From a program P , we derive a set of Boolean variables that we denote $V(P)$. We use φ as a truth assignment of the variables to range over $V(P) \rightarrow \text{Bool}$. The idea is that $\varphi([C]) = \mathbf{1}$, then the reducer should keep class C and otherwise remove it. We have six kinds

of variables: $[C]$ toggles the class C , $[I]$ toggles the interface I , $[C.m()]$ toggles the method $C.m$ in C and $[I.m()]$ toggles the signature $I.m$ in I . The variable $[C \triangleleft I]$ signals if we should keep C implements I or we can change it to C implements `EmptyInterface`. Finally, the variable $[C.m()!code]$ signals if we should keep the body of method $C.m()$. Otherwise, we can replace it with a trivial body.

The reducer in Figure 4.5 implements the idea of the Boolean variables explained above. The reducer forms the core of our implementation for Java bytecode. For any mapping $\varphi : V(P) \rightarrow Bool$, we can construct a reduced program $reduce(P, \varphi)$.

Generating Type-checking Constraints Figure 4.6 and Figure 4.7 show the type rules. For a program P , we use the notation $\vdash P \mid \pi$ to denote that we simultaneously type check P and generate a propositional formula π that uses variables in $V(P)$. We use the notation $\varphi \models \pi$ to denote that φ satisfies π .

The helper rules for FJI in Figure 4.6 are much like in Featherweight Java, there are, however two differences. The first is that we extended method type lookup to apply to interfaces, and that now the subtyping rules generate constraints that model the connection between a class and its interface. The second is a new group of rules for method choice. For a class C and a method m in a program P , the constraint $mAny(P, m, C)$ is a disjunction of variables of the form $[C.m()]$. If we need C to implement a method m in the reduced program, then we can require $mAny(P, m, C)$ to be true. This will ensure that the reducer will preserve at least one such method m .

The type rules for FJI in Figure 4.7 are like the type rules for Featherweight Java except for new rules related to interfaces and signatures, plus the generation of constraints. In the rule for class typing, the constraints says that if we preserve class C , then we also need to preserve class D plus the types of the fields. Additionally, if we preserve that class C implements interface I , then we need to preserve both C and I . In the rule for method typing, the constraints say that if we preserve method m , then we also need to preserve the enclosing class C and the parameter types and the return type. Additionally, if we preserve the method

Field lookup

$$fields(P, \text{Object}) = \bullet$$

$$\frac{P(C) = \text{class } C \text{ extends } D \text{ implements } I \{ \bar{T} \bar{f}; K \bar{M} \} \quad fields(P, D) = \bar{U} \bar{g}}{fields(P, C) = \bar{U} \bar{g}, \bar{T} \bar{f}}$$

Method type lookup

$$\frac{P(C) = \text{class } C \text{ extends } D \text{ implements } I \{ \bar{T} \bar{f}; K \bar{M} \} \quad U \ m(\bar{U} \ \bar{x}) \{ \text{return } e; \} \in \bar{M}}{mtype(P, m, C) = (\bar{U} \rightarrow U)}$$

$$\frac{P(C) = \text{class } C \text{ extends } D \text{ implements } I \{ \bar{T} \bar{f}; K \bar{M} \} \quad U \ m(\bar{U} \ \bar{x}) \{ \text{return } e; \} \notin \bar{M}}{mtype(P, m, C) = mtype(P, m, D)}$$

$$\frac{P(I) = \text{interface } I \{ \bar{S} \} \quad U \ m(\bar{U} \ \bar{x}) \in \bar{S}}{mtype(P, m, I) = (\bar{U} \rightarrow U)}$$

Method choice

$$mAny(P, m, \text{Object}) = 0$$

$$\frac{P(C) = \text{class } C \text{ extends } D \text{ implements } I \{ \bar{T} \bar{f}; K \bar{M} \} \quad U \ m(\bar{U} \ \bar{x}) \{ \text{return } e; \} \in \bar{M}}{mAny(P, m, C) = [C.m()] \vee mAny(P, m, D)}$$

$$\frac{P(C) = \text{class } C \text{ extends } D \text{ implements } I \{ \bar{T} \bar{f}; K \bar{M} \} \quad U \ m(\bar{U} \ \bar{x}) \{ \text{return } e; \} \notin \bar{M}}{mAny(P, m, C) = mAny(P, m, D)}$$

$$\frac{P(I) = \text{interface } I \{ \bar{S} \} \quad U \ m(\bar{U} \ \bar{x}) \in \bar{S}}{mAny(P, m, I) = [I.m()]}$$

Subtyping

$$P \vdash T \leq T \mid \mathbf{1} \quad \frac{P \vdash T \leq T' \mid \pi_1 \quad P \vdash T' \leq T'' \mid \pi_2}{P \vdash T \leq T'' \mid \pi_1 \wedge \pi_2}$$

$$\frac{P(C) = \text{class } C \text{ extends } D \text{ implements } I \{ \bar{T} \bar{f}; K \bar{M} \}}{P \vdash C \leq D \mid \mathbf{1}}$$

$$\frac{P(C) = \text{class } C \text{ extends } D \text{ implements } I \{ \bar{T} \bar{f}; K \bar{M} \}}{P \vdash C \leq I \mid [C \triangleleft I]}$$

Valid method overriding

$$\frac{mtype(P, m, D) = \bar{U} \rightarrow U \text{ implies } \bar{U} = \bar{T} \text{ and } U = T}{override(P, m, D, \bar{T} \rightarrow T)}$$

Figure 4.6: FJI helper rules.

Program typing

$$\frac{P = (\overline{R} e) \quad \overline{R} \text{ OK in } P \mid \overline{\pi} \quad P, \emptyset \vdash e : T \mid \pi}{\vdash P \mid \overline{\pi} \wedge \pi}$$

Class typing

$$\frac{\begin{array}{l} \text{fields}(P, D) = \overline{U} \overline{g} \quad K = C(\overline{U} \overline{g}, \overline{T} \overline{f}) \{ \text{super}(\overline{g}); \text{this}.\overline{f} = \overline{f}; \} \\ P \vdash \overline{M} \text{ OK in } C \mid \overline{\pi} \quad P(I) = \text{interface } I \{ \overline{S} \} \quad P \vdash \overline{S} \text{ OK in } I \text{ for } C \mid \overline{\tau} \end{array}}{\text{class } C \text{ extends } D \text{ implements } I \{ \overline{T} \overline{f}; K \overline{M} \} \text{ OK in } P \mid \frac{([\mathbf{C}] \Rightarrow ([\mathbf{D}] \wedge [\overline{\mathbf{U}}] \wedge [\overline{\mathbf{T}}])) \wedge ([\mathbf{C}] \triangleleft I) \Rightarrow ([\mathbf{C}] \wedge [I]) \wedge \overline{\pi} \wedge \overline{\tau}}{}}$$

Interface typing

$$\frac{P \vdash \overline{S} \text{ OK in } I \mid \overline{\pi}}{\text{interface } I \{ \overline{S} \} \text{ OK in } P \mid \overline{\pi}}$$

Method typing

$$\frac{\begin{array}{l} P(C) = \text{class } C \text{ extends } D \text{ implements } I \{ \overline{U} \overline{f}; K \overline{M} \} \quad \text{override}(P, m, D, \overline{T} \rightarrow T) \\ P, (\overline{x} : \overline{T}, \text{this} : C) \vdash e : U \mid \pi_1 \quad P \vdash U \leq T \mid \pi_2 \end{array}}{P \vdash T \ m(\overline{T} \overline{x}) \{ \text{return } e; \} \text{ OK in } C \mid \frac{([\mathbf{C.m}()]) \Rightarrow ([\mathbf{C}] \wedge [\mathbf{T}] \wedge [\overline{\mathbf{T}}]) \wedge ([\mathbf{C.m}()! \text{code}]) \Rightarrow ([\mathbf{C.m}()]) \wedge \pi_1 \wedge \pi_2)}{}}$$

Signature typing

$$P \vdash T \ m(\overline{T} \overline{x}) \text{ OK in } I \mid [\mathbf{I.m}()] \Rightarrow ([\mathbf{I}] \wedge [\mathbf{T}] \wedge [\overline{\mathbf{T}}])$$

Signature typing relative to a class

$$\frac{\text{mtype}(P, m, C) = \overline{T} \rightarrow T}{P \vdash T \ m(\overline{T} \overline{x}) \text{ OK in } I \text{ for } C \mid ([\mathbf{C}] \triangleleft I) \wedge [\mathbf{I.m}()] \Rightarrow \text{mAny}(P, m, C)}$$

Expression typing

$$P, \Gamma \vdash x : \Gamma(x) \mid \mathbf{1} \quad \frac{P, \Gamma \vdash e : C \mid \pi \quad \text{fields}(P, C) = \overline{T} \overline{f}}{P, \Gamma \vdash e.f_i : T_i \mid \pi}$$

$$\frac{P, \Gamma \vdash e : T \mid \pi_1 \quad \text{mtype}(P, m, T) = \overline{U} \rightarrow U \quad P, \Gamma \vdash \overline{e} : \overline{T} \mid \overline{\pi} \quad P \vdash \overline{T} \leq \overline{U} \mid \pi_2}{P, \Gamma \vdash e.m(\overline{e}) : U \mid [\mathbf{T}] \wedge \pi_1 \wedge \text{mAny}(P, m, T) \wedge \overline{\pi} \wedge \pi_2}$$

$$\frac{\text{fields}(P, C) = \overline{T} \overline{f} \quad P, \Gamma \vdash \overline{e} : \overline{U} \mid \overline{\pi} \quad P \vdash \overline{U} \leq \overline{T} \mid \pi}{P, \Gamma \vdash \text{new } C(\overline{e}) : C \mid [\mathbf{C}] \wedge \overline{\pi} \wedge \pi} \quad \frac{P, \Gamma \vdash e : U \mid \pi}{P, \Gamma \vdash (T) e : T \mid [\mathbf{T}] \wedge \pi}$$

Figure 4.7: FJI type rules.

body, then we need to preserve the enclosing method. In the rule for signature typing, the constraints say that if we preserve a signature, then we must preserve the enclosing interface as well as the parameter types and the return type. In the rule for signature typing relative to a class C , the constraints say that if we preserve that C implements interface I and we preserve that I has a signature m , then C needs to implement a method m in the reduced program. In the rules for expressions, the constraints ensure that the result type is preserved in the reduced program. Additionally, the constraint for method calls ensures that at least one appropriate method is preserved. We also require that the dispatch type exist. This is not strictly required in FJI because the dispatch type is determined by the expression; however, in Java bytecode, this might be a problem. We have added this constraint to be compatible with the constraints generated by our implementation.

Reduction is Type-Safe Our main theorem is that a reduced program type checks. This means that reduction with any solution to the constraints preserves typability.

