Logical Bytecode Reduction

Christian Gram Kalhauge*
Computer Science Department
University of California, Los Angeles (UCLA)
California, USA
chrg@dtu.dk

Jens Palsberg
Computer Science Department
University of California, Los Angeles (UCLA)
California, USA
palsberg@ucla.edu

Abstract

Reducing a failure-inducing input to a smaller one is challenging for input with internal dependencies because most sub-inputs are invalid. Kalhauge and Palsberg made progress on this problem by mapping the task to a reduction problem for dependency graphs that avoids invalid inputs entirely. Their tool J-Reduce efficiently reduces Java bytecode to 24% of its original size, which made it the most effective tool until now. However, the output from their tool is often too large to be helpful in a bug report. In this paper, we show that more fine-grained modeling of dependencies leads to much more reduction. Specifically, we use propositional logic for specifying dependencies and we show how this works for Java bytecode. Once we have a propositional formula that specifies all valid sub-inputs, we run an algorithm that finds a small, valid, failure-inducing input. Our algorithm interleaves runs of the buggy program and calls to a procedure that finds a minimal satisfying assignment. Our experiments show that we can reduce Java bytecode to 4.6% of its original size, which is 5.3 times better than the 24.3% achieved by J-Reduce. The much smaller output is more suitable for bug reports.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Theory of computation → Logic**.

Keywords: input reduction, type-safe code transformation

ACM Reference Format:
Christian Gram Kalhauge and Jens Palsberg. 2021. Logical Bytecode Reduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454091

*Also with DTU Compute, Technical University of Denmark.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454091

1 Introduction

We have an input to a program that makes the program fail. The input is valid so the program should handle it; however, the input is so massive that we cannot determine the nature of the bug. The process of input reduction, first introduced by Zeller and Hildebrandt [28], addresses this problem by finding a small input that reproduces the failure. Their algorithm, ddmin, produces a *sub-input* (the original input with pieces removed) that reproduces the failure. They found that running on a sub-input can have three outcomes: the failure still happens, the failure is gone, and don't know. The "don't know" outcome happens when the sub-input is invalid, despite that the original input was valid. An invalid sub-input is of no help with finding the bug, as noted by Regehr et al. [22] who coined the term *the test-case validity problem.*

The test-case validity problem can arise for multiple reasons. In the context of finding bugs in C compilers, Regehr et al. [22] noted that C programs can be both statically invalid and dynamically invalid. In particular, they defined a dynamically invalid program to be one that executes "an operation with undefined behavior or rely on unspecified behavior" [22]. Their widely used tool C-Reduce [22] reduces large C programs to programs that are two orders of magnitude smaller and fit for inclusion in bug reports. In the context of finding bugs in tools that process Java bytecode, Kalhauge and Palsberg [13] noted that Java bytecode has many internal dependencies. For example, if a Java method constructs an object from a class, then it depends on that class: without the class, the method no longer type-checks. Their tool J-Reduce [13] reduces large Java bytecode programs to programs that are four times smaller but often still too large for the reader of a bug report.

Is reduction inherently harder for Java bytecode? For C, the most challenging component of the test-case validity problem is dynamically invalid programs. C-Reduce solves this by using a semantics-checking C interpreter to detect dynamically invalid input. For Java bytecode, the biggest challenge is the many internal dependencies. J-Reduce solves this by creating a dependency graph that avoids statically invalid inputs entirely. Thus, the reduction challenges for C and Java are different, and so far, C reduction has been more successful. We will show that more detailed modeling of internal dependencies can bring the effectiveness of Java bytecode reduction much closer to that of C reduction.

```
class A implements I {
  String m(){ /* bug */ }   B n(){ ... }
}
class B implements I {
  String m(){ ... }         B n(){ ... }
}
interface I { String m(); B n(); }
```

(a) The input program.

```
class A implements I {
  String m() { /* bug */ }
}
interface I { String m(); }
```

(b) The optimal reduction.

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

(c) Shared by both (a) and (b).

Figure 1. The example input program which 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.

We build on a long tradition of gradually modeling more and more of the internal dependencies of the input to avoid invalid sub-inputs. In Misherghi and Su [17]'s paper on hierarchical delta debugging (HDD), they avoided many invalid inputs by exploiting the syntax tree of the inputs. Sun et al. [25] took this a step further and used a syntax tree as the model in their tool Perses, which enabled the tool to avoid all syntactically invalid sub-inputs. Beyond models of syntax, Kalhauge and Palsberg [13] used a dependency graph to model semantic dependencies.

In this paper, we introduce a new model of dependencies that goes beyond dependency graphs. Consider, for example, the Java program in Figure 1a, which is the input to a tool and makes the tool crash. While a programmer quickly can reduce Figure 1a to Figure 1b, automatic reduction based on a dependency graph produces suboptimal reductions. 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`. Note that because the example is small, line-oriented reduction techniques such as ddmin may well able to reduce Figure 1a to Figure 1b. For larger examples with many internal dependencies, ddmin tends to produce disappointing results [13].

In this paper, 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 1b. The claim of our paper is:

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

In a typical case from Kalhauge and Palsberg [13], J-Reduce reduces the number of lines in the decompiled program from 7,661 to 6,918, while our tool reduces it to 815. This is a difference of almost an order of magnitude.

After a dive into the example in Figure 1 which illustrates why previous techniques are unable to model all dependencies (Section 2), we have structured the rest of this paper 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 which previous techniques are unable to model. We then discuss the extensions needed to model Java bytecode (Section 3).
- We introduce the Generalized Binary Reduction algorithm, which given a model of the internal dependencies of a failure-inducing input, finds a valid failure-inducing sub-input in polynomial time. Our algorithm interleaves runs of the buggy program and calls to a procedure that finds a minimal satisfying assignment. We have proved the correctness, polynomial time complexity, and an optimality property of the algorithm (Section 4).
- We have implemented our approach and evaluated it on the benchmarks from the J-Reduce paper. Our tool reduces to 4.6% of the original size, while J-Reduce only reduces to 24.3%. This is 5.3x more reduction than J-Reduce (Section 5).

Finally, we go over related work in Section 6 and conclude in Section 7.

Most of our proofs are in supplementary material. Our implementation and benchmarks are available [14].

