Static Program Reduction via Type-Directed Slicing

ANONYMOUS AUTHOR(S)

A traditional program slicer constructs a smaller variant of a target program that computes the same result with respect to some target variable—that is, program slicing preserves the original program's *run-time semantics*. We propose *type-directed slicing*, which constructs a smaller program that guarantees that a typechecker will produce the same result on the sliced program when considering only a target program location—that is, a type-directed slicer preserves the target program's *compile-time semantics*, from the view of a specific typechecker, with respect to some location.

Type-directed slicing is a useful debugging aid for designers and maintainers of typecheckers. When a typechecker produces an unexpected result (a crash, a false positive warning, a missed warning, etc.) on a large codebase, the user typically reports a bug to the maintainers of the typechecker without an accompanying test case showing the analysis' misbehavior in isolation. State-of-the-art approaches to this *program reduction problem* are dynamic: they require repeatedly running the typechecker on the full program. A type-directed slicer solves this problem statically, without rerunning the typechecker, by exploiting the modularity inherent in a typechecker's type rules. Our prototype type-directed slicer for Java is fully-automatic, can operate on incomplete programs, and is fast. It automatically produces a small test case that preserves typechecker misbehavior for 25 of 28 (89%) historical bugs from the issue trackers of three widely-used typecheckers: the Java compiler itself, NullAway, and the Checker Framework; in each of these 25 cases, it preserved the typechecker's behavior even without the classpath of the target program. And, it runs in under a minute on each benchmark, whose size ranges up to millions of lines of code, on a free-tier CI runner.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: program reduction, type systems, typechecker, program minimization

ACM Reference Format:
Anonymous Author(s). 2025. Static Program Reduction via Type-Directed Slicing. 1, 1 (February 2025), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program analysis is an important tool for ensuring the correctness of programs. Many programming languages include a program analysis in their compiler in the form of a static type system, including popular languages like Java, C, and Rust. Other program analyses are also widely-deployed in industry. For example, Airbus uses abstract interpretation [1]; Uber [2] and Amazon [3] use pluggable type systems; Google [4] and Meta [5] have built their own program analysis platforms; etc.

Like all programs, the implementations of program analyses can have bugs. When they do, the maintainers of the analysis desire small test cases that reproduce those bugs. However, a typical report from a user of a static analysis includes the entire program on which the analysis failed. The problem that we address in this work is converting a full program on which an analysis fails at some known location to a small test case on which the analysis fails in the same way, to assist analysis maintainers with debugging. Prior works on this *program reduction problem* like C-reduce [6] and Perses [7] are dynamic: they use a delta-debugging-like algorithm [8] whose "interesting-ness" function is defined by the presence or absence of the analysis behavior of interest. An undesirable

ACM XXXX-XXXX/2025/2-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

property of these dynamic approaches is that they require repeatedly running the analysis on the project in which the failure originally occurred. This property is undesirable for two reasons: 1) analysis run times are often long, making the debugging process slow, and 2) analysis maintainers may not know how to build the program on which the analysis fails. In practice, therefore, many analysis maintainers hand-craft minimized test cases for reported bugs in their analyses.

The key insight in this work is that there is a restricted but popular class of program analyses—typecheckers—with two properties that we can exploit to help solve this problem statically: 1) typecheckers are *modular*: they consider each part of the program in isolation, and require programmers to write summaries (i.e., types); and 2) typecheckers have a well-defined formalism (type rules) that captures their modularity, which we can transform into a *type rule dependency map* from a specific program element e to (a superset of) the other program elements that the typechecker could reason about when analyzing e. This set of program elements is both finite and relatively small: the code immediately around the location, and the summaries (but not the contents) of program elements that are used nearby. This insight implies that we can over-approximate the set of summaries that might be relevant to typechecking a given program location *statically*, while still keeping that set small enough to be a useful test case. This means that we can dramatically reduce the size a program while provably preserving the behavior of a typechecker.

To take advantage of this insight, we propose *type-directed slicing*, a technique that slices based on the dependencies *between the type rules* that are relevant to the target program location, rather than the run-time dependencies. Traditional slicing [9] reduces the size of a program while preserving the run-time behavior of a program with respect to some program location; our type-directed slicing approach preserves the compile-time behavior of a typechecker's analysis of the program, instead. In other words, type-directed slicing preserves all of the types that will be used in a typing judgment for some particular program location.

For practicality, we also introduce a distinction between two kinds of type-directed slicing: exact and approximate. *Exact* type-directed slicing guarantees that analysis behavior is preserved, but requires the user to provide the full input program. In practice, this means providing the object code associated with any libraries that are used (e.g., the classpath for a Java program); it is often inconvenient to recover this from a large project when debugging a typechecker failure. For that reason, *approximate* type-directed slicing can be applied to incomplete programs (i.e., programs where not all symbols are solvable)—instead, it creates symbols as needed—but in turn it weakens the guarantees of the slicer. We have found that while both exact and approximate type-directed slicing are useful in practice, approximate slicing (when it works) is much more convenient.

Our type-directed slicing approach occupies a new part of the design space for program reduction tools. It has two key advantages over state-of-the-art dynamic reduction tools: it is much faster, because it avoids the slowest part of dynamic reduction (repeatedly running the analysis whose behavior is being preserved); and, in approximate mode it can be applied to incomplete programs, making it easier to apply in realistic debugging scenarios. However, it has a key limitation relative to state-of-the-art dynamic tools: it is limited to preserving the behavior of typecheckers, rather than being applicable to an arbitrary program analysis. In summary, our contributions are:

- our novel exact type-directed slicing technique (section 3);
- a proof that exact type-directed slicing preserves the compile-time semantics of its target with respect to a typechecker; intuitively, this proof works because the exact type-directed slicer faithfully preserves anything that a type rule might reason about (theorem 3.4);
- a novel approximate type-directed slicing technique, which relaxes some requirements and guarantees of exact type-directed slicing in exchange for ease of application to complex or incomplete programs, which is useful in realistic debugging scenarios (section 4),

- an implementation of a type-directed slicer for Java typecheckers with both exact and approximate modes, in a tool called TypeSlice (section 5); and
- an evaluation of TypeSlice's effectiveness as a program reduction tool for historical bugs in three typecheckers for Java: the type system in the Java compiler itself, the NullAway nullability analysis [2], and the Checker Framework [10] (a collection of "pluggable" typecheckers that extend the Java type system), which shows TypeSlice's ability to faithfully reproduce analysis behavior in 89% of cases in its approximate mode, as well as its speed: it terminates in an average of 15 seconds on our benchmarks (section 6).

2 MOTIVATING EXAMPLE

To motivate static program reduction, consider issue #3850 of the Checker Framework [11]; fig. 1 shows a screenshot of this bug report. The Checker Framework is a tool for layering additional "pluggable" type systems on top of the base type system supported by Java; it comes with a set of type systems for users to choose from, including a nullness checker that is used by a number of open-source projects. One such open-source project is Apache Calcite [12], an open-source dynamic data management framework. An Apache Calcite maintainer opened issue #3850 on the Checker Framework's bug tracker on GitHub, reporting a regression in Checker Framework version 3.7.1 in the nullness checker, versus the previous 3.7.0 version. The Calcite maintainer provided the stack trace from a crash in the framework and a link to the code in Calcite that was being analyzed when the framework crashed.