Theorem 2. *If $\vdash P \mid \sigma$ and $\varphi \models \sigma$, then $\exists \sigma'$ such that $\vdash \text{reduce}(P, \varphi) \mid \sigma'$.*

We present the proof in the full version of the paper this chapter is based on.

4.3.2 Generating the Constraints in the Example

The code in Figure 4.1a is FJI if we assume that every class extends `Object`, that its constructor is implicit, and that `M` implicitly implements `EmptyInterface`. Finally, we assume that there exists a class `String`, which we preserve while reducing the program. We can now fill in the blanks of Section 4.2 and show how we generate the constraints in Figure 4.2. The constraints in Figure 4.2 are structured after dependency kind; however, in our implementation, we extract them in one go.

From the program typing rules (Figure 4.7), we can see that we can process the classes in parallel and then conjoin the results. Let's start with `A`. We first look at the class typing rule in fig. 4.7. We can see that we have to generate the dependencies for the superclass

and the constructor, the constructor has no parameters so our constraint is $[A] \Rightarrow [\text{Object}]$. The second conjunct generates the constraints for the `implements` statement ($[A \triangleleft I] \Rightarrow [A] \wedge [I] \wedge \bar{\pi} \wedge \bar{\tau}$). Now let's focus on the methods requirements $\bar{\pi}$. There are two methods in `A`, `String m() {...}` and `n () {...}`. For `m` we use the method typing rule to see that:

$$([A.m()] \Rightarrow [A] \wedge [\text{String}]) \quad \wedge \quad ([A.m()!\text{code}] \Rightarrow [A.m()] \wedge \pi_1 \wedge \pi_2)$$

For `n` we can see:

$$([A.n()] \Rightarrow ([A] \wedge [B])) \quad \wedge \quad ([A.n()!\text{code}] \Rightarrow [A.n()] \wedge \pi_1 \wedge \pi_2)$$

We assume, for these two methods, that the expression (π_1) , and the return cast (π_2) do not create any constraints. With that out of the way, we create the constraints $\bar{\tau}$ from the interfaces. Here we use the “Signature typing relative to a class” rules, where we generate constraints that require all the signatures of a class to be implemented by one of its superclasses.

$$([A \triangleleft I] \wedge [I.m()]) \quad \Rightarrow \quad mAny(P, m, A) = ([A.m()] \wedge mAny(P, m, \text{Object})) = [A.m()]$$

The same happens for `n`: $([A \triangleleft I] \wedge [I.n()]) \Rightarrow [A.n()]$.

Since we do not reduce `String` and `Object` we replace their variables with `true`, furthermore we expand all the implications so that they become clauses. We now we have generated the 9 constraints for `A`.

$$\begin{array}{lll} [A \triangleleft I] \Rightarrow [A] & [A \triangleleft I] \Rightarrow [I] & [A.n()] \Rightarrow [A] \\ [A.n()] \Rightarrow [B] & [A.n()!\text{code}] \Rightarrow [A.n()] & [A.m()] \Rightarrow [A] \\ [A.m()!\text{code}] \Rightarrow [A.m()] & [A \triangleleft I] \wedge [I.n()] \Rightarrow [A.n()] & [A \triangleleft I] \wedge [I.m()] \Rightarrow [A.m()] \end{array}$$

The constraints for **B** is follows the same structure. $[B.n()] \Rightarrow [B]$ occurs twice, so we remove the duplicate.

$$\begin{array}{lll}
[B \triangleleft I] \Rightarrow [B] & [B \triangleleft I] \Rightarrow [I] & [B.n()] \Rightarrow [B] \\
[B.n()!code] \Rightarrow [B.n()] & [B.m()] \Rightarrow [B] & [B.m()!code] \Rightarrow [B.m()] \\
[B \triangleleft I] \wedge [I.n()] \Rightarrow [B.n()] & [B \triangleleft I] \wedge [I.m()] \Rightarrow [B.m()] &
\end{array}$$

The constraints for **I** are also straight forward, and uses the “Interface typing” and “Signature typing” rules:

$$\begin{array}{lll}
[I.m()] \Rightarrow [I] & [I.n()] \Rightarrow [I] & [I.n()] \Rightarrow [B]
\end{array}$$

Finally generate the constraints for **M**. **M** has **Object** as super class and **EmptyInterface** as interface, and no parameters in the constructor. This means we only have to generate constraints for **M.x()** and **M.main()**. From “Method typing” we get $[M.x()] \Rightarrow ([M] \wedge [String] \wedge [I])$ and $[M.x()!code] \Rightarrow ([M.x()] \wedge \pi_1 \wedge \pi_2)$. $\pi_2 = \mathbf{1}$ from the “Subtyping” rule ($P \vdash String \leq String \mid \mathbf{1}$). To find that $\pi_1 = [I.m()]$ we have to type-check the expression **a.m()**, using the “Expression typing” rule for method calls. Because the method have no parameters it is relatively simple:

$$\frac{\frac{}{P, \Gamma \vdash a : (\Gamma(a) = I) \mid \mathbf{1}} \quad \frac{\dots \quad String \ m() \ \{\dots\} \in \overline{M}}{mtype(P, m, I) = \bullet \rightarrow String}}{P, (\Gamma = (a : I, \mathbf{this} : C)) \vdash \mathbf{a.m()} : String \mid [I] \wedge \mathbf{1} \wedge (mAny(P, m, I) = [I.m()])}$$

The total constraints for **M.x()** is

$$\begin{array}{lll}
[M.x()] \Rightarrow [M] & [M.x()] \Rightarrow [I] & [M.x()!code] \Rightarrow [M.x()] \\
[M.x()!code] \Rightarrow [I.m()] & [M.x()!code] \Rightarrow [I] &
\end{array}$$

Finally, let us generate the constraints from **M.main()**. We use the “Methods typing” rule to see that $[M.main()] \Rightarrow [M]$ and $[M.main()!code] \Rightarrow ([M.main()] \wedge \pi_1 \wedge \pi_2)$

π_2 is **1** is because $(P \vdash \text{String} \leq \text{String} \mid \mathbf{1})$. Again we find π_1 by type-checking the expression:

$$\frac{\frac{\text{fields}(P, M) = \emptyset}{P, \emptyset \vdash \text{new } M() : M \mid [M]} \quad \frac{\dots \quad \text{String } x(I a)\{\dots\} \in \overline{M}}{mtype(P, x, M) = I \rightarrow \text{String}}}{\frac{\frac{\text{fields}(P, A) = \emptyset}{P, \emptyset \vdash \text{new } A() : A \mid [A]} \quad \frac{P(A) = \text{class } A \text{ extends Object implements } I \{\dots\}}{P \vdash A \leq I \mid [A \triangleleft I]}}{P, \emptyset \vdash \text{new } M().x(\text{new } A()) : U \mid [M] \wedge [M] \wedge (mAny(P, x, M) = [M.x()]) \wedge [A] \wedge [A \triangleleft I]}}$$

So the dependencies from M is:

$$\begin{array}{lll} [M.\text{main}()] \Rightarrow [M] & [M.\text{main}()!\text{code}] \Rightarrow [M.\text{main}()] & [M.\text{main}()!\text{code}] \Rightarrow [M] \\ [M.\text{main}()!\text{code}] \Rightarrow [M.x()] & [M.\text{main}()!\text{code}] \Rightarrow [A] & [M.\text{main}()!\text{code}] \Rightarrow [A \triangleleft I] \end{array}$$

Altogether, we have generated 31 of the 32 constraints from Figure 4.2. We add the last constraint ($[M.\text{main}()!\text{code}]$) after constraint generation because we know that the tool will not work without the body of $[M.\text{main}()]$.

4.3.3 Java Bytecode

We have implemented a reducer and a constraint generator for Java bytecode, where the above model is the core. Java bytecode is not as neat as Featherweight Java, so we had to model more dependencies, as follows.

Variables We have a total of 11 kinds of items that can be removed. Some of them give rise to trivial constraints, which we only use to make sure that the bytecode is formatted correctly. We add support for constructors (modeled as methods), fields $[C.f]$, if the fields are final $[C.f!\text{final}]$, and super-classes relations $[C \blacktriangleleft C']$. Removing constructors and fields simply removes them from the bytecode file. If there are no constructors left, then the standard implementation assumes that the default constructor with no parameters exists. We can only remove constructors if there are no final fields in the class, which is also why

we have a variable for whether a field is final or not. Removing a super-class relation simply sets `Object` as the superclass.

Extended Class Hierarchy The Featherweight Java class hierarchy is much simpler than the full Java Class Hierarchy. Besides also handling removal of superclasses we also have to handle abstract classes, interfaces implementing other interfaces, and classes that implement multiple interfaces. The extended class hierarchy plays the most prominent role when we want to preserve methods and fields. We no longer just have to look at whether the method has been implemented by some superclass, but also if the class still transitively extends the superclass. For example, given that `A` extends `B`, and both `A` and `B` implement `m()`. Then if we call `A.m()` we know that either `[A.m()]` or `[B.m()]` have to exist. But now, because we can set the superclass of any class to `Object`, we have to also require that the superclass of `A` is `B` `[A ◀ B]`. This can create long paths, for example if `B` extends `C` which also implements `m()`, then the constraint looks like this:

$$[A.m()] \vee ([A \blacktriangleleft B] \wedge [B.m()]) \vee ([A \blacktriangleleft B] \wedge [B \blacktriangleleft C] \wedge [C.m()])$$

Because we can both implement and extend classes, we can implement the same interface in more than one way. For example, suppose `A` extends the abstract class `B` and implements the interface `I` which requires the method `n()` to be implemented. Because `B` is abstract it does not have to implement `n()`. Suddenly there are two requirement paths to implement `n()`. So to remove `A.n()` it is not enough to make `A` not implement `I`, we have to break all paths to `I`:

$$([A \triangleleft I] \wedge [I.n()]) \vee ([A \blacktriangleleft B] \wedge [B \triangleleft I] \wedge [I.n()]) \Rightarrow [A.n()]$$

These paths are so prevalent that we compute all extends and implements paths in the hierarchy, and use these to generate the constraints. We declare `SubtypePaths(A)` as a set of all classes and interfaces and paths of implements and extends statements that have to re-

main for that class or interface to remain a subtype. In our two examples $\text{SubtypePaths}(C) = \{(A, [A \blacktriangleleft B], [B \blacktriangleleft C]), (B, [B \blacktriangleleft C])\}$ and $\text{SubtypePaths}(I) = \{(A, [A \blacktriangleleft B], [B \triangleleft I]), (B, [B \triangleleft I])\}$.

Casting Java has both up-casting and down-casting of objects: we can always cast an A to B by $(B)(\text{Object}) A$. To simplify the type-system in Featherweight Java is, casting was made type-safe by design. In Java bytecode, this operation is in two steps with the `checkcast` instruction. Each of these operations still carries constraints. For example, if we cast $(A) B$, then A has to be a subtype of B or B has to be a subtype of A .