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 1a. It contains a Java source program, which, when compiled, functions 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

Variables:

In A: [A] [A ◁ I] [A.m()] [A.m()!code] [A.n()] [A.n()!code]. *In B:* [B] [B ◁ 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] ⇒ [A.n()]	[A.n()] ⇒ [A]	[A.m()!code] ⇒ [A.m()]	[A.m()] ⇒ [A]
[B.n()!code] ⇒ [B.n()]	[B.n()] ⇒ [B]	[B.m()!code] ⇒ [B.m()]	[B.m()] ⇒ [B]
[A ◁ I] ⇒ [A]	[B ◁ I] ⇒ [B]	[I.m()] ⇒ [I]	[I.n()] ⇒ [I]
[M.x()!code] ⇒ [M.x()]	[M.x()] ⇒ [M]	[M.main()!code] ⇒ [M.main()]	[M.main()] ⇒ [M]

Referential Semantic Dependencies:

[A ◁ I] ⇒ [I]	[B ◁ I] ⇒ [I]	[A.n()] ⇒ [B]	[B.n()] ⇒ [B]
[I.n()] ⇒ [B]	[M.x()] ⇒ [I]	[M.x()!code] ⇒ [I.m()]	[M.x()!code] ⇒ [I]
	[M.main()!code] ⇒ [M.x()]	[M.main()!code] ⇒ [A]	[M.main()!code] ⇒ [M]

Non-Referential Semantic Dependencies:

[A ◁ I] ∧ [I.m()] ⇒ [A.m()] [A ◁ I] ∧ [I.n()] ⇒ [A.n()] [B ◁ I] ∧ [I.m()] ⇒ [B.m()] [B ◁ I] ∧ [I.n()] ⇒ [B.n()]
[M.main()!code] ⇒ [A ◁ I] [M.main()!code]

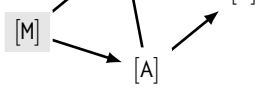
Figure 2. The variables (20) and dependency constraints (32 + 1 duplicate (gray)). The constraints is conjoined.

always requires `M.main()` to run at all. We want to reduce the input program while preserving the error.

Kalhauge and Palsberg [13] described an approach to reduce Java bytecode. Their tool, J-Reduce, models the dependencies between classes using a graph, which allows them to produce smaller results an order-of-magnitude faster than ddmin [28]. The modeling language is a conjunction of required classes [A] and dependencies between classes [A] ⇒ [B]. Every transitive closure in the graph (henceforth called simply a *closure*) corresponds to a valid sub-input. Their algorithm proceeds in five steps, which we quote from [13]:

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."

In step (1), the algorithm derives dependencies from the program: if a class *A* mentions a class *B*, then we have a dependency from *A* to *B*. In step (4), the reduction algorithm can be ddmin [28], binary reduction [13], etc. For the example in Figure 1a, the class dependencies are: [M], ([M] ⇒ [A]), ([M] ⇒ [I]), ([A] ⇒ [I]), ([A] ⇒ [B]), ([B] ⇒ [I]), and ([I] ⇒ [B]). Or in a graph:



Since we want to preserve the code of `M.main()` we require [M]. However, the result is disappointing: the graph only have one closure that contains [M]: the one that contains all classes. So, we cannot reduce the input using the technique from Kalhauge and Palsberg [13].

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 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 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. Kalhauge and Palsberg [13] 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. 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.

Differently from the previous work on graphs, items are nested. The nested structure of the items means that we cannot remove an 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 list them in fig. 2.

Additional Dependencies. The syntactic and referential dependencies by themselves are not enough to model valid inputs correctly. We can see this in Figure 3, which is the dependency graph created from the syntactic and referential dependencies. This graph contains closures that are invalid 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 generate 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 a `C.m()`. We need a more general concept for defining dependencies.

Our First Contribution. In our 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 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:

$$[\texttt{M.main()!code}] \Rightarrow [\texttt{A} \lhd \texttt{I}]\,.$$

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()`:

$$[\texttt{I.m()}] \wedge [\texttt{A} \lhd \texttt{I}] \Rightarrow [\texttt{A.m()}]\,.$$

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 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 in Section 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. In our formulation, [x] is a variable that indicates whether 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. We can represent a dependency graph as a conjunction of implications and variables, as we saw above; from now on we will refer to such a constraint as a *graph constraint*. A graph constraint can be converted to the grammar above by converting each edge [x] ⇒ [y] to ¬([x] ∧ ¬ [y]).

We can iterate through all satisfying truth assignments to the constraints in Figure 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 1b:

$$[\texttt{A} \lhd \texttt{I}]\,,[\texttt{A.m()}]\,,[\texttt{A.m()!code}]\,,[\texttt{A}]\,,[\texttt{I.m()}]\,,[\texttt{I}]\,,$$
$$[\texttt{M.x()!code}]\,,[\texttt{M.x()}]\,,[\texttt{M.main()!code}]\,,[\texttt{M.main()}]\,,[\texttt{M}]$$

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 [26]. 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. Additionally, any attempt to decrease the number of runs is hampered by the fact that the union of two satisfying truth assignments is not always a satisfying truth assignment. Our Generalized Binary Reduction algorithm (Section 4) finds the optimal solution by checking only 11 inputs.

3 Modeling Dependencies

We will formalize how we model dependencies and we will discuss aspect of our implementation that go beyond the formal model.

Featherweight Java with Interfaces. Featherweight Java with Interfaces (FJI) is a modest extension of Featherweight Java [9]: each class implements a single interface. An interface consists of a collection of signatures. While we can model the dependencies of Featherweight Java with graph constraints, 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 3.1).

For examples in FJI, our formalization generates the same constraints as our implementation. In particular, the core of

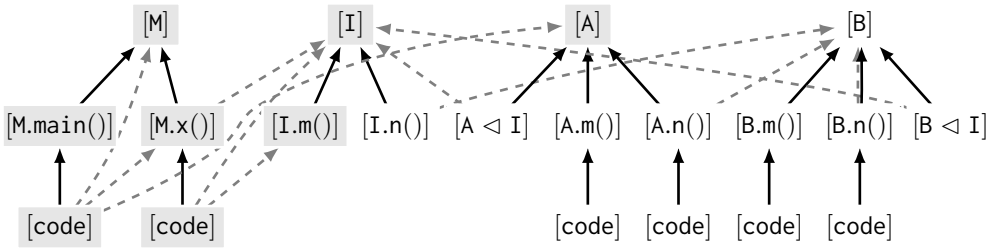


Figure 3. The dependency graph containing syntactic (solid, black) and referential (dashed, gray) dependencies. We have abbreviated the code variables to [code]. We have shaded the variables part of the minimal closure from M.

the example in Section 2 is an FJI program, and our formalization generates the constraints listed in Section 2, as we will show in Section 3.

Syntax. Figure 4 shows the grammar of FJI. Our metanotation for Featherweight Java is similar to the one used in the original paper on Featherweight Java [9]; we refer the reader to that paper for details.