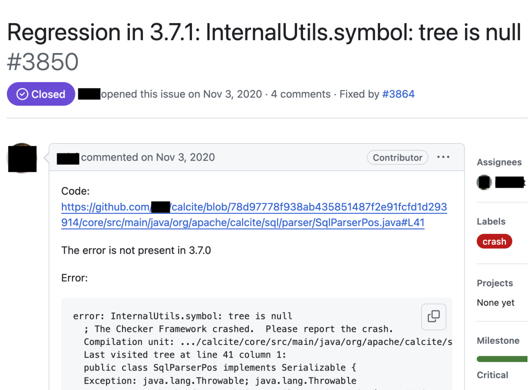

Fig. 1: The initial bug report for the motivating example.

This bug report is typical for an analysis tool that is widely deployed. A user—usually themselves a maintainer of a large project—encounters a regression of some kind from a previous version when they try to update the analyzer to the latest release; in this case, a crash. The user delays upgrading the analysis tool and instead files a bug with the analyzer's issue tracker. It is then up to the analysis developer to build a minimized test case that triggers the bug: it is clearly impractical to test the analyzer by re-running it on the user's entire application (for example, in this case Apache Calcite is about 365,000 lines of non-comment, non-blank Java code).

Creating a minimal test case is not always trivial. The analysis developer needs to look at the stack trace and the corresponding code and build up a model of what parts of the target project matter in reproducing the crash, and what parts do not. Building up this model requires deep expertise in how the analysis tool works: the analysis developer is reasoning "like the tool would" in order to build the test case.

Our key insight is that this "reasoning like the tool would" process need not be manual: for a typechecker, we can formalize *what the tool is allowed to reason about* from the type rules—and thereby automate this process. Formalizing what the analysis is allowed to reason about is the core idea behind our type-directed slicing technique. Of course, the analysis developer has a better grasp on what is relevant to *any particular bug*: in the case of issue #3850, the maintainer is able to write a test case that is only 11 lines of code and still reproduces the crash. Our practical type-directed slicer (TypeSlice) produces a 119 line program *totally automatically*, without any manual effort,

$$\frac{\Gamma \vdash x : C_0 \qquad \text{fields}(C_0) = \overline{C}\,\overline{f}}{\Gamma \vdash x.f_i : C_i}\ \text{T-FIELD}$$

(a) The standard type rule for a field read in a Java-like language, using the formalism of Featherweight Java [13]. The "fields" helper is as defined in Fig. 1 of [13].

$$M_T(x.f_i) = \begin{cases} M_T(C_0) \\ M_T(C_i) \\ \text{declaration of } f_i \text{ in } C_0 \end{cases}$$

(b) The derived type rule dependency map entry for a field read.

Fig. 2: An example type rule from Java and the corresponding entry in our type rule dependency map. Note how the map preserves exactly the structures used in the type rule. The map's entry for a type declaration (which is invoked by $M_T(C_0)$ and $M_T(C_i)$ but is not shown for space reasons) preserves the type declaration itself as well as the declarations of any extended or implemented classes or interfaces. The notation $\overline{C}\,\overline{f}$ denotes a vector of field declarations (a class C and a name f).

that can also reproduce the bug, in just 28 seconds. The reason that TypeSlice's output is larger is that it produces a program that will reproduce *any* bug that the target typechecker could encounter at the target location: that is, the slice that it produces is an *over-approximation* of the program elements required to reproduce the specific bug.

In the anticipated deployment scenario for TypeSlice, a typechecker developer quickly and easily uses TypeSlice to get a reasonably-small test case, and then can further minimize that test case (by hand or with an extant dynamic reducer like Perses [7]) as they work on a fix. The key advantage of TypeSlice in this deployment scenario is that it allows the typechecker developer to skip the tedious and error-prone process of manually extracting an independently-compilable, bug-reproducing test case from the large project in which the bug originally occurred.

3 EXACT TYPE-DIRECTED SLICING

This section describes the core of our type-directed slicing technique: *exact* type-directed slicing, which assumes access to the full input program. An exact type-directed slicer guarantees that its output exactly preserves the behavior of a specific typechecker on its target component(s) (theorem 3.4). It takes as input a program P, a target component C in P, and a typechecker T defined by a set of type rules. We assume in our presentation that P is written in a typical object-oriented programming language with subtyping, single inheritance, type variables, etc., such as Java or C#; the same ideas should be applicable in other language paradigms, but exploring that is beyond the scope of the present work. For simplicity of presentation, we assume that methods are the components of interest.

3.1 Type Rule Dependency Map

The core insight behind our exact type-directed slicing technique is that we can construct a *type rule dependency map* from the type rules T_R of the target typechecker T. Each type rule $t \in T_R$ (e.g., the type rule in fig. 2a) is a pair of a set of *premises* (traditionally written above the line in a typing judgment) and a set of *conclusions* (traditionally written below the line). Given such a set of type rules, we construct a type rule dependency map by intuitively *inverting* them: for a program element e in the conclusions of a rule (such as the field read below the line in fig. 2a), the type rule dependency map maps e to the elements in the premises or elsewhere in conclusions of that type rule that may have been involved in a typing judgement about e (such as the declarations of the types C_0 and C_i in fig. 2a). More formally:

```java
int a = 0;
final int b = two() + 1;
int c = two();
static int two() { return 2; }
void target() {
    c++;
    int d = b + 1;
}
```

(a) The example input. `target()` is the component of interest.

```java
final int b = 0;
int c;
void target() {
    c++;
    int d = b + 1;
}
```

(b) The output produced by TypeSlice for the input in fig. 3a.

Fig. 3: Example input and output described in example 3.2 that uses the field read rule from fig. 2.

Definition 3.1. Given a typechecker T with a set of type rules T_R, a *type rule dependency map* M_T is a function from a program element e to the set of other program elements E that appear in the typing judgment derived via the rules in T_R when typing e.

Intuitively, we can view the type rule dependency map as a function that answers the question "what other program elements are relevant when we derive a typing judgment for e?" It is straightforward to derive a type rule dependency map for a typechecker directly from its type rules. For example, part of the specific type rule dependency map from our experiments (for Java type systems) is given in fig. 2b. We constructed this type rule dependency map by consulting the type rules for Java, such as the T-FIELD rule in fig. 2a (which comes from the Featherweight Java formalism in [13]). Each type rule includes a set of premises (above the line), which indicate the "inputs" to that type rule. For example, the type rule for field reads in fig. 2a includes the type C_0 of the receiver expression x as a premise. To build the map, we translated the premises of each type rule in Java into a rule like the one that appears in fig. 2b. Note how this map directly reflects the *reasoning* that a type system does—for each expression's type rule, the type rule dependency map encodes the facts that the type system uses to typecheck that kind of expression.

Example 3.2. Consider the example input/output programs in fig. 3, assuming that TypeSlice's target component is `target()` (i.e., TypeSlice's goal is to preserve everything in the input program that is needed to typecheck `target()`). First, consider the field `a`, which is not used in `target()`. TypeSlice should remove it entirely, because typechecking `target()` cannot ever require looking up `a`'s type: the typing judgment will never include a reference to it. Next, consider the `final` field `b`, which is used by `target()`. Its declaration (the type and the `final` modifier) is visible to a typechecker, and so must be preserved. However, the initializer expression (to the right of the `=`) is not: the type of the initializer is not relevant for typechecking *uses* of the field, and so it can be removed—except, `final` fields in Java *must* be initialized, so TypeSlice adds a type-compatible initializer to a default value (`0`). `c` is not a final field, so its initializer can be removed entirely. Finally, the `two()` method used in the original initializers of both `b` and `c` can be removed.