Handling Type Inference Featherweight Java only contains expressions and no statements, so it contains no real control flow. Java bytecode does not save all the type information. It, therefore, uses some amount of type-inference. One interesting case is that, because of control flow, a variable can be assigned multiple types. Consider the following code, where both A and B extends C .

```
if (x) {l = new A ();} else {l = new B ();} l.toString();
```

Depending on the value of x , l will contain an object of type A or B . The Java type-checker claims that l will be of type C , as it is the first superclass of both A and B . We could model this behavior in by requiring that A is a subtype of C and B is a subtype of C , and then require that C has a `toString` method. This would require that we preserve C . This is too restrictive, as follows. If we set both $[A \blacktriangleleft C]$ and $[B \blacktriangleleft C]$, the program would still type check because they both have `Object` as a superclass and `Object` implements `toString`. To fix this, we keep a list of possible types for each variable when we type-check the input program. Then we generate constraints for all possible types of each variable. In our case, we get that both A and B transitively implement `toString`.

Stubbing Methods In Featherweight Java, we can replace a method body with a small expression. For Java bytecode, we have more options for stubbing a method, as follows. If the return type is `void`, then we can remove the entire body, and otherwise, we can return

a default value of the return type. For example, a stubbed method that returns an `Object` will return `null` and a stubbed method that returns an `int` return `0`.

Java Generics We have strived to make our model sound while still producing as many interesting sub-inputs as possible. When we compile Java to bytecode, the compiler erases some of the generic type information. To correctly model generics, we would have to infer the types of many of the constructs in the code. This problem is undecidable [Gri17]. We can construct an example of the problem of type-erasure by adding generic type information to a local variable:

$$\text{Class}\langle? \text{ extends } B\rangle a = A.\text{class} \longrightarrow \text{ldc } [\text{class } A]$$

When we compile the source line, we get a single load constant instruction (`ldc`) and none of the generic information. Given the original code, we would have created a constraint for $[A \triangleleft B]$, but we erased that information when we compiled it to bytecode. To be sound we could overapproximate and require that the class-hierarchy never changes, but doing so would significantly restrict the number of sub-inputs. Instead, we chose an unsound middle ground. It turns out that the example is a particular case that is very prevalent in some kinds of applications. We do a partial overapproximation, and add dependencies from the code, which inspects a class in this way, to every implements and extends variable in the class-hierarchy of that class. If the code above is in a method $C.m()$, then we would add the constraint $[C.m()] \Rightarrow [A \triangleleft B]$. In practice, we have seen that our model is good enough and we never end up with an invalid sub-input, which illustrates that our algorithm does not need a completely sound model to get good results.

4.4 Logical Reduction

In this section, we will restate the “Input Reduction Problem” using logic and show how the Generalized Binary Reduction algorithm enables us to efficiently reduce the input. But first, we will introduce some notation which we use throughout this section.

4.4.1 Notation

We use small letters to refer to variables v , capital letters to refer to sets L , and calligraphic letters \mathcal{L} to reference to sets of sets. 2^X indicates the power set of a set X .

A *solution* M is a satisfying assignment to a logical statement R ($M \models R$). We write all solutions as the set of *true* variables. For example $\{x\} \models x \wedge \neg y$ but $\{x, y\} \not\models x \wedge \neg y$. We can also get the solutions ($M \in \text{SOLS}(R)$), and the variables $\text{VARS}(R)$ of an logical statement. When it introduces no confusion we use $R(M)$ to mean ($M \models R$). We can also condition, or update, logical expressions ($R \mid x = \mathbf{1}, y = \mathbf{1}$), which effectively substitutes $x = \mathbf{1}$ and $y = \mathbf{1}$ in R . This also works for sets ($R \mid X = \mathbf{1}$).

Monotonicity over sets is a key property of this chapter. We write it as this proposition: If $X \subseteq Y$, then $A(X) \sqsubseteq A(Y)$, where A is the monotone function, and \sqsubseteq is the lattice relation for the different types: $\sqsubseteq \equiv \Rightarrow$ for Booleans ($\mathbf{0} \sqsubseteq \mathbf{1}$), $\sqsubseteq \equiv \subseteq$ for sets, and $\sqsubseteq \equiv \leq$ for integers.

A conjunctive normal form (CNF) is a representation of a logical expression using a conjunction of clauses. A clause is a disjunction of variables and negated variables, and a term is a conjunction of variables and negated variables. A transposed graph is a graph with all edges inverted, and the post order of a graph is a list of variables extracted in a post-order fashion from a depth-first forest of the graph. Furthermore, we work a lot with sets of sets, prefixes, clauses, and terms, so we define the following shorthands:

$$\mathcal{D}^{\cup} = \bigcup_{D \in \mathcal{D}} D \quad \mathcal{D}_{\leq r}^{\cup} = \bigcup_{j \leq r} \mathcal{D}_j \quad L^{\vee} = \bigvee_{l \in L} l \quad L^{\wedge} = \bigwedge_{l \in L} l$$

4.4.2 Formalizing the Problem

We are now ready to formalize the Logical Input Reduction Problem. Using the *reduce* function from the last section, we represent an input as a set of variables I and the sub-inputs as subsets of I . We represent the tool that may have a bug as a predicate P that is true on a sub-input that induces a bug in the tool. Finally, we generate constraints R_I from

the input. The problem is now:

Definition 2 (Logical Input Reduction Problem). *Instance:* (I, P, R_I) , where I is an input, represented as a set of variables, P is a predicate on sub-inputs of I that determines whether a sub-input preserves the error, and R_I is a propositional Boolean formula generated from I that determines whether a sub-input is valid. *Assumptions:* P can be evaluated in polynomial time, both $P(I)$ and $R_I(I)$ are true, and P is monotonic over valid sub-inputs: if $X \subseteq Y$ and $R_I(X)$ and $R_I(Y)$, then $P(X) \Rightarrow P(Y)$. *Problem:* find the smallest sub-input S of I such that $P(S)$ and $R_I(S)$.

Contrary to the Weighted Input Reduction Problem from the previous chapter, this problem definition does not have a cost-function. In the previous chapter, we changed the inputs, so we reduced sets of inputs, where all unions were valid sub inputs, instead of individual items. We needed the cost function to represent that each item now could contain multiple "real" items. In this case, we don't change the input. Instead, we model the input validity using R_I . Since we do not map the input, we do not need a cost function. Of course, there also exists a Weighted Logical Input Reduction Problem that additionally has a cost function, but we'll leave the analysis of that problem for future work.

The problem is still NP-hard, which we can see by reducing from SAT. Given a formula φ , we can invoke the Logical Input Reduction Problem on $(\text{VARS}(\varphi), P, \varphi \vee \text{VARS}(\varphi)^\wedge)$ where $P(X) = \mathbf{1}$ and get a smallest solution S . We know that S is a solution to $\varphi \vee \text{VARS}(\varphi)^\wedge$, so if S is smaller than $\text{VARS}(\varphi)$ or if $\varphi(\text{VARS}(\varphi))$ then we know that φ is satisfiable.

4.4.3 The Generalized Binary Reduction

In Algorithm 2, we present the Generalized Binary Reduction (GBR) algorithm. The GBR extends the Binary Reduction algorithm [KP19b, the previous chapter] with a generic progression subroutine (PROGRESSION) that, given a set of learned sets (\mathcal{L}) and the current search space, produces a progression of valid inputs. We will present the subroutine in Section 4.4.4.

Algorithm 2: Generalized Binary Reduction

Input: (I, P, R_I) where $P(I)$ and $R_I(I)$.

Output: A subset of I which satisfy $P(I)$ and $R_I(I)$.

Data: The learned sets $\mathcal{L} \leftarrow \emptyset$ and progression $\mathcal{D} \leftarrow \text{PROGRESSION}_{R_I}(\mathcal{L}, I)$

while $\neg P(\mathcal{D}_0)$ **do**

$r \leftarrow \min r$ st. $r > 0 \wedge P(\mathcal{D}_{\leq r}^{\cup})$
 $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathcal{D}_r\}$
 $\mathcal{D} \leftarrow \text{PROGRESSION}_{R_I}(\mathcal{L}, \mathcal{D}_{\leq r}^{\cup})$

end

return \mathcal{D}_0

The algorithm works like this. The learned sets (\mathcal{L}) represent the growing knowledge we get about P while running the input. Each input that could satisfy P intersects with all sets in \mathcal{L} . In the beginning, we know nothing about P , so \mathcal{L} is empty. We then build the first \mathcal{D} progression from the input I . Like the original algorithm, we check if the first element is a solution by itself. Otherwise, we search the prefixes of the progression $\mathcal{D}_{\leq r}^{\cup}$, with a binary search, until we find a minimal r . Because each prefix that did not contain \mathcal{D}_r , failed P , we know that at least one element in that set must be part of the solution, so we add the set to \mathcal{L} . Finally, we recompute the progression over the new learned sets, and we limit the search space to variables in the union of the found prefix. We continue like this until the first element of the progression is a solution to $P(\mathcal{D}_0)$, in which case we return it.

Progression Properties and Invariants The GBR only works if the progression algorithm has some properties. In Definition 3, we have identified four properties of the progression, which are crucial for GBR to work correctly.

Definition 3 (Progression Properties). *Given an reduction model R_I , a set of learned sets \mathcal{L} , and an input space J , then for a progression $\mathcal{D} = \text{PROGRESSION}_{R_I}(\mathcal{L}, J)$, assuming $R_I(J)$ and that $\forall L \in \mathcal{L}. J \cap L \neq \emptyset$:*

- *The progression is a non-empty split of the input space J .*

$$|\mathcal{D}| > 0 \quad \wedge \quad \mathcal{D}^{\cup} = J \quad \wedge \quad \forall i, j. i \neq j \Rightarrow \mathcal{D}_i \cap \mathcal{D}_j = \emptyset. \quad (\text{SPLIT})$$

- The progression is correct, if all non-empty prefixes of the progression is a solution to R_I and intersects with all elements in \mathcal{L} .

$$\forall r \geq 0. \quad R_I(\mathcal{D}_{\leq r}^{\cup}) \quad \wedge \quad \forall L \in \mathcal{L}. \mathcal{D}_{\leq r}^{\cup} \cap L \neq \emptyset \quad (\text{CORRECT})$$

- The progression is semi-optimal if the first set D_0 in the progression is minimal:

$$\forall T \subseteq \mathcal{D}_0. \quad R_I(T) \wedge (\forall L \in \mathcal{L}. T \cap L \neq \emptyset) \Rightarrow T = \mathcal{D}_0 \quad (\text{SEMIOPT})$$