Figures 6 and 7 show the helper rules and type rules of FJI. In the type rules, we use Γ to range over type environments, that is, mappings from identifiers to types. We use π, σ to range over logical formulas. We use the abbreviation $\overline{\pi} = \pi_1 \wedge \ldots \wedge \pi_n$. 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:

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

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 $V(P) \to \texttt{Bool}$. The idea is that $\varphi([\texttt{C}]) = \mathbf{1}$, 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 ◁ 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 5 implements the idea of the Boolean variables explained above. The reducer forms the core of our implementation for Java bytecode. For any mapping $\varphi : V(P) \to \mathit{Bool}$, we construct a reduced program $\mathit{reduce}(P, \varphi)$.

Generating Type-checking Constraints. Figures 6 and 7 show the type rules. For a program P, we write $\vdash P \mid \pi$ 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 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 $\mathit{mAny}(P, m, C)$ is a disjunction of variables that all are of the form $[C.m()]$. If we need C to implement a method m in the reduced program, then we can require $\mathit{mAny}(P, m, C)$ to be true. This will ensure that the reducer will preserve at least one such method m.

The type rules for FJI in Figure 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 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.

$$
\begin{array}{rcll}
P & ::= & \overline{R}\ e & \textit{programs} \\
R & ::= & L \mid Q & \textit{type declarations} \\
T, U & ::= & C \mid I & \textit{type names} \\
L & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\} & \textit{classes} \\
Q & ::= & \texttt{interface}\ I\ \{\overline{S}\} & \textit{interfaces} \\
K & ::= & C(\overline{T}\ \overline{f})\ \{\ \texttt{super}(\overline{f});\ \texttt{this}.\overline{f} = \overline{f};\ \} & \textit{constructors} \\
M & ::= & T\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \} & \textit{methods} \\
S & ::= & T\ m(\overline{T}\ \overline{x}); & \textit{signatures} \\
e & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \texttt{new}\ C(\overline{e}) \mid (T)\ e & \textit{expressions}
\end{array}
$$

Figure 4. The syntax of Featherweight Java with Interfaces (FJI).

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

Figure 5. Our *reduce* function of FJI.

- 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, for compatibility with our implementation for full Java.

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

We leave to future work to settle whether the converse of Theorem 3.1 holds. For example, the converse statement could read: *if* $\vdash P \mid \sigma$ *and* $\vdash \textit{reduce}(P, \varphi) \mid \sigma'$, *then* $\varphi \models \sigma$. Such a result would indicate that the constraint matches the type system.

Generating the Constraints in the Example. The code in Figure 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. Now we show highlights of how we generate the constraints in Figure 2.

From the program typing rules (Figure 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. 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 $[\texttt{A}] \Rightarrow [\texttt{Object}]$. The second conjunct generates the constraints for the `implements` statement $([\texttt{A} \lhd \texttt{I}] \Rightarrow ([\texttt{A}] \wedge [\texttt{I}])) \wedge \overline{\pi} \wedge \overline{\tau}$. Now let's focus on the methods requirements $\overline{\pi}$. There are two methods in `A`, `String m() {...}` and `B n () {...}`. For `m` we use the method typing rule to see that: $([\texttt{A.m()}] \Rightarrow ([\texttt{A}] \wedge [\texttt{String}]))$ and $([\texttt{A.m()!code}] \Rightarrow [\texttt{A.m()}] \wedge \pi_1 \wedge \pi_2)$ For

Field lookup

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

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

Method type lookup

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

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

$$\frac{P(I) = \texttt{interface}\ I\ \{\overline{S}\} \qquad U\ m(\overline{U}\ \overline{x}) \in \overline{S}}{\mathit{mtype}(P, m, I) = (\overline{U} \to U)}$$

Method choice

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

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

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

$$\frac{P(I) = \texttt{interface}\ I\ \{\overline{S}\} \qquad U\ m(\overline{U}\ \overline{x}) \in \overline{S}}{\mathit{mAny}(P, m, I) = [\texttt{I.m()}]}$$

Subtyping

$$P \vdash T \le T\ \mid\ \mathbf{1} \qquad \frac{P \vdash T \le T'\ \mid\ \pi_1 \qquad P \vdash T' \le T''\ \mid\ \pi_2}{P \vdash T \le T''\ \mid\ \pi_1 \wedge \pi_2}$$

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

$$\frac{P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}}{P \vdash C \le I\ \mid\ [\texttt{C} \lhd \texttt{I}]}$$

Valid method overriding

$$\frac{\mathit{mtype}(P, m, D) = \overline{U} \to U \text{ implies } \overline{U} = \overline{T} \text{ and } U = T}{\mathit{override}(P, m, D, \overline{T} \to T)}$$

Figure 6. FJI helper rules.

Program typing

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

Class typing

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

Interface typing

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

Method typing

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

Signature typing

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

Signature typing relative to a class

$$\frac{\mathit{mtype}(P, m, C) = \overline{T} \to T}{\begin{array}{l}P \vdash T\ m(\overline{T}\ \overline{x})\ \text{OK in}\ I\ \text{for}\ C\ \mid \\ ([\texttt{C} \lhd \texttt{I}] \wedge [\texttt{I.m()}]) \Rightarrow \mathit{mAny}(P, m, C)\end{array}}$$

Expression typing

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

$$\frac{\begin{array}{c}P, \Gamma \vdash e : T\ \mid\ \pi_1 \qquad \mathit{mtype}(P, m, T) = \overline{U} \to U \\ P, \Gamma \vdash \overline{e} : \overline{T}\ \mid\ \overline{\pi} \qquad P \vdash \overline{T} \le \overline{U}\ \mid\ \pi_2\end{array}}{P, \Gamma \vdash e.m(\overline{e}) : U\ \mid\ [\texttt{T}] \wedge \pi_1 \wedge \mathit{mAny}(P, m, T) \wedge \overline{\pi} \wedge \pi_2}$$

$$\frac{\mathit{fields}(P, C) = \overline{T}\ \overline{f} \qquad P, \Gamma \vdash \overline{e} : \overline{U}\ \mid\ \overline{\pi} \qquad P \vdash \overline{U} \le \overline{T}\ \mid\ \pi}{P, \Gamma \vdash \texttt{new}\ C(\overline{e}) : C\ \mid\ [\texttt{C}] \wedge \overline{\pi} \wedge \pi}$$

$$\frac{P, \Gamma \vdash e : U\ \mid\ \pi}{P, \Gamma \vdash (T)\ e : T\ \mid\ [\texttt{T}] \wedge \pi}$$

Figure 7. FJI type rules.

n we can see: $([\texttt{A.n()}] \Rightarrow ([\texttt{A}] \wedge [\texttt{B}]))$ and $([\texttt{A.n()!code}] \Rightarrow [\texttt{A.n()}] \wedge \pi_1 \wedge \pi_2)$

We assume for these two methods that the expression (π_1), and the return cast (π_2) create no constraints. With that out of the way, we create the constraints $\overline{\tau}$ 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.

$$\begin{aligned}([\texttt{A} \lhd \texttt{I}] \wedge [\texttt{I.m()}]) &\Rightarrow \mathit{mAny}(P, m, A) \\ &= ([\texttt{A.m()}] \wedge \mathit{mAny}(P, m, \texttt{Object})) \\ &= [\texttt{A.m()}]\end{aligned}$$

The same happens for n: $([\texttt{A} \lhd \texttt{I}] \wedge [\texttt{I.n()}]) \Rightarrow [\texttt{A.n()}]$.