Note that the type rule dependency map need not be exact: any *over-approximate* mapping will do. For example, our type rule dependency map for Java only includes the declaration of the field actually being read (f_i) in its rule for field reads, but the premise for the type rule includes all the fields of the containing class (fields(C_0)). It would be sound for our type rule dependency map to include all of the fields and their types (i.e., $\overline{C}\ \overline{f}$), but because the conclusion only uses f_i and C_i it suffices to only include them—the other fields in the premise are extraneous to the typing judgment. With this in mind, we can notice that there are many valid type rule dependency maps for a given typechecker—including a "trivially sound" map that returns the whole input program. Our approach exploits the type rules to derive a relatively "precise" map.

Algorithm 1: The core algorithm for producing a type-directed slice. The slice is iteratively built up, starting from the target component and proceeding outwards according to the type rule dependency map.

input : Program P, component (i.e., set of program elements) $C \in P$, type rule dep. map M_T
output: A type-directed slice of P with respect to C

```
worklist ← C
slice ← ∅
while worklist ≠ ∅ do
    c ← randomChoice(worklist)
    worklist ← worklist \ c
    if c ∉ slice then
        slice ← slice ∪ c
        worklist ← worklist ∪ M_T(c)
return slice
```

We note that similar analyses can share type rule dependency maps—for example, between multiple type systems for the same language, since the type rules generally have the same structure—even if the specific rules differ. Our experiments use the same map for Java's base type system and for seven Checker Framework pluggable type systems (each of which is a distinct analysis proving a different property). To support NullAway's typechecker, we had to make just one small change to this map: NullAway can issue an error in a constructor if there exists a field that is not initialized by the constructor, so NullAway's type rule dependency map must map a class' constructor to all of the fields of that class. A similar map could probably be reused for any other typechecker that targets Java, possibly with minor modifications like the one necessary to faithfully model NullAway, which means that although there is some manual work required to construct the map from the type rules, this cost can be amortized across many typecheckers for the same language.

3.2 Core Algorithm and Its Properties

The worklist algorithm in algorithm 1 is the core of exact type-directed slicing; much like traditional slicing [9], the core algorithm is simple. The inputs are the target program P, a component within it C, and a type rule dependency map M_T for the typechecker whose behavior is to be preserved. The algorithm creates a worklist containing each element of C—for example, if C is a method, then the initial worklist contains the declaration and body of the method—and then iteratively builds the slice outwards from the input component, guided by the type rule dependency map. At each step, algorithm 1 removes an arbitrary program element c from the worklist (in a practical implementation, the worklist is a queue). If that element is already in the slice, then the algorithm continues to the next worklist element. If not, then the element is added to the slice, and then the modularity model is used to update the worklist.

Theorem 3.3. *Algorithm 1 terminates.*

Proof. Algorithm 1 adds each element of the original program to the slice at most once. Assuming the input program is finite, this guarantees termination. □

The run time of algorithm 1 is at most linear in the size of the input program; this worst case occurs if the output slice is exactly equal to the full input program. For type systems, the type rule dependency map can be relatively precise (since it was directly derived from the type rules), so the actual run time of the type-directed slicer is usually (much) faster. It is a pleasant property of type-directed slicing that the algorithm gets *faster* as the output gets closer to minimality—improving the type rule dependency map's precision makes the algorithm both quicker and more useful.

Theorem 3.4. *Type-directed slicing preserves the output of a typechecker T on a target component C, given a sound type rule dependency map M_T for T.*

Informally, theorem 3.4 states that a type-directed slicer guarantees that the behavior of a typechecker will be the same within the target method, regardless of whether we run the typechecker on the type-directed slice or on the original program. That is, theorem 3.4 is a preservation theorem: the behavior of the typechecker is preserved.

Proof. The proof follows directly from the derivation of the type rule dependency map and its soundness. It is always possible to construct a sound type rule dependency map for a type system by deriving it from the type rules. If the type rule dependency map is sound and so over-approximates the related components that appear in the typing judgment derived by T for C, then algorithm 1 must include each related component in the slice. □

If the type rule dependency map is not sound, then type-directed slicing may not preserve the behavior of the typechecker. We discuss intentionally relaxing this soundness requirement in section 4, producing "approximate" type-directed slices. However, the soundness requirement does point to one limitation of type-directed slicing for debugging: when debugging a problem with a typechecker's modularity, type-directed slicing may not be effective.

For both program reduction and slicing, precision is usually defined by *minimality*: that is, how close the slice is to the smallest program that preserves the property of interest (run-time semantics for slicing, typechecker behavior for program reduction). Type-directed slicing does *not* guarantee minimality: allowing over-approximation in the type rule dependency map precludes any such guarantee. In practice, however, we would like our type-directed slices to be relatively small: that is, we would like them not to include too many unnecessary program elements. How close type-directed slicing comes to this ideal depends on the precision of the type rule dependency map: the type rule dependency map "all elements of the input program are related to each other" is over-approximate, but will never lead to small type-directed slices. In section 6.4, we show that our implementation is reasonably precise because it produces slices that are within an order of magnitude of the size of human-minimized test cases.

3.3 Discussion

3.3.1 What Type-Directed Slicing Does Not Preserve. Apart from the target component C, other components in the program are either emptied (if the type rule dependency map only includes their type signatures) or removed. Thus, type-directed slicing *destroys* the run-time behavior of the program completely, making the output program completely unrunnable—type-directed slicing makes no attempt to preserve the program's concrete semantics. However, since type-directed slicing preserves the specifications of the involved elements, it can preserve the compile-time semantics of the input program, which makes sense—we designed type-directed slicing to help typechecker developers with debugging, which does not involve running the analyzed program, anyway. We note that human-written test cases for typecheckers, like those discussed in section 6.4, are also "unrunnable" in this way.

3.3.2 Relationship to Traditional Slicing. A traditional slicer preserves the run-time behavior of a program P at some location L with respect to the language's *concrete semantics*. One way to understand the type-directed slicing algorithm presented above is that it generalizes slicing to an *abstract semantics*—in particular, the abstract semantics of the typechecker. In other words, if we view the typechecker as an abstract interpretation [14] (which we can do, since typechecking and abstract interpretation are isomorphic [15]), then type-directed slicing produces a slice that contains the parts of the program that can contribute to the abstract interpretation's result at the target location.

4 APPROXIMATE TYPE-DIRECTED SLICING

An advantage of static program reduction (over dynamic techniques in prior work) is that an approximate type-directed slicer can operate on *incomplete programs*. In a scenario like the one in section 2, this ability is particularly useful: a user reports a bug that is only reproducible on a large program to the maintainers of a typechecker. It is inconvenient and time-consuming for the maintainers to learn to build every bug-triggering program—many programs in the wild have unusual, quirky build processes.