- The progression is limited, if we only can add the last element in the progression $\mathcal{D}_{|\mathcal{D}|-1}$ to \mathcal{L} a polynomial number of times in $|I|$:

$$\text{For constant } p \quad \exists k \leq O(|I|^p). \quad |\mathcal{D}^k| = 1 \quad (\text{LIMITED})$$

where $\mathcal{D}^n = \text{PROGRESSION}_{R_I}(\mathcal{L} \cup \{\mathcal{D}_{|\mathcal{D}^i|-1}^i \mid i < n\}, J)$

We naturally expect the progression to split the input space (**SPLIT**), and that it is correct (**CORRECT**). The correctness property requires that we only output prefixes that are solutions to R_I and have a chance to satisfy P , namely that it intersects with all sets in \mathcal{L} . Given these two requirements alone, we get the following invariant:

Lemma 1 (Invariant). *Given a correct progression ((**SPLIT**) and (**CORRECT**)), then for every step of the Generalized Binary Reduction algorithm on (I, P, R_I) the following invariant holds:*

$$\text{Inv}(\mathcal{L}, \mathcal{D}) \equiv P(\mathcal{D}^{\cup}) \quad \wedge \quad |\mathcal{D}| > 0 \quad \wedge \quad \mathcal{D}^{\cup} \subseteq I \quad (\text{INV-1})$$

$$\wedge (\forall r \geq 0. \quad R_I(\mathcal{D}_{\leq r}^{\cup}) \wedge \forall L \in \mathcal{L}. \mathcal{D}_{\leq r}^{\cup} \cap L \neq \emptyset) \quad (\text{INV-2})$$

$$\wedge (\forall T \subseteq \mathcal{D}^{\cup}. \quad P(T) \wedge R_I(T) \Rightarrow \forall L \in \mathcal{L}. T \cap L \neq \emptyset) \quad (\text{INV-3})$$

From (INV-1), we see that the full progression should always satisfy P , have more than

one element, and be contained in the initial input. The second part, (INV-2), simply illustrates that each progression produced are correct, which is really only used to prove (INV-3). The last piece of the invariant is the core piece of information. It says that all subsets of the current progression that satisfy P and are solutions to R_I intersect with each of the learned sets. This is a powerful invariant that trivially allows us to prove that GBR produces a local minimum, while only requiring that the first element in the progression is semi-optimal (SEMIOPT).

Theorem 3 (Local Minima). *The Generalized Binary Reduction algorithm finds a local minimal solution S , for $T \subseteq S$ then $P(T) \wedge R_I(T)$ if and only if $T = S$, Furthermore, it solves the Logical Input Reduction Problem (I, P, R_I) if the progression algorithm is semi-optimal and correct.*

Finally, we have identified a property of the progression (LIMITED), which is sufficient to guarantee polynomial-time reduction.

Theorem 4 (Polynomial Time). *The Generalized Binary Reduction algorithm (I, P, R_I) finds a S s.t. $P(S) \wedge R_I(S)$ in polynomial time, given that the progression algorithm runs in polynomial time, is correct, and limited.*

The full proof of these two theorems is two pages and is included in full version of the accompanying paper.

4.4.4 Our Progression

We have designed a polynomial-time progression algorithm that is correct and limited, and which is semi-optimal for constraints that encode dependency graphs. Using it in the GBR means that we can reduce in polynomial time. In the special case where R_I can be described using a dependency graph, it finds a local minimum.

It is possible to see our progression algorithm as running a best-first spanning forest on a logical expression. While there are variables left, we compute and output the closures

from those variables. Because we are working with logical expressions, there is some guessing involved if we want a polynomial-time algorithm. We distill this guessing to a variable order, which we will describe later. The full proof that our progression uphold all the progression properties is three pages and is in the full version of the accompanying paper.

Definition 4 (Our Progression). *Our progression $\mathcal{D} = \text{PROGRESSION}_{R_I}(\mathcal{L}, J)$, is a progression of logical closures LC_{\preceq} , where \preceq is a variable order based on R_I .*

$$\mathcal{D}_i = \begin{cases} \text{LC}_{\preceq}(R) & \text{if } i = 0 \\ \text{LC}_{\preceq}(R \wedge x \mid \mathcal{D}_{\leq k}^{\cup} = \mathbf{1}) & \text{if } i = k + 1 \text{ and } \exists x \in \min_{\preceq} J \setminus \mathcal{D}_{\leq k}^{\cup} \end{cases}$$

Where $R = (R_I \wedge \bigwedge_{L \in \mathcal{L}} L^{\vee} \mid \text{VARS}(R_I) \setminus J = \mathbf{0})$

Our progression works by first building a logical closure of R , and then while there still exists a variable that we have not visited (x), we find a solution to $R \wedge x$ given the variables in the prefix so far is true. It is straightforward to see that our progression splits the input space (SPLIT): we output at least one element, and we output every element exactly once. Furthermore, we can see that every prefix we output is a solution to R , which means that it is a solution to R_I and intersects with all elements in \mathcal{L} , which is precisely the requirement for correctness (CORRECT).

Implicative Positive Form (IPF) We represent a constraint in implicative positive form (IPF). An IPF is a CNF where each clause has at least one positive variable in each clause. An IPF has two characteristic features. One, it has at least one solution: all variables are true. Two, conditioning with a positive variable gives back an IPF. For every program, we transform the generated constraint into IPF using the Tseitin transformation [Tse83], which is possible because we assumed that the constraint is satisfiable. In our reasoning, we will also use the dual notion of dual-IPF, where at least one variable in each clause is negative, and where \emptyset is a solution.

Logical Closure We compute the logical closure by choosing the first variable, according to a variable order \preceq , from the clause with only positive variables (\mathcal{O}_R). These clauses will not automatically be satisfied when we set the remaining variables to false. Because R is in IPF, we can condition R with $x = \mathbf{1}$ ($R \mid x = \mathbf{1}$), and $(R \mid x = \mathbf{1})$ will still be an IPF. We can continue doing this until there are clauses with only positive variables left. At this point, R is also dual-IPF, because there are no clauses with only positive variables, which means that the \emptyset is a solution to R . The union of the variables chosen so far will be a solution to R .

Definition 5 (Logical Closure). *Given a variable order \preceq and an IPF R*

$$\text{LC}_{\preceq}(R) = \begin{cases} \{x\} \cup \text{LC}_{\preceq}(R \mid x = \mathbf{1}) & x \in \min_{\preceq} \mathcal{O}_R^{\cup} \\ \emptyset & o/w \end{cases}$$

Where \mathcal{O}_R is the set of clauses in R that has no false variables.

The logical closure is stable, and will always return the smallest element from each set in \mathcal{O}_R , which means that in the full progression algorithm, we will never add a set to \mathcal{L} with the same minimal element. Since there are only $|I|$ distinct elements, we can only do this $|I|$ times, which in essence shows that our progression is limited (**LIMITED**). The logical closure does, however, not guarantee we get the smallest solution. The minimality of the solution depends on the choice of variable order \preceq .

The Variable Order This part of the algorithm is a heuristic. We choose a variable order that works well if R_I represents a dependency graph. To compute the variable order, we compute the transposed graph from R_I . Each clause give rise to edges from all positive variables to all negative variables: $X^{\wedge} \Rightarrow Y^{\vee}$ gets turned into the edges $y \rightarrow x$ for $(x, y) \in X \times Y$ in the graph. Notice we reverse edges of the clauses in R . The variable order is the reverse post-order of this graph.

If R_I is effectively a graph, then our progression is a best-first forest algorithm. Since our

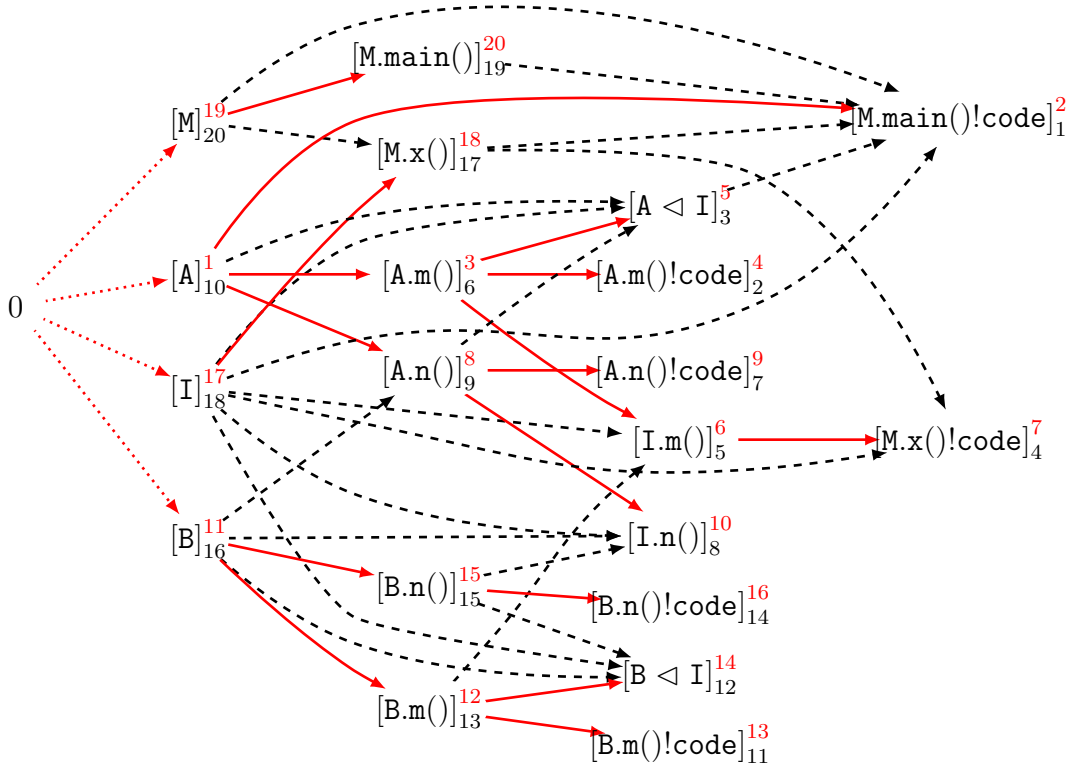


Figure 4.8: The transposed graph used to generate the variable order (red and dashed arrows). The red arrows and numbers symbolise the order of the depth-first forest. The 0 and the dotted red lines represent the root of the forest. The black numbers is the post-order of the forest. The reverse of those numbers is the variable order.

variable order is the reversed post order of the transposed graph, our progression corresponds directly with the Kosaraju–Sharir algorithm, which calculates the strongly connected components (SCC) [Sha81] of a directed graph. Because each element in our progression is an SCC in the graph, and we will only add SCC to \mathcal{L} . The first element in the progression will always be the closures of those SCC’s, which is the minimal solution. Thus, our progression is semi-optimal for graphs (SEMIOPT).

4.4.5 Running the Example

We can now run GBR on the example from Section 4.2, using the constraints from Figure 4.2 (on page 69). This example is produced by running our tool.