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 can generate constraints for `B`, `I`, and `M` in a similar fashion.

Altogether, we have generated 31 of the 32 constraints from Figure 2. We add the last constraint ([`M.main()!code`]) after constraint generation because we know that the tool will not work without the body of [`M.main()`].

Java Bytecode. We have implemented a reducer and a constraint generator for Java bytecode for which our model of FJI is the core. We have a total of 11 kinds of items that can be removed, including constructors, fields, and super-class relations. Constructs that require special attention during constraint generation include abstract classes, interfaces extending other interfaces, classes implementing multiple interfaces, and type casts. Additionally, we have to model Java generics and type inference. Type checking with Java generics is undecidable [6] and our approach approximates the type checking problem as part of the constraints so the undecidability is a limiting factor for us. We have designed an approximation that has worked well in our experiments. In particular, the model of Java generics has so far never led our tool to produce an invalid sub-input.

For example, consider the following source-level declaration of a local variable and the corresponding bytecode:

```
Class<? extends B> a = A.class  ⟶
            ldc [class A]
```

When we compile the source code, we get a load-constant instruction (`ldc`) and none of the generic-type information. For the source code, we could have generated a constraint that preserves that `A` extends `B`, but in the bytecode we have to approximate it. We do that by making all method bodies that do reflection on the class `A` depend on that `A` extends all its superclasses.

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.

4.1 Notation

We use small letters to refer to variables v, capital letters to refer to sets L, and the calligraphic letters $\mathcal{D}, \mathcal{L}$ to reference to sets of sets. 2^X indicates the power set of a set X. In lists we can access the nth element with a subscript $\mathcal{D}_n$.

A *solution* M is a satisfying assignment to a logical statement R, which we write $R(M)$. We write solutions as the set of *true* variables. For example $(x \wedge \neg y)(\{x\})$ is true and $(x \wedge \neg y)(\{x, y\})$ is false. We can also get the variables $\text{VARS}(R)$ of a logical statement. 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 $(R \mid X = \mathbf{1})$.

A conjunctive normal form (CNF) is a representation of a logical expression using a conjunction of clauses. A clause is a disjunction of literals and a term is a conjunction of literals. A literal is a normal or negated variable. Furthermore, we define the following shorthands:

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

4.2 Formalizing the Problem

Our formalization of the input reduction problem is akin to the formalizations in [13, 17], as we explain below. First, we represent an input as a set of variables I and we represent sub-inputs as subsets of I. In Section 3, this set I was called $V(P)$, and each sub-input was represented by a truth assignment φ defined on $V(P)$. We represent the program that may have a bug as a *black-box* predicate $\mathcal{P}$ that is true on a sub-input if and only if that sub-input induces a bug in the tool. The notion of black-box means that we can invoke $\mathcal{P}$ on subsets of I but we cannot inspect $\mathcal{P}$ in any other way. The notion of a black-box predicate models that we have to run the tool to know anything about it. Finally, we have a propositional Boolean formula R_I over I. In Section 3, this formula R_I was called σ and was generated from the input.

Definition 4.1 (Input Reduction Problem)**.** *Instance:* $(I, \mathcal{P}, R_I, k)$, where I is a set of variables, $\mathcal{P}$ is a black-box predicate on subsets of I, and R_I is a Boolean formula in CNF over I, and k is an integer representing a maximum cost. *Assumptions:* $\mathcal{P}$ can be evaluated in polynomial time, both $\mathcal{P}(I)$ and $R_I(I)$ are true, and $\mathcal{P}$ is monotonic on valid sub-inputs: if $X \subseteq Y$ and $R_I(X)$ and $R_I(Y)$, then $\mathcal{P}(X) \Rightarrow \mathcal{P}(Y)$. *Problem:* decide $\exists S \subseteq I : \mathcal{P}(S) \wedge R_I(S) \wedge |S| < k$.

Our input reduction problem differs from the one of Mishergi and Su [17] in that we model input validity and we require $\mathcal{P}$ to be monotonic on valid sub-inputs. Our input reduction problem also differs from the one of Kalhauge and

Palsberg [13] in that we have no cost function on the sub-inputs. However, all three input reduction problems are NP-complete and for the same reason: we can easily reduce the Hitting Set Problem (which is NP-complete [15]) to each of the input reduction problems.

Theorem 4.2. *The Input Reduction Problem is NP-complete.*

In the remainder of this section, we will focus on the optimization version of the Input Reduction Problem. Specifically, we will present polynomial-time algorithms that each finds a *small* solution to $(I, \mathcal{P}, R_I)$.

4.3 Lossy Encodings into Graph Constraints

For our benchmarks, 97.5% of the clauses are graph constraints (see Section 5). We know that the binary reduction algorithm [13] relies on that *all* the constraints are graph constraints. Can we give a lossy encoding of the remaining 2.5% of the clauses as graph constraints and then apply the binary reduction algorithm? Yes we can, as follows. Those other 2.5% of the clauses are of the form $(\wedge_{i=1}^{n} a_i) \Rightarrow (\vee_{j=1}^{m} b_j)$, where $n > 1 \vee m > 1$. For any i', j', where $1 \leq i' \leq n$ and $1 \leq j' \leq m$, we can approximate such a clause by the graph constraint $a_{i'} \Rightarrow b_{j'}$. This is because

$$(a_{i'} \Rightarrow b_{j'}) \quad \Rightarrow \quad [(\wedge_{i=1}^{n} a_i) \Rightarrow (\vee_{j=1}^{m} b_j)]$$

Thus, if we have a solution to $(a_{i'} \Rightarrow b_{j'})$, then it is also a solution to $(\wedge_{i=1}^{n} a_i) \Rightarrow (\vee_{j=1}^{m} b_j)$. This means that if we replace $(\wedge_{i=1}^{n} a_i) \Rightarrow (\vee_{j=1}^{m} b_j)$ with $(a_{i'} \Rightarrow b_{j'})$ and apply binary reduction, we will get a valid result. In Section 5 we show experiments with two variations of the lossy encoding: one where we pick $(i' = 1, j' = 1)$ in all cases, and one where we pick $(i' = n, j' = m)$. Both of them give much better results than J-Reduce [13].