To handle incomplete programs, we introduce *approximate* type-directed slicing, which does not require the full source code of P—any subprogram of P that contains the target location can be the input. While this approximation introduces unsoundness (i.e., violates theorem 3.4), it is often good enough in practice (which we show empirically in section 6). Our implementation (section 5) supports both exact and approximate type-directed slicing. The key theoretical difference between exact (as presented in section 3) and approximate type-directed slicing is that an incomplete program may contain *unsolved symbols*—that is, names (of classes, fields, etc.) that are not in the input. An approximate type-directed slicer has to create sensible "library" code to build a slice that includes these unsolved symbols. The key challenge in doing so is the various sources of *ambiguity* in a real programming language (e.g., Java) that can break the guarantees of theorem 3.4 for an approximate type-directed slicer. Section 4.1 describes the changes to the core exact slicing algorithm that are needed to support approximate slicing, and then section 4.2 discusses how we handle ambiguities in our approximate slicer in practice. The specific causes of such ambiguity may be different in a different programming language, but we expect that there will be significant similarities.

4.1 Context Inference for Unsolved Symbols

To support approximate type-directed slicing, we make one high-level modification to algorithm 1. The key idea is to check, for each component c that will be added to the slice, whether or not it is an unsolved symbol—that is, whether or not it is in the input program. If c is unsolved, then the approximate algorithm calls an *inferContext* function and adds its results to the slice (along with c). The *inferContext* function takes the given component c and produces a set of components that c requires. The implementation of *inferContext* is directly grounded in the type rule dependency map: From the dependency map, we can determine which components are needed, and for any unsolved ones, the map structure allows us to know—without executing the code—which components must be synthesized to maintain the expected behavior of the typechecker. For example, if c is an unsolved type name, the output of the *inferContext* function is a class declaring that type. More specifically, *inferContext* does the following for these kinds of unsolved symbols in our Java implementation (it might differ for other languages):

- c is an unsolved type expression (e.g., the type of a field declaration or of a parameter): *inferContext* creates a synthetic class for c.
- c is a read of an unsolved field: *inferContext* first checks if the field's type is unsolved. If so, it calls itself recursively on the type. Then, it adds the field to the synthetic class that it has created for the field's type.

- *c* is an unsolved method: *inferContext* first checks that each of the following types is not unsolved, and calls itself recursively on any types that are: the type containing the method, the return type (if applicable), thrown exception types, and the parameter types. Inside the synthetic class created for the type containing the method, *inferContext* will add the synthetic declaration of the method.

A naïve application of these rules can lead to incompatibilities between synthetic types and the rest of the slice, because the creation of a synthetic type does not depend on the usage context. For example, consider the field declaration `int x = MyClass.sizeCount;`. Without the source code for `MyClass`, the slicer cannot determine the type of the `sizeCount` field directly. To overcome this, the slicer initially assigns a synthetic type to `sizeCount`. But, the synthetic field clearly has the wrong type: the type must conform to `int`. To address this problem, *inferContext* runs the language's compiler on the created component(s) within the context of the slice. The slicer then uses the error messages to correct any synthetic types that are incompatible: for example, it can remove the synthetic type for `sizeCount` and replace it with `int` to fix the example above. Note that the cost of these compiler runs is small: proportional to the size of TypeSlice's output, not to the size of the original program; this technique can be thought of as "double-checking" the output of TypeSlice to catch ambiguous types that would prevent compilation.

4.2 Sources of Ambiguity in Java

This section explains a few of the specific sources of ambiguity in Java that we encountered while building our approximate type-directed slicer. This section's list is non-exhaustive (for space reasons), but other sources of ambiguity are broadly similar. We expect that an approximate slicer for another language might need to handle some or all of these issues, plus some other language-specific issues. So, this section's goal is to give the reader an understanding of the sorts of ambiguity that occur in approximate type-directed slicing.

```
import security.app.Data;
public class MainDatabase {
   public void main() {
    Data.clearCache(); } }
```

Fig. 4: `clearCache()`'s return type?

4.2.1 Type Ambiguities. Consider the example in fig. 4. Assume that `Data.java` is not available as source code: it is imported as a library. The return type of the `clearCache()` method is ambiguous to an approximate type-directed slicer. Instead, *inferContext* will create a synthetic return type for `clearCache()`. However, this synthetic return type is not the actual return type, which could lead to the resulting program not being compilable (e.g., if the result of `Data.clearCache()` were to be assigned into a local variable). While compiler errors can help resolve some type ambiguities, there are limitations. In cases where the type is inherently ambiguous, such as with `Data.clearCache()` (because its return type is not assigned anywhere), the compiler may not provide any error message, and our slicer will leave the synthetic type in the final output. In the event that this compromises the guarantees of theorem 3.4, the user would be required to switch to exact type-directed slicing by providing a classpath.

```
class Barley extends Grass {
    boolean hasLongLeaf() {
        return longLeaf; } }
```

Fig. 5: What class defines `longLeaf`?.

4.2.2 Lack of Information Regarding Superclasses. Another issue arises from the absence of information about superclass relationships. Consider three classes: `Barley`, `Grass`, and `Plant`, where `Barley` extends `Grass`, and `Grass` extends `Plant`. Suppose the input is (only) the code in fig. 5, without source files for `Grass` or `Plant` or a classpath that contains them. In this context, it is unclear whether the `longLeaf` field belongs to `Grass` or

Plant—in fact, the slicer has no way to know that Plant even exists, since it is not mentioned anywhere in the input. Our implementation assigns such unsolved fields to the nearest superclass (in this example, Grass).

```
Barley barley = new Barley() {
  @Override
  public int harvestTime() {
    return startDate + 60; } };
```

Fig. 6: Is startDate effectively final?

4.2.3 Final Variables in Anonymous Classes. In Java, an anonymous class can access local variables from its enclosing scope only if those variables are effectively final. For example, if the startDate variable in fig. 6 is not effectively final in the enclosing scope of barley, it is treated as a field of the Barley class. Keeping track of whether local variables are effectively final requires maintaining a complete symbol table, which is not possible if some of the enclosing scope is unsolved—for example, if this code is inside a class that extends another, unsolved class. We observe that it is unlikely that a field and an effectively final local would share the same name. Given this observation, for the purpose of symbol resolution, we assume that every local variable from the enclosing scope of an anonymous class is effectively final, resolving the ambiguity.

4.2.4 Relocating Inner Classes. The location of a class can also be ambiguous. For example, consider the human-written test case for a historical bug in our evaluation ("CF-577") in fig. 7. If Apple and Banana are in different source files, and the approximate type-directed slicer only has access to Banana, then it is ambiguous whether InnerApple is a class in the same package as Banana (and so usable with no import), or an inner class of the superclass Apple. Absent source code for Apple, an approximate type-directed slicer cannot know. In our implementation, we heuristically choose to always place classes used without an import (like InnerApple, above) in the same package as the corresponding source file. However, if such a class should actually be placed in the superclass Apple instead as an inner class, the behavior of a typechecker might change. In fact, this source of ambiguity does cause our implementation's approximate mode to fail to preserve the behavior of a typechecker for one historical bug (CF-577); see section 6.2.1.

```
class Banana extends Apple<int[]> {
    class InnerBanana extends InnerApple {
        // ...
    } }
```

Fig. 7: Ambiguity from "CF-577" benchmark.