R	\mathcal{O}_R	$x \in \min_{\preceq} \mathcal{O}_R^{\cup}$
R	$\{\mathbf{[M.main()!code]}\}$	$\mathbf{[M.main()!code]}$
$(R \mid \mathbf{[M.main()!code]} = \mathbf{1})$	$\{\{\mathbf{[M.main()]}\}, \{\mathbf{[M.x()]}\}, \{\mathbf{[A]}\}, \{\mathbf{[A \triangleleft I]}\}\}$	$\mathbf{[M.main()]}$
$(R \mid \mathbf{[M.main()]} = \mathbf{1})$	$\{\{\mathbf{[M]}\}, \{\mathbf{[M.x()]}\}, \{\mathbf{[A]}\}, \{\mathbf{[A \triangleleft I]}\}\}$	$\mathbf{[M]}$
$(R \mid \mathbf{[M]} = \mathbf{1})$	$\{\{\mathbf{[M.x()]}\}, \{\mathbf{[A]}\}, \{\mathbf{[A \triangleleft I]}\}\}$	$\mathbf{[M.x()]}$
$(R \mid \mathbf{[M.x()]} = \mathbf{1})$	$\{\{\mathbf{[I]}\}, \{\mathbf{[A]}\}, \{\mathbf{[A \triangleleft I]}\}\}$	$\mathbf{[I]}$
$(R \mid \mathbf{[I]} = \mathbf{1})$	$\{\{\mathbf{[A]}\}, \{\mathbf{[A \triangleleft I]}\}\}$	$\mathbf{[A]}$
$(R \mid \mathbf{[A]} = \mathbf{1})$	$\{\{\mathbf{[A \triangleleft I]}\}\}$	$\mathbf{[A \triangleleft I]}$
$(R \mid \mathbf{[A \triangleleft I]} = \mathbf{1})$	\emptyset	\bullet

$$\text{LC}_{\preceq}(R) = \{\mathbf{[A]}, \mathbf{[A \triangleleft I]}, \mathbf{[I]}, \mathbf{[M]}, \mathbf{[M.x()]}, \mathbf{[M.main()]}, \mathbf{[M.main()!code]}\}$$

Figure 4.9: The initial run of $\text{LC}_{\preceq}(R) = \mathcal{D}_0$. Each row corresponds to a recursive call to LC_{\preceq} . The column R represents the value of R at that call, \mathcal{O}_R is the set of positive closures in R , and x is the smallest variable in the union of the sets.

Before we start the algorithm we compute the variable order. We build the transposed graph in Figure 4.8, by drawing edges from the positive variables to the negated variables in each clause in Figure 4.2, effectively reversing each implication. The red arrows and numbers (top) are the result of a depth first search, and the black numbers are the post-order of that search. We extract the variable order, by reversing the post-order of the variables:

$$\begin{aligned} & \mathbf{[M]}_{20} \preceq \mathbf{[M.main()]}_{19} \preceq \mathbf{[I]}_{18} \preceq \mathbf{[M.x()]}_{17} \preceq \mathbf{[B]}_{16} \preceq \mathbf{[B.n()]}_{15} \preceq \mathbf{[B.n()!code]}_{14} \\ & \preceq \mathbf{[B.m()]}_{13} \preceq \mathbf{[B \triangleleft I]}_{12} \preceq \mathbf{[B.m()!code]}_{11} \preceq \mathbf{[A]}_{10} \preceq \mathbf{[A.n()]}_9 \preceq \mathbf{[I.n()]}_8 \preceq \mathbf{[A.n()!code]}_7 \\ & \preceq \mathbf{[A.m()]}_6 \preceq \mathbf{[I.m()]}_5 \preceq \mathbf{[M.x()!code]}_4 \preceq \mathbf{[A \triangleleft I]}_3 \preceq \mathbf{[A.m()!code]}_2 \preceq \mathbf{[M.main()!code]}_1 \end{aligned}$$

Now we can compute the initial progression. We can see that $\mathcal{L} = \emptyset$ and J is the full set of all variables I , so $R = R_I$. We calculate \mathcal{D}_0 by running the logical closure on the unmodified constraints. We illustrate this step by step in Figure 4.9.

The rest of the progression is calculated using $\text{LC}_{\preceq}(R \wedge x \mid \mathcal{D}_{\mathbf{k}}^{\cup} = \mathbf{1})$ with $x \in \min_{\preceq} J \setminus \mathcal{D}_{\preceq k}^{\cup}$. For our first choice after \mathcal{D}_0 , we choose $x = \mathbf{[B]}$, because it is the smallest variable in $J \setminus \mathcal{D}_0$.

$[B]$ implies no new variables, so the set $\mathcal{D}_1 = \text{LC}_{\leq}(R \wedge [B] \mid \mathcal{D}_0 = \mathbf{1}) = \{[B]\}$. We calculate the rest of the progression in the same way. We have annotated each set with the number it is in the progression:

$$\begin{aligned} \mathcal{D} = & \{[A], [A \triangleleft I], [I], [M], [M.x()], [M.main()], [M.main()!code]\}_0, \\ & \{[B]\}_1, \{[B.n()]\}_2, \{[B.n()!code]\}_3, \{[B.m()]\}_4, \{[B \triangleleft I]\}_5, \{[B.m()!code]\}_6, \{[A.n()]\}_7, \\ & \{[I.n()]\}_8, \{[A.n()!code]\}_9, \{[A.m()]\}_{10}, \{[I.m()]\}_{11}, \{[M.x()!code]\}_{12}, \{[A.m()!code]\}_{13} \end{aligned}$$

The progression is ideal, because the initial element is minimal, and every element after the first have size one. Before entering the body of the loop, we run P for the first time on \mathcal{D}_0 : no bug! In Figure 4.10a, we run a binary search over the prefixes of progression to find the smallest one. First, we try the prefix $\mathcal{D}_{\leq 7}^{\cup}$ indicated by the eight black squares in the first row. It fails the predicate, so the correct solution must be between 7–13. In binary search fashion, we cut the search space in half and try $\mathcal{D}_{\leq 10}^{\cup}$. This also fails. We continue this for two more tries until we conclude that the shortest satisfying prefix is the full progression. While we do not reduce the size of the search space, we have learned something important: $[A.m()!code]$ has to be in the solution. We therefore learn $\{[A.m()!code]\}$ by adding it to \mathcal{L} .

We now compute the new progression $\dot{\mathcal{D}}$. We use dots to differentiate the different progressions: $\dot{\mathcal{D}}$, $\ddot{\mathcal{D}}$, and so on. We can see that $R = R_I \wedge [A.m()!code]$. We start by computing $\dot{\mathcal{D}} = \text{LC}_{\leq}(R)$. The set is the same as \mathcal{D}_0 , but we also add $[A.m()!code]$ because it is in \mathcal{O}_R and $[A.m()]$ because we have $[A.m()!code] \Rightarrow [A.m()]$ from the constraints. The rest of our progression is straight forward:

$$\begin{aligned} \dot{\mathcal{D}} = & (\mathcal{D}_0 \cup \{[A.m()], [A.m()!code]\})_0 \\ & , \{[B]\}_1, \{[B.n()]\}_2, \{[B.n()!code]\}_3, \{[B.m()]\}_4, \{[B \triangleleft I]\}_5, \{[B.m()!code]\}_6 \\ & , \{[A.n()]\}_7, \{[I.n()]\}_8, \{[A.n()!code]\}_9, \{[I.m()]\}_{10}, \{[M.x()!code]\}_{11}. \end{aligned}$$

We now start our second iteration of the algorithm. We try $P(\dot{\mathcal{D}}_0)$, which is false. This is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	$P(\mathcal{D}_{\leq r}^{\cup})$
■	■	■	■	■	■	■	■	·	·	·	·	·	·	0
■	■	■	■	■	■	■	■	■	■	■	·	·	·	0
■	■	■	■	■	■	■	■	■	■	■	■	·	·	0
□	□	□	□	□	□	□	□	□	□	□	□	□	□	1
□	□	□	□	□	□	□	□	□	□	□	□	□	⊗	1

0	1	2	3	4	5	6	7	8	9	10	11	$P(\mathcal{D}_{\leq r}^{\cup})$
■	■	■	■	■	■	■	·	·	·	·	·	0
■	■	■	■	■	■	■	■	■	■	·	·	0
■	■	■	■	■	■	■	■	■	■	■	·	0
□	□	□	□	□	□	□	□	□	□	□	□	1
□	□	□	□	□	□	□	□	□	□	□	⊗	1

(a) The first binary search
(b) The second binary search

Figure 4.10: The two binary searches performed by GBR on our example. Each row indicates a call to P . \square in n 's column means that \mathcal{D}_n was part of a successful call to P , and \blacksquare means that set was part of an unsuccessful call to P . The last row is the result of the binary search, where \boxtimes indicates the final selected \mathcal{D}_r .

our sixth invocation of P . In Figure 4.10b, we run our second binary search over the prefixes of the progression. Again we find that the entire progression is needed to satisfy P , but we learn that $[\mathbf{M.x}()!\text{code}]$ has to be part of the solution.

Now $\mathcal{L} = \{\{[\mathbf{A.m}()!\text{code}]\}, \{[\mathbf{M.x}()!\text{code}]\}\}$ and $J = \dot{\mathcal{D}}_{\leq 11}^{\cup}$. We recompute the progression and we get that the initial set in the progression adds $[\mathbf{M.x}()!\text{code}]$ and $[\mathbf{I.m}()]$. The latter is added because $[\mathbf{M.x}()!\text{code}]$ mentions it.

$$\ddot{\mathcal{D}}_0 = \left\{ \begin{array}{l} [\mathbf{A} \triangleleft \mathbf{I}], [\mathbf{A.m}()!\text{code}], [\mathbf{A.m}()], [\mathbf{A}], [\mathbf{I.m}()], [\mathbf{I}], \\ [\mathbf{M.main}()!\text{code}], [\mathbf{M.main}()], [\mathbf{M.x}()!\text{code}], [\mathbf{M.x}()], [\mathbf{M}] \end{array} \right\}$$

The rest of the progression is unimportant because we run our eleventh (11) and last invocation of P on $\ddot{\mathcal{D}}_0$, and this time it succeeds. $\ddot{\mathcal{D}}_0$ is the optimal solution we presented in Section 4.2. We run our *reduce* function on it and produce the sub-input in Figure 4.1b. We can see that all the variables in \mathbf{M} are in the solution, so \mathbf{M} remains the same. We can remove \mathbf{B} entirely because $[\mathbf{B}]$ is not in the solution. Finally, we can see that $[\mathbf{I.m}()]$ and $[\mathbf{A.m}()]$ are not in the solution, so we remove the \mathbf{m} methods from both \mathbf{I} and \mathbf{A} . The rest of the variables in \mathbf{A} and \mathbf{I} is part of the solution, so we do not reduce them further.