For example, consider Figure 2, which has four clauses that go beyond graph constraints:

$$[\texttt{A} \lhd \texttt{I}] \wedge [\texttt{I.m()}] \Rightarrow [\texttt{A.m()}] \qquad [\texttt{A} \lhd \texttt{I}] \wedge [\texttt{I.n()}] \Rightarrow [\texttt{A.n()}]$$
$$[\texttt{B} \lhd \texttt{I}] \wedge [\texttt{I.m()}] \Rightarrow [\texttt{B.m()}] \qquad [\texttt{B} \lhd \texttt{I}] \wedge [\texttt{I.n()}] \Rightarrow [\texttt{B.n()}]$$

If we pick $(i' = 1, j' = 1)$ in all cases, we replace the above clauses with these ones:

$$[\texttt{A} \lhd \texttt{I}] \Rightarrow [\texttt{A.m()}] \qquad [\texttt{A} \lhd \texttt{I}] \Rightarrow [\texttt{A.n()}]$$
$$[\texttt{B} \lhd \texttt{I}] \Rightarrow [\texttt{B.m()}] \qquad [\texttt{B} \lhd \texttt{I}] \Rightarrow [\texttt{B.n()}]$$

If we add these graph constraints to the other graph constraints in Figure 2, then running binary reduction will preserve both [B] and [A.m()], which is nonoptimal.

The lossy encodings ignore important elements of the constraints, which raises the question of whether we can do better. For example, we might use a SAT-solver to produce a minimal satisfying assignment. But, we also have to run the black-box predicate, preferably at most a polynomial number of times, which cannot be guaranteed by a SAT-solver. Now we present a new algorithm that both does better than the lossy encodings and runs the black-box predicate a polynomial number of times.

Algorithm 1: Generalized Binary Reduction

Input: $(I, \mathcal{P}, R_I)$ where $\mathcal{P}(I)$ and $R_I(I)$.
Output: $\mathcal{D}_0 \subseteq I$, where $\mathcal{P}(\mathcal{D}_0)$ and $R_I(\mathcal{D}_0)$.
Data: The variable order $\preceq$ (a total order of I).
Data: The learned sets $\mathcal{L} \subseteq 2^I$ and the current progression $\mathcal{D} \in List(2^I)$.

$\mathcal{L} \leftarrow \emptyset$
$\mathcal{D} \leftarrow \text{PROGRESSION}_{R_I}(\mathcal{L}, I)$
while $\neg\mathcal{P}(\mathcal{D}_0)$ **do**
  $r \leftarrow \min_r \mathcal{P}(\mathcal{D}^{\cup}_{\leq r})$
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathcal{D}_r\}$
  $\mathcal{D} \leftarrow \text{PROGRESSION}_{R_I}(\mathcal{L}, \mathcal{D}^{\cup}_{\leq r})$
end
return $\mathcal{D}_0$

$\text{PROGRESSION}_{R_I}(\mathcal{L}, J)$ calculates $(\mathcal{D}_0, \mathcal{D}_1, \ldots)$:

$$\mathcal{D}_0 = \text{MSA}_{\preceq}(R^+)$$
$$\mathcal{D}_{i+1} = \text{MSA}_{\preceq}(R^+ \wedge x \mid \mathcal{D}^{\cup}_{\leq i} = \mathbf{1}) \textbf{ if } \exists x \in \min_{\preceq} J \backslash \mathcal{D}^{\cup}_{\leq i}$$

$R^+ = R_I \wedge \bigwedge_{L \in \mathcal{L}} L^{\vee}$ with vars not in J set to $\mathbf{0}$

4.4 Generalized Binary Reduction

Generalized Binary Reduction (GBR, Algorithm 1) solves the Input Reduction Problem approximately in polynomial time. GBR uses two building blocks: evaluation of the black-box predicate $\mathcal{P}$ and computation of an approximate *minimal* satisfying assignment (MSA). We define minimal to mean that the MSA assigns true to as few variables as possible [21], which is an NP-complete problem so we settle for an approximate solution. GBR learns from the outcomes of calls to $\mathcal{P}$ and MSA, which enables it to pick good inputs to later calls.

The Main Algorithm. GBR extends Binary Reduction [13] with a subroutine PROGRESSION that produces a *progression* of valid inputs. A progression is a list where every prefix represents a valid input (INV-PRO). GBR applies the black-box predicate $\mathcal{P}$ to only prefixes of progressions, hence only to valid inputs.

GBR maintains three data structures. First, the variable order $\preceq$ (a total order of I) helps the main loop terminate in polynomial time; it also helps us design $\text{MSA}_{\preceq}$ that runs in polynomial time (see the appendix). Second, the current progression $\mathcal{D}$ represents the current search space. The search space satisfies the black-box predicate $\mathcal{P}$, and $\mathcal{D}$ divides the search space into a non-empty list of disjoint subsets (INV-$\mathcal{D}$). Third, the *learned sets* in $\mathcal{L}$ represent the growing knowledge about $\mathcal{P}$. The main loop maintains that every learned set *both* overlaps with every valid sub-input that satisfies $\mathcal{P}$ (INV-$\mathcal{L}$), *and* overlaps with every prefix of the progression (INV-PRO).

Lemma 4.3 (Invariant). *The main loop of GBR on* $(I, \mathcal{P}, R_I)$ *has the invariant* $\mathrm{Inv}(\mathcal{L}, \mathcal{D})$*:*

$$\mathcal{P}(\mathcal{D}^{\cup}) \ \wedge \ |\mathcal{D}| > 0 \ \wedge \ \mathcal{D}^{\cup} \subseteq I \\ \wedge \ (\forall i, j.\ i \neq j \Rightarrow \mathcal{D}_i \cap \mathcal{D}_j = \emptyset) \qquad \texttt{(INV-}\mathcal{D}\texttt{)}$$

$$\left(\forall T \subseteq \mathcal{D}^{\cup}.\ \mathcal{P}(T) \wedge R_I(T) \Rightarrow \forall L \in \mathcal{L}.\ T \cap L \neq \emptyset\right) \qquad \texttt{(INV-}\mathcal{L}\texttt{)}$$

$$\left(\forall r \geq 0.\ R_I(\mathcal{D}^{\cup}_{\leq r}) \ \wedge \ \forall L \in \mathcal{L}.\ \mathcal{D}^{\cup}_{\leq r} \cap L \neq \emptyset\right) \qquad \texttt{(INV-PRO)}$$

The idea of the main loop is that the more we learn about $\mathcal{P}$, the better chance we have of finding a valid input that satisfies $\mathcal{P}$. The main loop checks whether the first set $\mathcal{D}_0$ in the progression satisfies $\mathcal{P}$. If $\mathcal{D}_0$ does satisfy $\mathcal{P}$, then the main loop returns it, and otherwise, the main loop executes three steps.