4.2.5 Lambdas and Function Types. Lambda expressions are another source of ambiguity. Java's lambda support was added to the language late, and functions are not truly first-class—function types are not in the regular type hierarchy. When an approximate type-directed slicer encounters a lambda expression used as the right-hand side of some pseudo-assignment whose left-hand side is unsolved (e.g., the lambda expression is passed as an argument to an unsolved method), it needs choose a compatible type for the synthetic left-hand side (such as the parameter). Our implementation uses a straightforward heuristic: it creates a synthetic functional interface type for each combination of function arity (i.e., 0-parameter, 1-parameter, etc.) and presence of return type (i.e., void return or not) with fully-unconstrained generic parameters, and uses the matching functional interface for the actual lambda expression.

4.2.6 Annotation Targets. Java makes a distinction between "declaration" and "type" annotations. The former, introduced in Java 5, can be applied to method declarations, field declarations, etc. The latter, introduced in Java 8, can be applied anywhere that a Java type could appear. There are some contexts where it is not clear whether an annotation is a declaration annotation or a type annotation, so an approximate type-directed slicer has to choose. For example, consider an annotation that is only used to annotate fields. There are two possibilities: it is a declaration annotation applied to a field declaration, or it is a type annotation applied to the type of the field. These two kinds of annotations are not technically mutually-exclusive in Java, so our implementation treats unsolved annotations as *both*. However, this can change the behavior of a typechecker, if the typechecker expects a particular annotation to be one or the other.

4.2.7 Wildcard Imports. Each import statement in Java either names a specific symbol or imports all symbols in a particular package by appending `.*` after the package name in the import declaration. The latter is called a "wildcard import." Wildcard imports introduce ambiguity into the symbol resolution process in incomplete Java code. For example, without any wildcard imports, a type that is not imported must be in the same package as the class in which it is used. However, if that class has a wildcard import (and the imported package is not available as source), then the slicer cannot distinguish between the case where an unsolved type belongs to the imported package and the case where the type belongs to the same package as the class in which it is used. While the type rule dependency map that our implementation uses assumes that a typechecker can reason about the package that a class is defined in, in our experience it rarely matters in practice.

5 IMPLEMENTATION

We implemented an open-source type-directed slicer for Java, called TypeSlice [16], on top of the JavaParser [17] library for Java abstract syntax tree manipulation. TypeSlice uses JavaParser both to parse the input Java code and to connect related symbols (using JavaParser's Symbol Solver module). In exact mode, TypeSlice takes the classpath of the target program (a list of `.jar` archives containing Java bytecode) as an additional input. Then, it uses the Vineflower decompiler [18] to turn these bytecode files back into source files, so that it has access to the full program as source. Our data and experimental scripts are open-source and available [19].

6 EVALUATION

Our evaluation addresses these research questions:

RQ1 Can TypeSlice preserve the behavior of real Java typecheckers on a historical dataset of typechecker bugs?
RQ2 Is TypeSlice's running time acceptable?
RQ3 How close is the output of TypeSlice to what a human developer would do by hand?
RQ4 How does TypeSlice compare to Perses [7], Vulcan [20], and T-Rec [21], state-of-the-art dynamic program reduction tools?
RQ5 Is TypeSlice applicable to a broad range of programs?

6.1 Methodology

We performed a single experiment to answer RQs 1-3: we ran TypeSlice on historical bugs that we gathered from the issue trackers of three Java typecheckers:

- `javac`'s type system, via the OpenJDK bug repository;
- Uber's NullAway [2] type-based nullness analysis; and
- the Checker Framework [10], a collection of pluggable typecheckers.

Our goal is to determine if TypeSlice would have been useful in debugging reported issues in these tools. First, we reviewed each issue tracker for issues meeting the following criteria:

(1) the issue reports a problem with a typechecker (i.e., the problem occurs during the typechecking phase of compiling a Java program), such as a crash, false positive, or false negative.
(2) the issue's reporter provided a non-minimal program on which the issue can be reproduced.
(3) the issue occurred in a released version of the typechecker.

We collected 28 such issues: 2 bugs in the Java compiler, 9 from NullAway, and 17 from the Checker Framework. We did not find any other bugs in these issue trackers that meet our inclusion criteria. Most of these typechecker bugs (26 of the 28) were accompanied by a small test case. Since the state-of-the-practice is manual test case reduction, these small, independent test cases were presumably hand-minimized. They serve as ground truth for the output that TypeSlice should produce. By coincidence, exactly half (14 of 28) of the bugs we found are false positives (i.e., a typechecker issues an unexpected error); the other half are all crashes within the typechecker itself; we did not find any false negative bugs that meet our inclusion criteria (probably because these checkers are designed to minimize false negatives at the cost of more false positives). Our artifact contains links to each issue, the test case that we extracted from the fix, the version of the typechecker with the issue, and the reported issue's symptoms from the issue tracker.

We then built a pipeline to run TypeSlice on the program that first exhibited each of these issues (targeting the nearest enclosing method or field declaration to the place cited in the issue description), re-run the version of the relevant typechecker from the time that the issue was reported on TypeSlice's output, and compare the result to the issue's reported symptoms automatically. Our artifact includes all of the scripts necessary to run this pipeline, which we also use as a continuous integration build for TypeSlice itself.

To compare the results of the typecheckers on TypeSlice's output to the reported symptoms, we had to allow for some deviation: the output of the typecheckers (especially when they crash) directly includes information specific to the program being analyzed, such as the line number at which the problem occurred. To deal with this, we wrote regular expressions based on the issue logs to derive a "signature pattern" for the cause of each issue; for example, for issues that cause the typechecker to crash, the signature is the stack trace from the typechecker. We then used an automated script to compare the outcomes of relevant typechecker for both the original program and TypeSlice's output, by checking whether running the typechecker on TypeSlice's output matches the signature pattern for each issue.

6.2 RQ1: Behavior Preservation

Table 1 contains the main results of our experiment. The headline result is that in approximate mode, TypeSlice preserves the behavior of the target typechecker for 25 of the 28 targets (89%), even without access to the classpath of the target program. With a classpath, TypeSlice can preserve one more issue (26/28, 93%). This large percentage suggests that TypeSlice preserves typechecker behavior most of the time, even on complex programs. There are a few things to note about the table. For CF-577, CF-689, and CF-691, we only passed the file in which the problem occurs to TypeSlice, after relocating it from the JDK to a new, synthetic package. For NA-323 and CF-4614, the reporter provided a small but non-minimal test case; the actual target program was not open-source. A human maintainer minimized the provided test case further.

6.2.1 Why Approximate Slicing Fails to Preserve Behavior. In this section, we describe the specific causes of each case for which our approximate type-directed slicer fails to preserve the behavior of the target typechecker. For each of these cases, we also attempted to use the exact mode of our type-directed slicer; this succeeded in only one of the three cases.

NA-705. TypeSlice does not preserve NA-705 in approximate mode because NullAway expects that a particular annotation is (only) a type annotation (i.e., the ambiguity described in section 4.2.6). In exact mode, TypeSlice does preserve the typechecker's behavior.

CF-577. Approximate slicing fails to preserve CF-577 because an unsolved inner class must be placed inside a specific other unsolved class in order to trigger the bug (i.e., the ambiguity described

Table 1: TypeSlice's performance on historical Javac, NullAway ("NA"), and Checker Framework ("CF") bugs. "Kind" is the kind of bug: "FP" for false positive, or "Cr" for a crash. "LoC" is non-comment, non-blank lines of code after running google-java-format [22]. "T. LoC" and "H. LoC" are, respectively, the sizes of the TypeSlice-produced and human-written minimized tests. In the "Pr.?" column, ✓ means behavior is preserved in both modes; = in exact but not approximate mode; and X in neither mode. The clock symbol in the last column indicates reduction time in seconds. In the last row, the test cases sizes and run times are averages; others are totals.