4.5 Experimental Evaluation

This section answers this research question:

Does the use of propositional logic for modeling internal dependencies lead to an effective and efficient reduction of complex inputs in practice?

To which the answer is: yes! Our tool reduces Java bytecode to 4.6% of its original size, which is 5.3 times better than the 24.3% achieved by J-Reduce. It does this while only being 3.1 times slower. If we only want the amount of reduction produced by J-Reduce, we can achieve that with our new reducer in only 6 minutes. This is below 10% of the total running time of J-Reduce.

4.5.1 Experimental Setup

Our implementation and evaluation of this chapter are written as an extension to the artifact [KP19b, KP19a], which we produced for the previous chapter. Our logical model is built in a Haskell eDSL and is around 800 lines of code.

Benchmarks We use the benchmarks from J-Reduce’s artifact, which is a collection of 100 programs from the NJR project [PL18], together with three decompilers. We have removed four benchmarks from the benchmarks set. Three of them because they did not type check. This was not a problem in the previous chapter, because it did not type check the programs. The fourth is a reimplementaion of the standard library, which caused all kinds of precedence problems between it and the real standard library, so we excluded it.

In this evaluation, a decompiler is buggy if the output does not compile. The goal of the evaluation is to reduce the input program while preserving the full error message of the compiler. We have changed the formatting of the error messages in four ways:

1. We made the maximum number of reported bugs by the compiler explicit by setting it to 100, which ensures that it stays constant across compiler versions.

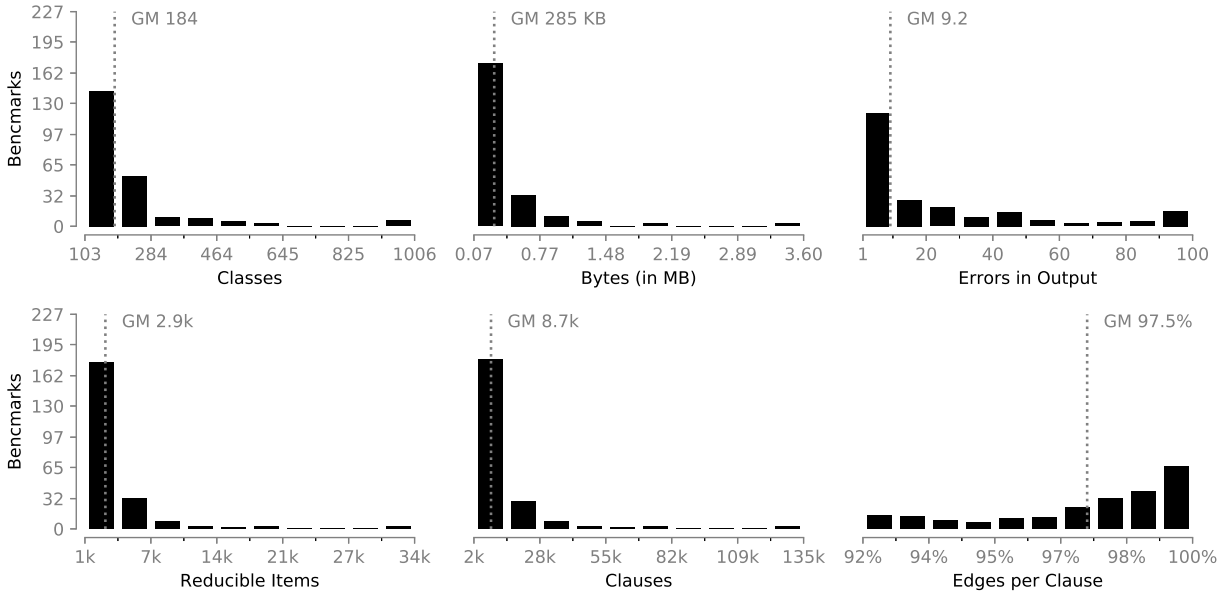


Figure 4.11: The distribution of benchmarks over number of classes, number of bytes, number of lines of errors in the output, number of reducible items, number of clauses in the model, and the percentage of the clauses that can be represented as edges in a graph.

2. We removed all warnings from the compiler output. Code transformations can make the decompilers produce warnings, which made the predicate non-monotone.
3. We removed line numbers and detailed bug descriptions from the output. The line numbers and the detailed bug descriptions of the bugs can change if we remove items before that line in the code.
4. We sorted the output, to limit some of the nondeterminism in the decompilers.

Stats In total, the benchmarks contain 227 instances where the decompilers produce source-code that does not compile. We can distribute these over the different metrics we use. In Figure 4.11, we see histograms of the number of classes, bytes, lines of error produced by the compiler, reducible items, clauses, and finally the percentage of clauses that are edges, per benchmarks. A clause can be represented as an edge in a graph if there exactly one positive and negative literal in the clause. We also include the geometric mean for each metric, which is 184 classes, 285 KB, 9.2 errors produced by the compiler, 2.9k reducible

items, 8.7k clauses in the model, and 97.5% edges per clause.

Running the Benchmarks To support our findings, we have evaluated two reduction strategies:

- *J-Reduce*: A modification of the implementation of J-Reduce, from the previous chapter, which writes the class-files instead of using symbolic links. This gives better bytecode compression and allows for direct comparison with our techniques.
- *Our Reducer*: Our new reducer with the model from Section 4.3 and algorithm from Section 4.4.

We ran each of the strategies in parallel in batches of 8 on every benchmark. We did this concurrently on three 24 Intel(R) Xeon(R) Silver 4116 CPU, 2.10 GHz core machines with 188 GB RAM. The machines ran OpenJDK version 1.8.0_222.

We also ran the original unmodified implementation of J-Reduce, to check that it is equivalent to our implementation except for the final size in bytes. On our modification of the implementation of J-Reduce, we get 12.3% smaller bytecode while only getting a little worse performance (4.5%). As expected, the number of classes after reduction are equivalent (<1% difference).

4.5.2 Analysis

To get an overall sense of the run, we have plotted a cumulative frequency diagram of each of the three metrics: time spent reducing, and the final relative size in both number classes and bytes left, see Figure 4.12. By inspecting the first figure, we can see that J-Reduce finishes running on all benchmarks within an hour, while for some benchmarks, our new reducer takes up to 10 hours. We can, however, see that it has finished on most (>95%) of the benchmarks within two hours. For this extra running time, we get much more reduction. We can see that we reduce half of the benchmarks to below 10% in classes and 5% in bytes, where J-Reduce only reduce to around 40%.

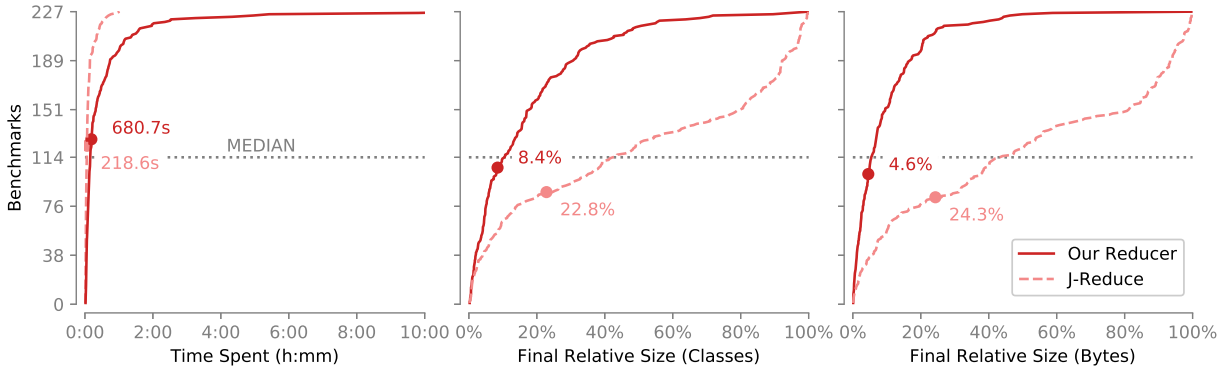


Figure 4.12: Cumulative frequency diagrams of the time spent and relative final size, both in term of number of classes and number of bytes. In all figures, steeper is better. The dots represents the geometric mean.

We can see that J-Reduce’s and our new reducers geometric mean (GM) running time is 218.6 s and 680.7 s, respectively, which means that our new reducer is 3.1 times slower than J-Reduce. The reduction of our new reducer is much better: for number of classes, we can reduce to 8.4% while J-Reduce gets 22.8%, and for bytes we reduce to 4.6% while J-Reduce gets 24.3%. We perform 2.7 times better on classes, and 5.3 times better on bytes.

However, this comparison is only fair if we assume that we have 10 hours to reduce. A much more likely scenario is that we have a fixed time window, and we want the algorithm to reduce as much as it can in that time frame. We can stop both algorithms at any point in the execution and use the smallest input until that point that preserves the error message. To illustrate this, in Figure 4.13, we have plotted the mean reduction over time. The top diagrams show the geometric average over the first two hours of the reduction. Because removing 10 out of 100 classes from a program does not have the same worth as removing 10 out of 11 classes, we also present the bottom two diagrams that more accurately depict the value of the reduction. In the bottom two diagrams, the x-axis is linear over the number of times the reducer has made the benchmarks smaller. The two horizontal lines represent the average reduction after one hour and after every run has completed.