First, the main loop finds the minimal prefix $\mathcal{D}^{\cup}_{\leq r}$ of the progression that satisfies $\mathcal{P}$. Such a prefix exists because the entire progression satisfies $\mathcal{P}$, and we can find it efficiently by binary search.

Second, the main loop adds the prefix' last set $\mathcal{D}_r$ to $\mathcal{L}$. Since $\mathcal{P}$ is monotone and $\mathcal{D}_r$ is the shortest prefix of $\mathcal{D}$ that satisfies $\mathcal{P}$, at least one element in $\mathcal{D}_r$ must be part of any solution within the current search space. So, adding $\mathcal{D}_r$ to $\mathcal{L}$ maintains (INV-$\mathcal{L}$). This set $\mathcal{D}_r$ is new to $\mathcal{L}$, which we can see after a few steps of reasoning, as follows. Notice that since $\neg\mathcal{P}(\mathcal{D}_0)$ and $\mathcal{P}(\mathcal{D}^{\cup}_{\leq r})$, we have $r \neq 0$, hence $\mathcal{D}_r \cap \mathcal{D}_0 = \emptyset$ (INV-$\mathcal{D}$). From this and that $\mathcal{D}_0$ overlaps with all sets in $\mathcal{L}$ (INV-PRO), $\mathcal{D}_r$ must be missing at least one element in each of the sets in $\mathcal{L}$. So, $\mathcal{D}_r \notin \mathcal{L}$.

Third, we build a new progression using $\mathcal{L}$ and $\mathcal{D}^{\cup}_{\leq r}$.

The Progression Subroutine. If $J \subseteq I$, then the call $\mathrm{PROGRESSION}_{R_I}(\mathcal{L}, J)$ produces a non-empty list of disjoint subsets of J whose union is J.

The first step is to map R_I to a stronger constraint R^+, by conjoining it with a clause for each set in $\mathcal{L}$, and limiting R^+ to only have variables in J, by setting all other variables to false. Now, every satisfying assignment to R^+ is a solution to R_I and overlaps with all sets in $\mathcal{L}$.

The second step is to build the progression recursively. The first entry of the progression is an approximate MSA of R^+. The $k+1$'th entry is constructed by picking a variable that doesn't occur in the earlier entries and then constructing an approximate MSA of $(R^+ \wedge x)$, while setting all the variables used in the earlier entries to true. The recursion ends when we run out of variables.

Correctness. When GBR returns $\mathcal{D}_0$, we have $\mathcal{P}(\mathcal{D}_0)$ and $\mathcal{D}_0 = \mathrm{MSA}_{\preceq}(R^+)$, hence $R_I(\mathcal{D}_0)$.

Execution time. We know that $\mathcal{D}_0$ contains an element from every $L \in \mathcal{L}$. Indeed, as we prove in detail in the appendix, $\mathcal{D}_0$ contains the $\preceq$-smallest variable in each set in $\mathcal{L}$. When the main loop adds $\mathcal{D}_r$ to $\mathcal{L}$, we know that $\mathcal{D}_r$ is new to $\mathcal{L}$ and that $\mathcal{D}_r \cap \mathcal{D}_0 = \emptyset$. So, adding $\mathcal{D}_r$ has a $\preceq$-smallest variable that is different from the $\preceq$-smallest variables in the other sets in $\mathcal{L}$. This means that the maximum number of times we can add a set $\mathcal{D}_r$ to $\mathcal{L}$ is equal to the number of variables. So, the main loop terminates after at most $|I|$ iterations.

Each iteration of the main loop evaluates $\mathcal{P}$ a polynomial number of times, and each of those evaluations runs in polynomial time, by assumption. Additionally, each iteration computes $\mathrm{MSA}_{\preceq}$ a polynomial number of times, and each of those computations takes polynomial time. So, the grand total is that GBR does at most $|I|$ iterations that each runs in polynomial time, hence GBR runs in polynomial time.

Theorem 4.4. *GBR finds an approximate solution to the Input Reduction Problem in polynomial time.*

Note that our use of $\preceq$ can make $\mathcal{D}_0$ larger than the smallest possible solution. For example, consider $(a \wedge b \Rightarrow c) \wedge (c \Rightarrow b)$, in which $(a \wedge b \Rightarrow c)$ is not a graph constraint. Define the predicate $\mathcal{P}$ to be true iff b is true; assume that every subset of $\{a, b, c\}$ is valid; and pick the variable order (c, b, a). The first progression is $(\{b, c\}, \{a\})$, so our algorithm returns $\{b, c\}$. This is suboptimal: a smaller solution is $\{b\}$.

Minimality for graph constraints. In the appendix we show that if all the clauses in R_I are graph constraints and we pick $\preceq$ well, then GBR produces a solution that is locally minimal. Here, *locally minimal* means that no proper subset of the solution satisfies $\mathcal{P}$.

Theorem 4.5. *If R_I consists of only graph constraints, then GBR produces a locally minimal solution.*

4.5 Running on the Example in Section 2

Now we show a run GBR on the example in Section 2. First we compute a variable order and the initial progression. We have $\mathcal{L} = \emptyset$ and $J = I$, so $R^+ = R_I$, and we get D_0:

$$\{\ [\texttt{A}], [\texttt{A} \lhd \texttt{I}], [\texttt{I}], [\texttt{M}], [\texttt{M.x()}], \\ [\texttt{M.main()}], [\texttt{M.main()!code}]\ \}$$

The rest of the progression is calculated using $\mathrm{MSA}_{\preceq}$ on $R \wedge x \mid \mathcal{D}^{\cup}_{\leq k} = \mathbf{1}$ with $x \in \min_{\preceq} J \setminus \mathcal{D}^{\cup}_{\leq k}$. For our first choice after $\mathcal{D}_0$, we choose $x = [\texttt{B}]$, because it is the smallest variable in $J \setminus \mathcal{D}_0$. $[\texttt{B}]$ implies no new variables, so the set $\mathcal{D}_1 = \mathrm{MSA}_{\preceq}(R \wedge [\texttt{B}] \mid \mathcal{D}_0 = \mathbf{1}) = \{[\texttt{B}]\}$. We calculate the rest of the progression in the same way. We have annotated each set with the number it is in the progression:

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

This 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 $\mathcal{P}$ for the first time on $\mathcal{D}_0$: no bug! Then we run a binary search over the prefixes of the progression to find the shortest one that satisfies $\mathcal{P}$. First, we try the prefix $\mathcal{D}^{\cup}_{\leq 7}$, which fails. So the correct choice must be between 7 and 13. In binary-search fashion, we cut the search space in half and try $\mathcal{D}^{\cup}_{\leq 10}$, which also fails. After two more tries, we conclude that the shortest satisfying prefix is the full progression. While this didn't reduce the size of the search space, we learned something important: [A.m()!code] has to be in the solution. So, we add {[A.m()!code]} to $\mathcal{L}$.