Issue ID	Source	Kind	Target Program	LoC	Pr.?	T. LoC	H. LoC	🕒
JDK-8319461	javac	FP	mmm-property	7,970	✓	82	n/a	3
JDK-8288590	javac	FP	assertj	225,630	✓	36	36	14
NA-97	NA	FP	JDK	4,672,613	✓	111	13	49
NA-102	NA	FP	Caffeine	50,260	✓	67	9	5
NA-103	NA	FP	Caffeine	50,219	✓	286	n/a	11
NA-176	NA	Cr	Dropwizard	49,849	✓	205	14	5
NA-323	NA	FP	? (proprietary)	69	✓	27	22	2
NA-347	NA	FP	otr4j	25,095	✓	71	8	4
NA-389	NA	Cr	acs-aem-commons	70,748	✓	115	32	6
NA-705	NA	FP	Caffeine	77,487	=	80	6	4
NA-791	NA	Cr	Caffeine	79,934	✓	34	18	6
CF-577	CF	Cr	JDK	60	**X**	n/a	11	4
CF-689	CF	Cr	JDK	353	✓	320	12	25
CF-691	CF	Cr	JDK	3,045	✓	36	5	8
CF-1291	CF	FP	Daikon	119,563	✓	129	14	17
CF-3020	CF	Cr	guava	518,488	✓	55	10	12
CF-3021	CF	Cr	guava	518,479	✓	320	7	23
CF-3022	CF	FP	guava	518,479	✓	241	10	25
CF-3032	CF	FP	nomulus	169,783	**X**	n/a	51	5
CF-3619	CF	FP	calcite	351,992	✓	161	20	26
CF-3850	CF	Cr	calcite	364,957	✓	119	11	28
CF-4614	CF	FP	? (proprietary)	61	✓	14	21	2
CF-6019	CF	Cr	kafka-sensors	10,686	✓	42	5	4
CF-6030	CF	Cr	Cassandra	632,826	✓	97	19	35
CF-6060	CF	Cr	jOOQ	328,032	✓	54	6	25
CF-6077	CF	FP	Cassandra	633,142	✓	198	44	45
CF-6282	CF	Cr	Chronicle-Core	22,024	✓	29	17	4
CF-6388	CF	Cr	beam	827,527	✓	94	83	30
Totals and Averages:			28 issues	10,329,371	25	116	19	15

in section 4.2.4 and shown in fig. 7). In particular, continuing fig. 7's example, triggering the bug requires TypeSlice to place the `InnerBanana` class *inside* the `Banana` class. Instead, TypeSlice produces two top-level classes (which otherwise have the correct content), and so the bug is not reproduced. We tried running our exact slicer to resolve the issue, but because the bug is triggered by the JDK, providing the whole JDK (in this case, via `rt.jar`, as this historical bug requires Java 8) as input causes TypeSlice's dependency Vineflower to run out of memory.

CF-3032. TypeSlice cannot process this target in both modes due to a limitation of TypeSlice's dependency JavaParser related to method references [23]: JavaParser's symbol resolution does not fully model Java's search process for the correct method reference.

6.3 RQ2: Running Time

We also evaluated how quickly TypeSlice runs on the target programs (last column of table 1). On an `ubuntu-latest` GitHub Actions CI runner with 4 virtual CPUs and 16GB of RAM, the average reduction time was 15 seconds (minimum 2 seconds, maximum 49 seconds). Note that the subject programs are not trivial: most are in the hundreds of thousands of lines of non-comment, non-blank code, and TypeSlice scales very well from small examples to large programs. TypeSlice's speed is an advantage of *static* program reduction: the slowest part of dynamic program reduction is running the typechecker whose behavior we want to preserve, but TypeSlice does not need to run the typechecker at all. Section 6.5 directly compares TypeSlice's run time to dynamic program reduction tools on a subset of the benchmarks in table 1.

6.4 RQ3: Similarity to Human-written Tests

To evaluate whether TypeSlice's output program is similar to the test cases written by typechecker maintainers, we 1) evaluated the size of the TypeSlice-produced test cases, and 2) manually examined the test cases produce by TypeSlice and compared them to the human-written minimized tests. The average difference between the size of the human-written test cases and the TypeSlice-produced test cases is about 5x: the TypeSlice-produced test cases average 116 lines of non-comment, non-blank code, while the human-written test cases average just 19. TypeSlice theoretically should never produce smaller test cases than the humans, but in two instances it surprisingly did: JDK-8288590 and CF-4614. The reason for this in both cases is that the human-written "minimal" test case also includes one or more interesting variations that the human maintainer wanted to make sure that the typechecker could also handle—in other words, the human-written test is not really minimal at all.

Broadly speaking, the test cases produced by TypeSlice are larger since it is conservative about what to keep: it never removes a program element that could contribute to a typechecker bug, even if it does not contribute to the bug that is currently being targeted; in particular, TypeSlice-generated tests tend to include things like complete synthetic definitions of used but not relevant annotations (which a human would remove). The human maintainers are not so constrained. Qualitatively, though, the TypeSlice-generated test cases are small enough and easy enough to understand that we think they will be useful to typechecker maintainers. And, the TypeSlice-generated test cases share other, qualitative similarities to the human-written test cases: for example, both are often "unrunnable" (as discussed in section 3.3.1) in the sense that they lack any non-exceptional execution traces. That is, both TypeSlice and the human maintainers test typecheckers with program snippets intended only to trigger some specific behavior inside the typechecker.

6.5 RQ4: Comparison to Dynamic Program Reduction

Perses [7], Vulcan [20], and T-Rec [21] are state-of-the-art dynamic program reduction tools that uses input grammars to generalize across programming languages. Vulcan [20] improves on the core Perses algorithm's reduction performance (i.e., it produces smaller programs); T-Rec [21] further improves "canonicalization" performance, by reducing programs that differ syntactically but not semanticly to the same output more often. All three of these tools are part of the same open-source project [24], and are implemented as different command-line flags to the same tool. For simplicity, we will refer to that tool as "Perses" throughout this section. It takes as input the target program, a language grammar, and a "test script" that triggers an undesirable analysis behavior that should be preserved. An advantage of all three Perses variants is that they are fully generic over the target language and the analysis whose behavior is to be preserved, unlike TypeSlice, which is restricted to typecheckers for a specific language (in our case, Java). For these experiments, we used Perses version 2.0 (released January 10, 2025), which we downloaded from the project's

Table 2: Comparison of TypeSlice ("TS"), Perses ("P"), Vulcan ("V"), T-Rec ("TR"), and TypeSlice followed by Perses ("TS+P"). Note the time may differ from table 1, as this table's experiments ran on a different machine. Bugs below the horizontal line were hand-combined into a single file. The "*" marks a difference with table 1; see section 6.5.1 for the explanation. The "" means that Vulcan ran out of memory on our usual machine for this bug, so we ran it on an AWS `r5.xlarge` instance with 32GiB of RAM and 4 vCPUs; time numbers are not directly comparable across machines.**

	Time (s)					Output LoC					Preserved?				
Issue ID	TS	P	V	TR	TS+P	TS	P	V	TR	TS+P	TS	P	V	TR	TS+P
CF-4614	2	48	4089	487	343	14	14	15	15	13	✓	✓	✓	✓	✓
NA-323	2	242	3883**	424	205	27	8	6	8	8	✓	✓	✓	✓	✓
CF-577	n/a	1044	3912	1471	n/a	n/a	32	22	24	n/a	**X**	✓	✓	✓	n/a
CF-689	11	375	3573	988	92	132	90	41	9	17	✓	✓	✓	✓	✓
CF-691	4	2798	18242	4685	396	151*	167	164	167	134	✓	✓	✓	✓	✓

GitHub page [24]. The experiments in this section (only) were conducted on a 2024 Macbook Air with 16GB of Unified Memory.