From these graphs, we can conclude two interesting facts. Our reducer performs the bulk of the reduction in both classes and bytes in the first hour. The remaining 9 hours only finds

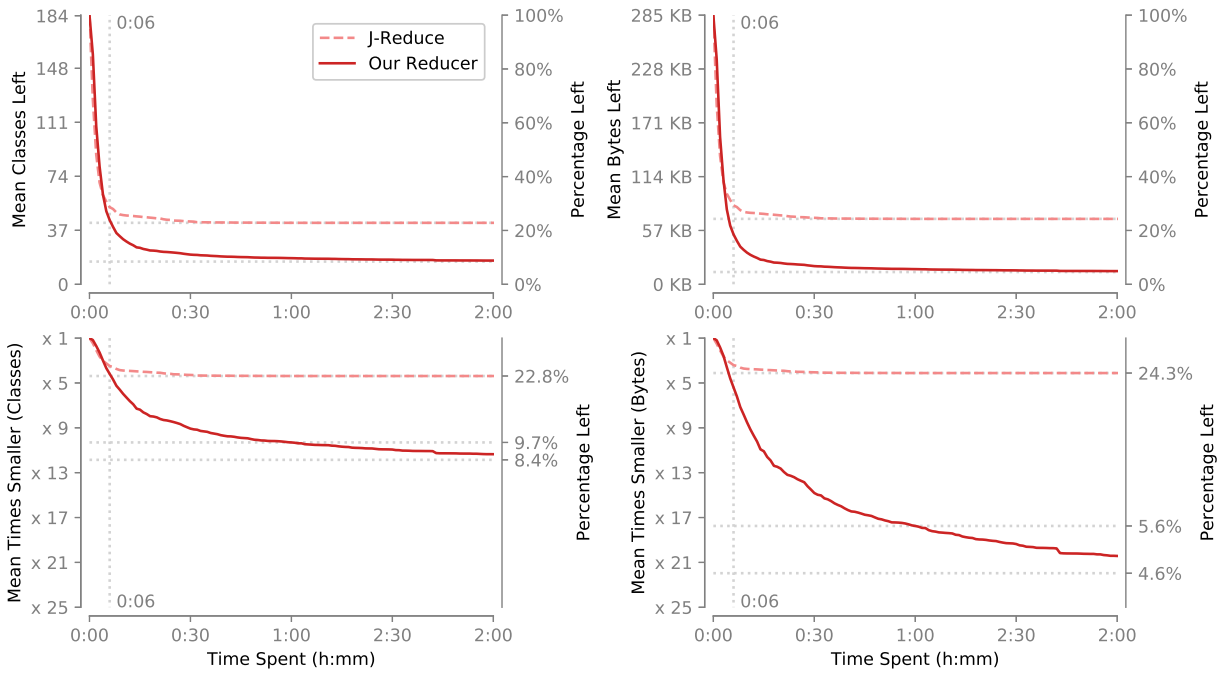


Figure 4.13: The reduction over time. The two top diagrams show the reduction on a linear scale in the number of classes and bytes left. The bottom two diagrams show the reduction on a linear scale of the number of times the item has gotten smaller.

an input with 1.15x fewer classes and 1.22x fewer bytes. The second fact is that we can see that our new reducer outperforms J-Reduce’s total reduction in both classes and bytes after only 6 minutes of reduction. This means that, on average, if a user has more than 6 minutes, it would be better to run our new reducer than to run J-Reduce to completion.

4.6 Related Work

We will now discuss related work. We have found four categories where the findings of this chapter relates to other work: input reduction, fuzz testing, internal input reduction, debloating, type-safe code transformations, and search-based testing.

4.6.1 Input Reduction

We have already talked about `ddmin` [ZH02], `HDD` [MS06b], and `J-Reduce` [KP19b] in the introduction.

`C-Reduce` [RCC12b], to this date, produces the smallest reductions for C, it achieves this using 30 custom-made transformations, one of which is a variant of delta debugging. These rules are C-specific and not easily used with other languages. `C-Reduce` runs through these transformations one by one, in a loop, until it reaches a fix-point. Because of this, and because `C-Reduce` may produce invalid inputs, it is slow. In contrast, `GBR` is general and applies to any form of input for which we can represent dependencies using propositional Boolean logic.

Sun et al.’s tool `Perses` [SLZ18], models the input using a grammar. Compared to the other techniques, it allows them to model the validity of the input more closely. For example, they can specify that at least one element in a list of elements must remain. Their technique is, however, limited to inputs that can be described using a grammar. Our technique is only limited by the expressiveness of propositional Boolean logic: we can model Java’s type system, which `Perses` cannot. `Perses` also uses the grammar to do simple transformations such as promoting syntactic sub-elements. We believe that we can do something similar with

a more complicated *reduce* function and syntactic dependencies. We leave this for future work.

Chisel is a tool that uses machine learning to learn the underlying dependency graph while reducing a C program [HLP18]. Future work could address whether a Chisel-like technique can learn dependencies expressed using propositional Boolean logic. This would make the technique applicable to Java bytecode. Chisel is a tool that uses machine learning to learn the underlying dependency model while reducing the input.

4.6.2 Fuzz Testing

There exist a vast work on fuzz testing, which modifies valid inputs to find bugs in a program. Even though fuzz testing focuses on finding bugs and not reducing existing input, some of the techniques are related.

Our technique is mostly related to black-box fuzzing, which has no knowledge of the system under test. To avoid testing invalid inputs, some of the approaches model the input validity using grammars [Ait02, HHZ12, ZGB19]. However, because they do not model the more complex dependencies in the input, they are often stuck testing the same validation paths in the program. To avoid this, white-box fuzz testing [GLM08, GKL08] records the program’s behavior while running to let it affect the fuzzing. Concolic engines [SA06, CDE08, Kal15] builds in symbolic path constraints to help choose inputs that find new paths.

In contrast, our work can encode complex semantic constraints in the model without inspecting the program.

Because we use propositional logic, we can not reason about variables that do not exist yet. This means that our approach can only generate valid sub-inputs from a valid input, which is ideal for reduction, but a more expressive model language is needed if we want to use our reduction approach for fuzzing. We’ll leave this for future work.

Since we know that combining grammar-based black-box techniques with white-box techniques gives better results [DDG07, GKL08, MGM19], it would be interesting to combine

our technique with white-box techniques, to get a better reduction. We'll also leave this for future work.

4.6.3 Input Generation and Internal Reduction

In Claessen and Hughes's QuickCheck [CH11], they randomly generate input using a specification created by the user. When QuickCheck finds an input that produces a fault, it is often quite large, and QuickCheck tries to reduce the input, using a sub-input generator provided by the user. These generators are hard to write and can often be slow. Hedgehog [Sta17] is a successor to QuickCheck, which can automatically reduce the inputs from the generator. When Hedgehog finds a faulty input, it tries to make it smaller by choosing smaller inputs to the generator. This is known as internal test-case reduction because the reduction is internal to the input generation. Because every reduced input is generated from the internal input, using the generator, it avoids producing invalid inputs. Hypothesis [MD20] is a similar tool implemented in python. Contrary to Hedgehog, which chooses smaller inputs for each sub-generator, they use a sequence of bits as the input to the generator. This allows them more flexibility in moving and removing substrings of the internal input.

While Hedgehog and Hypothesis should not produce invalid inputs, in theory, there are cases where they do. If a precondition defined by the user fails, or if Hypothesis produces too few bits. Recently, Zest [PLS19], showed that they were able to combine generator-based technique with whitebox fuzz testing techniques. This allowed them to do automatic discovery of semantic constraints to the input. They do however not combine this information with reduction. It would be interesting to augment internal reduction with our approach by allowing users to specify preconditions as constraints while writing the generator. We'll leave that for future work.

Furthermore, compared to these tools, our technique works on any input and not only inputs we have generated. This is useful if the input is part of a bug reported by a user.

4.6.4 Debloating

We can view debloating or application extraction as a special case of input reduction. We can use our tool as a debloater in the following way. Given a test suite, we define the predicate P in Definition 2 to be true if all tests pass. Using it this way, we would guarantee that the application preserves the behavior described by the test-suite.

Tip et al. introduced the first successful debloater for Java bytecode, named Jax [TLS99]. Besides removing methods and fields, they also inlined method calls and collapsed the class hierarchy. Since Jax is an entirely static technique, it has problems with dynamic features like reflection. In our tool, a missing target method in a reflection call would fail the predicate, which would trigger a search for the target method. Bruce et al. made a comparison between previous debloating techniques [BZA20], including Jax [TLS99], TamiFlex [BSS11], ProGuard [Gua20], and JRed [JWL16]. They concluded that most static debloaters are not semantic preserving. They also introduced their tool JShrink, which uses a checkpointing technique to maintain semantics. Checkpointing reverts any transformations that do not preserve the behavior of the test-cases. JShrink reduces program to 86,7% of their original size; however, the benchmark suite is very different from ours.

Porter et al.’s recent tool BlankIt has a novel technique on debloating where they try to load as little code as possible while running the program [PMB20], instead of removing as much as they can statically. The goals of the paper and this chapter are different, as BlankIt seeks to limit the access to safety-volubilities in C code, where we seek to produce smaller inputs.

The most significant difference between the tools mentioned here and our tool is that we are not limited to maintaining the program’s semantics. We can reduce using other predicates.

4.6.5 Type-Safe Code Transformations

When the input itself is a program, input reduction is an example of a program transformation. In particular, our reducer for Java bytecode is a program transformation, which by design is type-safe (Theorem 2) but may change the semantics of the program. This makes it different from a long line of work on type-safe, *semantics-preserving* program transformations. Good examples from that line of work include Morrisett et al.’s type-safe translation from System F to typed assembly language [MWC98], Glew and Palsberg’s type-safe method inlining [GP04], and Chen, Chugh, and Swamy’s type-safe compiler that helps enforce security properties [CCS10]. On the technical side, our proof of type safety differs from the proofs in the cited papers in the following way. While the cited papers prove that each typed program is transformed into a single typed program, we prove that an entire family of sub-programs all type check.

4.6.6 Search-Based Testing and Model Transformations

Another approach to retain valid inputs is using *search-based testing* or *Model Transformations*, which works by having a set of possible complex, valid transformations of the code. These can now searched through possible valid programs, using iterative A* [Kor85], Generic Algorithms [OKS12], Particle swam optimization [KSB08, KSB12], and many other techniques [ATF09]) to be directed towards an ideal solution or gain full coverage of all tests. C-Reduce [RCC12b], which we have already discussed, uses a version of search-based refactoring.

However, the approach is slow; full path coverage scales exponentially with the depth of the problem with the number of valid transformations as branching factors. Compared to our work, we present a full model of all valid sub-inputs of a valid input that produces a bug in a program, while a search-based technique can only know about the input it has generated. In other words, we are navigating a maze using a map, while search-based techniques have to find their way without one. Having this model allows us to plot a course and go directly

after the smallest possible input. The drawback of our technique is that it cannot generate new inputs, and the goal, a minimal satisfying input, is predetermined.

4.7 Summary

In this chapter, we have shown that the use of propositional logic for modeling internal dependencies leads to an effective and efficient reduction of complex inputs.

We have done this by modeling the type-system and many other dependencies of a Featherweight Java with Interfaces and proving the reduction correct. We have shown, experimentally, that the model extends to Java in full. Using logical dependencies, we can model Java closer than previous work. Our novel polynomial-time reduction algorithm, Generalized Binary Reduction, can use this validity model to get 5.3 times better results.

Work on other complex inputs is still needed, but we have opened a path to reduce inputs that were not effectively reduceable before. Our reduction engine is general and can be applied to all inputs, which can be modeled using propositional Boolean logic.

CHAPTER 5

Conclusion

In this dissertation, we have looked at three examples of modeling the validity of input programs to verify and reduce bug reported to metaprograms. We were able to verify bug reports correctly, and we reduced bug reports 12 times faster and got more than five times smaller results. We have solved the Input Validity Problem for reducing the classes of Java, Featherweight Java with Interfaces, and Java Bytecode without generics. We believe that the techniques described in this dissertation can be used on many other metaprograms and their input programs and that we have pushed the envelope of how to model the validity of useful inputs to metaprograms.

There are many avenues for future work. The biggest problem with our easiness analysis is the time to run the static analyses, which can be quite time-consuming. Integrating easiness analysis with machine learning techniques could be used approximate the hard features of a language.