Now we compute the next 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 $\dot{\mathcal{D}}_0 = \mathcal{D}_0 \cup \{[\texttt{A.m()!code}], [\texttt{A.m()}]\}$ where we add [A.m()!code] because it is in $\mathcal{L}$ and we add [A.m()] because we have [A.m()!code] $\Rightarrow$ [A.m()] from the constraints. The rest of the progression is straightforward:

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

We now start the second iteration of the algorithm. We try $\mathcal{P}(\dot{\mathcal{D}}_0)$, which is false. This is our sixth invocation of $\mathcal{P}$. Now we run our second binary search over the prefixes of the progression. Again we find that the entire progression is needed to satisfy $\mathcal{P}$, and we learn that [M.x()!code] has to be part of the solution.

Now $\mathcal{L} = \{\{[\texttt{A.m()!code}]\}, \{[\texttt{M.x()!code}]\}\}$ and $J = \dot{\mathcal{D}}^{\cup}_{\leq 11}$. We compute another progression and we get $\ddot{\mathcal{D}}_0 = \dot{\mathcal{D}}_0 \cup \{[\texttt{M.x()!code}], [\texttt{I.m()}]\}$

The rest of the progression is unimportant because we run our eleventh (11) and last invocation of $\mathcal{P}$ on $\ddot{\mathcal{D}}_0$ and this time it succeeds. Indeed, $\ddot{\mathcal{D}}_0$ is the optimal solution we presented in Section 2. Finally, we run our *reduce* function on it and produce the sub-input in Figure 1b. We can see that all the variables in M are in the solution, so M remains the same. We can remove B entirely because [B] is not in the solution. Finally, we can see that [I.m()] and [A.m()] are not in the solution, so we remove the m methods from both I and A. The other variables in A and I are part of the solution, so those items stay.

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 reducer in only 6 minutes. This is below 10% of the total running time of J-Reduce.

Implementation. Our implementation and evaluation are written as an extension to the J-Reduce artifact [12, 13]. 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 [19], 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 J-Reduce paper, because it did not type check the programs. The fourth is a copy of the standard library, which caused us problems.

In this evaluation, a decompiler is buggy if the output does not compile. Each of a total of 94 input programs causes at least one of the decompilers to produce an output that does not compile. The goal of the evaluation is to reduce the input program while preserving the full error message of the compiler. A risk to validity is that multiple benchmarks may lead to failure because of the same bug in a decompiler, in which case the results may be skewed.

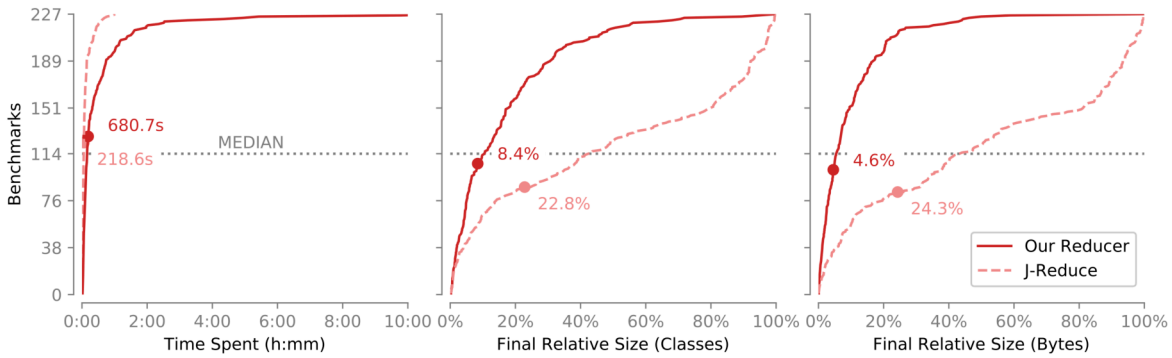
Statistics. In total, the benchmarks contain 227 instances where the decompilers produce source-code that fails to compile. A clause can be represented as an edge in a graph if there exactly one positive and negative literal in the clause. On average (geometric mean), those benchmarks have 184 classes, 285 KB, 9.2 errors produced by the compiler, 2.9k reducible items, 8.7k clauses in the model, and 97.5% edges among the clauses.

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

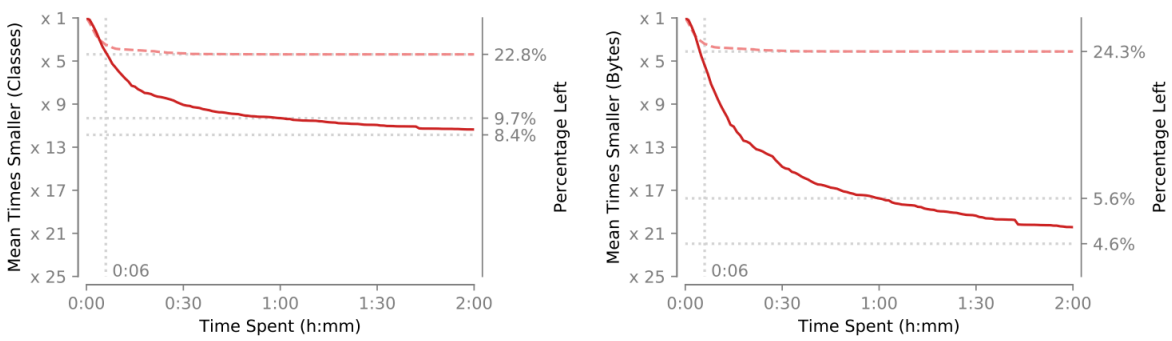
- *J-Reduce:* Our modification of the implementation of J-Reduce, which writes the class-files instead of using symbolic links.
- *Two Lossy Encodings:* The model from Section 3, the two lossy encodings from Section 4.3, and then our modification of J-Reduce.
- *Our Reducer with GBR:* The model from Section 3 and our GBR algorithm from Section 4.4.