Unfortunately, Perses has a technical design limitation that prevents it from minimizing most of the bugs in table 1: it can only minimize the parts of a program stored in a single input file. Because Java programs typically contain more than one Java file, the Perses implementation is not suitable for solving the problem we are interested in (program reduction for Java typechecker debugging) as an *off-the-shelf* tool. We suspect that this limitation of Perses is a historical artifact of its envisioned deployment scenario: reducing fuzzer-generated C programs to human-readable size in a CSmith [25]-like setup, for which multi-file reduction is not needed.

So, we instead did a case study that compares Perses and TypeSlice on the 5 bugs in table 1 whose input we could combine into a single Java file. We began with the two bugs in our dataset that already had single-file input programs ("CF-4614" and "NA-323"). We then tried to combine each program in the dataset by hand into a single file, starting from the program with the smallest size and proceeding towards the largest; we gave up once it became clear that no larger programs could be so combined. We succeeded at combining just three programs into a single file; anecdotally, namespace collisions become implausibly difficult to resolve for the larger benchmarks, as different files import classes with the same names from different packages. For each of these five (two single-file, three combined into a single file), we wrote a Perses test script that triggers the bug and executed Perses, Vulcan, and T-Rec on each input program using the Java grammar that Perses ships with; the results appear in table 2.

6.5.1 Results. At a high level, the results confirm our intuition: when successful, all three dynamic reducers are dramatically slower than TypeSlice because they run the typechecker whose behavior is to be preserved many times—e.g., Perses runs the analysis 237 times to minimize CF-4614. Of the three, Vulcan is consistently the slowest, but it also sometimes achieves the best reduction performance. Like TypeSlice, all three offer a soundness guarantee when the whole input program is provided, but unlike TypeSlice they cannot be used to minimize incomplete programs. However, they can be more precise than TypeSlice: the minimized test case that Perses finds for NA-323, for example, is much smaller than the test case that TypeSlice finds, because TypeSlice's test includes getters and setters that are used by the constructor in which the bug occurs, but actually are not required to trigger it; Perses deletes them entirely. CF-689 exhibits a similar phenomenon.

To our surprise, TypeSlice and the dynamic reducers produce similarly-sized outputs for CF-4614 and for CF-691. In CF-4614, TypeSlice removes a program element (a field) that Perses keeps (Perses removes the package declaration and a few other small elements, like `public` and `final` modifiers, so the resulting files are the same size). On CF-691, Perses and TypeSlice produce relatively-similar outputs, though TypeSlice's is a bit smaller: TypeSlice removes the body of a method and a field that Perses keeps, though neither is needed to reproduce the typechecker crash. The input file for CF-691 in this experiment is subtly different than the input file used for the experiment in table 1, to accommodate Perses' inability to process incomplete programs: other parts of the `java.util` package (which contains the input file) are imported explicitly in this experiment, whereas in table 1 they are treated as reducible code. These imports allow Perses to do reduction at all (they are necessary for compilation), but cause TypeSlice to retain more of the file (because TypeSlice assumes it cannot, e.g., remove an abstract method from an irreducible imported class).

All three of Perses, Vulcan, and T-Rec can preserve CF-577's behavior, unlike TypeSlice: TypeSlice incorrectly moves an inner class (discussed further in section 6.2.1), but all three Perses variants avoid the ambiguity that trips up TypeSlice due to their dynamic nature.

These results shows that in the best case scenario for TypeSlice, its outputs can be more-minimal than the output of Perses (as we see in CF-4614 and CF-691); however, we expect the average case looks more like NA-323, where the TypeSlice output is about thrice as large as the Perses output, or like CF-689, where the best dynamic reducer (T-Rec) finds a program that triggers the bug of only 9 lines, compared to TypeSlice's 132. In other words, in the typical case dynamic reduction produces test cases that are closer in size to the human-written minimized tests than TypeSlice's static reduction technique does, but at the cost of longer run time. Moreover, dynamic reduction's stronger guarantees about preservation make it a good choice when an absolute guarantee of preservation is required and ambiguity makes static reduction fail, as we see for CF-577. However, static reduction scales better: it can handle huge programs that are impossible to process with a dynamic reducer. In this sense, the two approaches are complementary.

6.5.2 Combining TypeSlice and Dynamic Reduction. The direct comparison between TypeSlice and Perses-based dynamic reduction tools in the previous section showed that TypeSlice is much faster, but the output of dynamic reduction once it finishes can be smaller. However, the two techniques need not be exclusive: if we first run TypeSlice and then a dynamic reducer *on TypeSlice's output*, we retain some of TypeSlice's speed but benefit from more precise dynamic reduction. We did this experiment for the bugs in table 2 ("TS+P" columns) by combining TypeSlice and (basic) Perses. The results are encouraging: both tools together are faster than Perses alone, and the minimized programs are at least as small as Perses' output. For example, on CF-689, running TypeSlice and then Perses produces a test case nearly as small as the T-Rec minimization, but takes only about 1/10th of the time as running T-Rec. For larger programs, the combination should be commensurately faster, since TypeSlice scales better than Perses with program size.

6.6 RQ5: Broad Applicability

The methodology in section 6.1 directly measures TypeSlice's usefulness for the task that we designed it for—minimizing historical typechecker bugs—but is limited in scope because of our strict inclusion criteria. To give a better sense of the broader applicability of the tool, we designed and carried out a larger experiment with TypeSlice that checks preservation of a simpler property that is easy to check automatically: compilability. Our goal with this experiment is to give the reader a sense of the scalability and broad applicability of the approach.

6.6.1 Methodology. We collected 32 Java projects from GitHub, using two approaches: we included 16 projects from the Defects4J [26] benchmark suite[1], and we searched GitHub using Source-Graph [27] public code search for Java repositories with at least 15 stars, and then sampled another

[1]All of the projects except closure-compiler, which caused issues with our experimental setup.

16 repositories from the result by convenience. We excluded repositories during this step that were collections of small snippets (for example, a collection of algorithms from a textbook) rather than a single, cohesive project. We also excluded all projects that did not successfully build using a single command on the server we used for these experiments, which was running Arch Linux with a Java 11 JDK. These 32 repositories together contain 1,892,104 lines of non-comment, non-blank Java code. For each of these projects, we automatically extracted a list of all valid method signatures (there were a total of 49,577 across all of these projects), and triggered a run of TypeSlice targeting each such signature. We attempted to compile the resulting TypeSlice-minimized program.