Previous work on reduction is using Grammars to presents a simple interface for the user to use. Currently, we write our model as an e-DSL in Haskell. Future work on improving the logical reduction interface, and maybe augment the grammar-based approach to accept logical semantic dependencies is needed before the technique is easily adaptable.

Finally, we have mostly focused on reduction and not transformations, which means that our reduction model can only remove items from the code and not do interesting things like collapsing the hierarchy or inlining. Some minimal inputs are only possible using these techniques. Future work is needed to fit these more complicated transformations in a framework that uses a logical validity model.

Bibliography

- [AH90] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *ACM SIG-Plan Notices*, volume 25, pp. 246–256. ACM, 1990.
- [Ait02] Dave Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, **105**:106, 2002.
- [AMN] E. S. Andreassen, A. Møller, and B. B. Nielsen. Systematic approaches for increasing soundness and precision of static analyzers.
- [Art11] Cyrille Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer*, **13**(3):223–246, 2011.
- [ATF09] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, **51**(6):957–976, 2009.
- [Avi85] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.
- [Ben] Lee Benfield. CFR – another Java decompiler. <http://www.benf.org/other/cfr/> (accessed Aug 24, 2018).
- [BGH14] 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*, pp. 109–120. ACM, 2014.
- [BS09] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of OOPSLA’09, Object-Oriented Programming Systems, Languages and Applications*, pp. 243–262, 2009.

- [BSS11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE, 33rd International Conference on Software Engineering*, May 2011.
- [BZA20] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. JShrink: In-depth investigation into debloating modern Java applications. In *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering — ESEC/FSE '20*. ACM, 2020. [To Appear].
- [CA76] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. *FTCS-8*, (8):3–9, 1976.
- [CCS10] Juan Chen, Ravi Chugh, and Nikhil Swamy. Type-preserving compilation for end-to-end verification of security enforcement. In *Proceedings of PLDI'10, ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2010.
- [CDE08] 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, pp. 209–224, 2008.
- [CH11] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, **46**(4):53–64, 2011.
- [CMW15] Maria Christakis, Peter Müller, and Valentin Wüstholtz. An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer. In *VMCAI'15, Verification, Model Checking, and Abstract Interpretation*, 2015.
- [CZ00] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. *arXiv preprint cs/0012009*, 2000.

- [CZ02] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *ISSTA*, 2002.
- [Cza18] Wiktor Czajkowski. Sneaky bugs and how to find them (with git bisect). *Netguru*, January 2018. <https://www.netguru.co/codestories/sneaky-bugs-and-how-to-find-them>.
- [DDG07] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 185–194, 2007.
- [Dev] Developers. Bisect. <https://git-scm.com/docs/git-bisect> (accessed Aug 24, 2018).
- [DFS15] J. Dolby, S. J. Fink, and M. Sridharan. T. J. Watson libraries for analysis. <http://wala.sourceforge.net>, 2015.
- [DSR17] Jens Dietrich, Li Sui, Shawn Rasheed, and Amjed Tahir. On the construction of soundness oracles. In *SOAP'17, Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2017.
- [ECH01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP, ACM Symposium on Operating Systems Principles*, 2001.
- [ECM06] ECMA. Standard ECMA-355, Common Language Infrastructure. June 2006.
- [FSS13] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *ICSE'13, International Conference on Software Engineering*, pp. 752–761, 2013.

- [GAZ16] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: delta debugging, even without bugs. *Software Testing, Verification and Reliability*, **26**(1):40–68, 2016.
- [GHK17] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1–11. ACM, 2017.
- [GKL08] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 206–215, 2008.
- [GLM08] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pp. 151–166, 2008.
- [GP04] Neal Glew and Jens Palsberg. Type-safe method inlining. *Science of Computer Programming*, **52**:281–306, 2004. Preliminary version in Proceedings of ECOOP’02, European Conference on Object-Oriented Programming, pages 525–544, Springer-Verlag (*LNCS* 2374), Malaga, Spain, June 2002.
- [Gri17] Radu Grigore. Java generics are turing complete. *ACM SIGPLAN Notices*, **52**(1):73–85, 2017.
- [Gua20] Guardsquare. ProGuard. <https://github.com/Guardsquare/proguard>, 2020.
- [HBB15] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 333–344. ACM, 2015.
- [HHZ12] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pp. 445–458, 2012.

- [HLP18] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 380–394. ACM, 2018.
- [IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 132–146, 1999.
- [JWL16] Yufei Jiang, Dinghao Wu, and Peng Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pp. 12–21. IEEE, 2016.
- [Kal15] Christian Gram Kalhauge. Hyperconcolic-finding parallel bugs in java programs, using concolic execution. 2015.
- [Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pp. 85–103. Plenum Press, 1972.
- [KL86] John C Knight and Nancy G Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on software engineering*, (1):96–109, 1986.
- [Kor85] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, **27**(1):97–109, 1985.
- [KP18] Christian Gram Kalhauge and Jens Palsberg. Sound deadlock prediction. In *Proceedings of OOPSLA’18, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2018.

- [KP19a] Christian Gram Kalhauge and Jens Palsberg. Artifact from "Binary Reduction of Dependency Graphs", June 2019.
- [KP19b] Christian Gram Kalhauge and Jens Palsberg. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pp. 556–566, New York, NY, USA, 2019. ACM.
- [KP19c] Christian Gram Kalhauge and Jens Palsberg. Results from "Binary Reduction of Dependency Graphs", February 2019.
- [KSB08] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model transformation as an optimization problem. In *International Conference on Model Driven Engineering Languages and Systems*, pp. 159–173. Springer, 2008.
- [KSB12] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. Search-based model transformation by example. *Software & Systems Modeling*, **11**(2):209–226, 2012.
- [LBK16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016. SEE.
- [Lho07] Ondrej Lhoták. Comparing call graphs. In *PASTE'07, Workshop on Program Analysis for Software Tools and Engineering*, 2007.
- [LOZ07] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 417–420. ACM, 2007.

- [LSS15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhotak, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: A manifesto. *Communications of the ACM*, **58**(2):44–46, February 2015.
- [LSV17] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of Java reflection: Literature review and empirical study. In *ICSE’17, International Conference on Software Engineering*, pp. 507–518, 2017.
- [LTS14] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-Inferencing Reflection Resolution for Java. In *ECOOP’14, European Conference on Object-Oriented Programming*, 2014.
- [LTX15] Yue Li, Tian Tan, and Jingling Xue. Effective Soundness-Guided Reflection Analysis. In *SAS’15, International Static Analysis Symposium*, 2015.
- [LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *Asian Symposium Programming Languages and Systems*, number 0326227, 2005.
- [LZR16] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Precise semantic history slicing through dynamic delta refinement. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pp. 495–506. IEEE, 2016.
- [McK98] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, **10**(1):100–107, 1998.
- [MD20] David R MacIver and Alastair F Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer. In *ECOOP’20*, 2020. [To Appear].

- [MGM19] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschle, and Andreas Zeller. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 548–560, 2019.
- [MRM12] Fadi Meawad, Gregor Richards, Floreal Morandat, and Jan Vitek. Eval Begone ! Semi-Automated Removal of Eval from JavaScript Programs. In *OOPSLA’12, Object-Oriented Programming Systems, Languages and Applications*, 2012.
- [MS06a] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *ICSE’06, International Conference on Software Engineering*, 2006.
- [MS06b] Ghassan Misherghi and Zhendong Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pp. 142–151. ACM, 2006.
- [MWC98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proceedings of POPL’98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 85–97, 1998.
- [MWG15] S McPeak, DS Wilkerson, and S Goldsmith. Berkeley delta. URL <http://delta.tigris.org>, 2015.
- [MZN15] Ravi Mangal, Xin Zhang, Aditya Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of FSE’15, ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2015.
- [OKS12] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. Search-based refactoring: Towards semantics preservation. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 347–356. IEEE, 2012.

- [PL18] Jens Palsberg and Cristina Lopes. NJR: A normalized Java resource. In *SOAP'18, Proceedings of ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2018.
- [PLS19] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 329–340, 2019.
- [PMB20] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 164–180, 2020.
- [PMF10] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. N-version disassembly: differential testing of x86 disassemblers. In *Proceedings of the 19th international symposium on Software testing and analysis*, pp. 265–274. ACM, 2010.
- [RCC12a] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, 2012.
- [RCC12b] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 335–346, 2012.
- [RHB11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval that men do: A large-scale study of the use of eval in JavaScript applications. In *ECOOP'11, European Conference on Object-Oriented Programming*, 2011.
- [RKE18] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In

SOAP'18, Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, 2018.

- [SA06] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification*, pp. 419–423, 2006.
- [Sc] Roman Shevchenko and other contributors. Fernflower. <https://github.com/fesh0r/fernflower> (accessed Aug 24, 2018).
- [SDE18] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the soundness of call graph construction in the presence of dynamic language features – a benchmark and tool evaluation. In *Proceedings of APLAS'18, Asian Symposium on Programming Languages and Systems*, 2018.
- [Sha81] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, **7**(1):67–72, 1981.
- [SKB15] Yannis Smaragdakis, George Kastrinis, George Balatsouras, and Martin Bravenboer. More Sound Static Handling of Java Reflection. In *APLAS'15, Asian Symposium on Programming Languages and Systems*, November 2015.
- [SLZ18] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *ICSE'18, International Conference on Software Engineering*, 2018.
- [Sta17] Jacob Stanley. Hedgehog. <https://hackage.haskell.org/package/hedgehog>, 2017.
- [Str] Mike Strobel. Procyon Java decompiler. <https://bitbucket.org/mstrobel/procyon/overview> (accessed Aug 24, 2018).

- [Thu06] Marc Thurley. sharpsat–counting models with advanced component caching and implicit bcp. In *International Conference on Theory and Applications of Satisfiability Testing*, pp. 424–429. Springer, 2006.
- [TLS99] Frank Tip, Chris Laffra, Peter F Sweeney, and David Streeter. Practical experience with an application extractor for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 292–305, 1999.
- [Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pp. 466–483. Springer, 1983.
- [VGH00] Raja Vallé-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pomerville, and Vijay Sundaresan. Optimizing Java bytecode using the soot framework: Is it feasible? In *Proceedings of CC’00, International Conference on Compiler Construction*. Springer-Verlag (*LNCS*), 2000.
- [Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pp. 439–449. IEEE Press, 1981.
- [XN05] Jingling Xue and Phung Hua Nguyen. Completeness Analysis for Incomplete Object-Oriented Programs. In *CC’05, International Conference on Compiler Construction*, 2005.
- [YCE11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pp. 283–294. ACM, 2011.
- [YLC12] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers’ perspectives. *Journal of Systems and Software*, **85**(10):2305–2317, 2012.

- [ZGB19] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.
- [ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, **28**(2):183–200, 2002.
- [ZHQ18] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. Measuring and disrupting anti-adblockers using differential execution analysis. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2018.