We ran 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.

Analysis. We first compare J-Reduce with our reducer. 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 8a. By inspecting the first figure, we can see that J-Reduce finishes



(a) 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.



(b) The reduction over time. Shows the reduction on a linear scale of the number of times the item has gotten smaller.

Figure 8. Our results

running on all benchmarks within an hour, while for some benchmarks, our 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%.

The long execution times stem from that decompilers take time to execute and that some cases have many distinct bugs. Each bug requires GBR to do an individual search. One of our long-running cases leads to many distinct bugs and a constraint with 9,207 variables, and we end up doing 73 searches with 13 steps each. In total, that case leads us to run 951 decompilations and compilations in 8 hours, each taking 33 seconds on average.

We can see that J-Reduce's and our reducers geometric mean running time is 218.6 s and 680.7 s, respectively, which means that our reducer is 3.1 times slower than J-Reduce. The reduction of our 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 fig. 8b, we have plotted the mean reduction over time.

The two lossy encodings from Section 4.3 have execution times that are similar to that of our reducer: the first one is 4% faster while the second is 2% slower. Additionally, they are almost as good of our reducer: the first one produces 5% more bytes while the second produces 8% more bytes. Similarly, the first one produces 6% more lines than our reducer while the second produces 8% more lines. Overall, our reducer is strictly better than the first lossy encoding for 48% of our benchmarks, and our reducer is stricly better than the second lossy encoding for 51% of our benchmarks. Those percentages increase to 79% and 84% for benchmarks with at least 5% non-graph constraints.

6 Related Work

Input Reduction. In Section 1 we discussed several tools for input reduction. Additionally, Chisel is a tool that uses machine learning to learn the underlying dependency graph while reducing a C program [8]. Future work could address whether a Chisel-like technique can learn dependencies expressed using propositional Boolean logic.

Internal Reduction. QuickCheck [4] randomly generates input using a specification created by the user. When it finds an input that produces a fault, it tries to reduce it. Hedgehog [24] and Hypothesis [16] are successors to QuickCheck that intelligently generate smaller inputs from scratch, instead of reducing an existing input. This is known as internal test-case reduction because the reduction is internal to the input generation. Compared to these tools, our technique and other input reducers work on any input and not only inputs generated internally.

Debloating. We can use our tool as a debloater in the following way. Given a test suite, we define the black-box predicate in Definition 4.1 to be true if all tests pass. This guarantees that the application preserves the behavior described by the test-suite. We leave to future work to compare such a debloater to tools such as the seminal Jax [27] and more recent debloaters such as JShrink [2], TamiFlex [1], ProGuard [7], JRed [10], and BlankIt [20].

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 type-safe program transformation (Theorem 3.1) that may change the semantics of the program. This makes it different from a long line of work on type-safe, *semantics-preserving* program transformations [3, 5, 18]. 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 one typed program, we prove that a family of sub-programs all type check.

7 Conclusion

We have shown that the use of propositional logic for modeling internal dependencies leads to an effective and efficient reduction of complex inputs. We did that by modeling the type-system of Featherweight Java with Interfaces and proving that a reduced program type checks. Additionally, we have shown experimentally that the model extends to full Java and models Java more closely than previous work. Our polynomial time reduction algorithm, Generalized Binary Reduction, uses this model to get 5.3 times better results. Much of this improvement can be achieved with simple lossy encodings, yet we have found that completing "the final mile" requires a powerful algorithm.

Acknowledgments

We thank Shuyang Liu, Zeina Migeed, Akshay Utture, the anonymous PLDI reviewers, and the first author's PhD committee [11] for helpful suggestions, and we thank our PLDI shepherd Cormac Flanagan for guidance. NSF award 1730697 and ONR award N00014-18-1-2037 supported us.

References

[1] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *ICSE, 33rd International Conference on Software Engineering*.

[2] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. 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.

[3] Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. 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*.

[4] Koen Claessen and John Hughes. 2011. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices* 46, 4 (2011), 53–64.

[5] Neal Glew and Jens Palsberg. 2004. Type-Safe Method Inlining. *Science of Computing Programming* 52 (2004), 281–306. Preliminary version in Proceedings of ECOOP'02, European Conference on Object-Oriented Programming, pages 525–544, Springer-Verlag (*LNCS* 2374), Malaga, Spain, June 2002.

[6] Radu Grigore. 2017. Java Generics are Turing Complete. *ACM SIGPLAN Notices* 52, 1 (2017), 73–85.

[7] Guardsquare. 2020. ProGuard. https://github.com/Guardsquare/proguard.

[8] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 380–394.

[9] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 132–146.

[10] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 12–21.

[11] Christian Gram Kalhauge. 2020. *Reporting Bugs in Metaprograms*. Ph.D. Dissertation. University of California, Los Angeles (UCLA).

[12] Christian Gram Kalhauge and Jens Palsberg. 2019. Artifact from "Binary Reduction of Dependency Graphs". https://doi.org/10.5281/zenodo.3262201

[13] Christian Gram Kalhauge and Jens Palsberg. 2019. 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)*. ACM, New York, NY, USA, 556–566. https://doi.org/10.1145/3338906.3338956

[14] Christian Gram Kalhauge and Jens Palsberg. 2021. Artifact and Dataset from "Logical Bytecode Reduction". https://doi.org/10.5281/zenodo.4679316

[15] Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher (Eds.). Plenum Press, 85–103.

[16] David R. MacIver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights From the Hypothesis Reducer. In *ECOOP'20*.

[17] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software engineering*. ACM, 142–151.

[18] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. 85–97.

[19] Jens Palsberg and Cristina Lopes. 2018. NJR: A Normalized Java Resource. In *SOAP'18, Proceedings of ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*.

[20] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. 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*. 164–180.

[21] Kavita Ravi and Fabio Somenzi. 2004. Minimal Assignments for Bounded Model Checking. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 31–45.

[22] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 335–346.

[23] Micha Sharir. 1981. A Strong-Connectivity Algorithm and its Applications in Data Flow Analysis. *Computers & Mathematics with Applications* 7, 1 (1981), 67–72.

[24] Jacob Stanley. 2017. Hedgehog. https://hackage.haskell.org/package/hedgehog.

[25] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *ICSE'18, International Conference on Software Engineering*.

[26] Marc Thurley. 2006. SharpSAT–Counting Models with Advanced Component Caching and Implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 424–429.

[27] Frank Tip, Chris Laffra, Peter F Sweeney, and David Streeter. 1999. Practical Experience with an Application Extractor for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 292–305.

[28] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.