6.6.2 Results. Across all projects, TypeSlice produces compilable output for 73.0% of method signatures (36,175 of 49,577). Of those that could not be successfully compiled, TypeSlice crashes for about half (7,214 of 13,402, 53.8%) and produces non-compilable output for the other half (6,188 of 13,402, 46.2%). Random sampling of failures suggest that effectively all of them are caused by bugs in the implementation of the tool, rather than fundamental limitations of our approach. Moreover, failures (especially crashes) are concentrated in just a few projects: JFreeChart alone triggers 3,372 of the 7,214 crashes we observed (46.7%), and just four of the 32 projects are responsible for 3,334 of the 6,188 compilation failures (53.9%).

7 LIMITATIONS AND THREATS TO VALIDITY

7.1 Threats to Validity

A threat to the external validity (generalizability) of our experiments is that we only tested TypeSlice on three closely-related type systems. Further, most (17 of 28) of the bugs we tested on came from the Checker Framework's issue tracker, so our approach may only be effective for pluggable typecheckers like the Checker Framework. It is also possible that some of our proxies may not be good—especially the *size* of the TypeSlice-generated test cases for usefulness to analysis developers.

7.2 Limitations

The most serious limitation of type-directed slicing is that it is only useful for preserving the behavior of typecheckers, not of other kinds of program analyses. Another serious limitation of type-directed slicing is that it is not useful for debugging bugs in the modularity of a typechecker—if the type rule dependency map is not respected, the type-directed slicer can offer no guarantees even in exact mode. Another, related limitation is the need to define the type rule dependency map in the first place: building a type-directed slicer requires a formalism like type rules. In practice, this means that type-directed slicing is only useful in conjunction with typecheckers, and not with other kinds of program analyses (though we hope to extend the underlying ideas to other kinds of analyses with well-defined formalisms in future work; see section 9.1).

Like any practical tool, TypeSlice can have bugs which compromise the theoretical guarantees of type-directed slicing. We have mitigated this problem by testing TypeSlice: its test suite contains 211 small, synthetic test programs totaling 2,884 lines of non-comment, non-blank code. We also run integration tests on the 28 bugs in table 1 on each change to TypeSlice as a continuous integration build. Despite our efforts, bugs like the one that prevents TypeSlice from processing issue CF-3032 and those that caused the compilation failures in section 6.6 sometimes do occur.

Another limitation is that while TypeSlice can handle *incomplete* programs, those programs must still be *well-formed*: that is, they must be valid in the language's grammar and therefore parseable to an abstract syntax tree, and they must be subprograms of some program that typechecks. TypeSlice will fail if either of these assumptions is violated: if the program cannot be parsed, TypeSlice will fail immediately; if the program could not typecheck, then TypeSlice's output probably will not typecheck, either.

8 RELATED WORK

Type-directed slicing is related to work in program reduction, in traditional slicing, and in incremental compilation.

8.1 Program Reduction

Other researchers have investigated the problem of program reduction in the context of analysis debugging and built practical tools like C-Reduce and Perses [6, 7, 20, 21, 28–31]. These works are united by their dynamic approach to the problem: their analysis strategy is based around a delta-debugging-like divide-and-conquer core [8]; most recent research contributions are refinements to this basic algorithm that improve performance, make the approach more language-agnostic, and/or exploit program structure to avoid testing syntactically-invalid programs. Recent work has also proposed using machine learning models to guide these dynamic tools [32, 33]. We compare directly to Perses [7] and its follow-up works Vulcan [20] and T-Rec [21] in section 6.5. The most closely-related work to ours from this line of work is JReduce [29] which, like TypeSlice, uses dependency relationships in its input program to aid in reduction. However, JReduce computes dependency closures at the file or class level rather than being type-directed, like TypeSlice. We could not do a direct comparison with JReduce as it operates on bytecode rather than source code.

A type-directed slicer is a static analysis that solves the same problem, and offers different trade-offs compared to its dynamic cousins described in the previous paragraph. For example, the user of a type-directed slicer need not run the analysis being debugged, which makes type-directed slicing much faster in practice than the dynamic approach. If running the typechecker is expensive (as may be the case if, for example, it is enforcing a dependent type system), this run time becomes unwieldy in practice quickly. The only other extant static analysis for this problem, to our knowledge, is Trimmer [34], which removes program paths that are not relevant to some target assertion that a program analyzer is trying to prove or disprove. Like TypeSlice, Trimmer is inspired by slicing: it uses traditional slicing techniques to slice away paths whose effects cannot change the truthiness of the assertion; TypeSlice's approach of slicing on the modularity boundaries of the type rules is technically quite different. Like our approximate slicer, PAClab's *Transformer* component [35] has the ability to "automatically make Java classes compilable outside of their host project." However, it does not guarantee preservation of compile-time semantics, unlike our exact approach.

8.2 Slicing

Since it was first introduced by Weiser [9], researchers have investigated a wide variety of techniques related to program slicing: too many to discuss here. We refer the interested reader to surveys on the topic [36–39]. The key difference between our type-directed slicing approach and "traditional" slicing techniques is our focus on slicing to preserve *compile-time* rather than *run-time* behavior; as far as we are aware, prior work in slicing has not attempted to preserve the compiler's (or some other typechecker's) behavior, as we do. The Symbiotic symbolic execution tool [40] uses slicing in a similar way to remove "unrelated" parts of the program before running a static analysis. It uses a traditional slicer, though, because the goal is to then symbolically execute the program.

8.3 Incremental Compilation

Modern IDEs require a form of incremental compilation based on a specific target method—in particular, the method in which the user is currently making edits—to facilitate auto-completion, error-reporting, etc. Significant research effort has gone into developing incremental parsers for this scenario [41–43]. Modern IDEs use *language servers* to ease the implementation of such tools [44]. However, the compilers that IDEs rely on (e.g., Roslyn for C# [45] or the TypeScript compiler [46]) are fundamentally different than a type-directed slicer: while they start at a particular location and build up their internal representation of the program outwards, like TypeSlice, they are 1) still interested in preserving the run-time behavior of code that is not directly used by their target method, and 2) for speed, they rely on previous, cached compilation passes. TypeSlice does neither, because it is targeted at exactly the problem of preserving the behavior of a semantic analysis (in particular, a typechecker). The Roslyn developers have said that they avoided attempting incremental semantic analysis, because they were concerned that the implementation would be too complex [47]. A type-directed slicing approach could enable them to do so in the future.

9 FUTURE WORK

9.1 Generalizability to Analyses Other Than Type Systems

Our claims in this paper are limited to type systems: we have not attempted to show that type-directed slicing is useful for any other kind of analysis. In this section, though, we speculate on whether we could build something like a type-directed slicer for other kinds of analyses in the future, based on the properties of type systems that type-directed slicing relies on. In particular, type systems are *intra-procedural* program analyses: that is, they examine each program component in isolation and use summaries (type annotations) to communicate information across procedure boundaries. This property is a key requirement for type-directed slicing. The following (informal) definition of modularity captures this intuition:

Definition 9.1. A *modular* program analysis directly reasons about only the code in one target component C of the program at a time. A modular analysis can request and use *summaries* of other components that are used by C, but it does not and cannot directly reason about the code in those other components (instead, it must trust the summaries).

Type-directed slicing might plausibly generalize to any analysis that 1) meets this definition of modularity, and 2) for which we can construct a well-defined dependency map from some formal description of the analysis. The reason that we focus on typecheckers in this work is that constructing such a map for them is straightforward: the derivation from the type rules is obvious.

Some other analyses besides type systems meet this definition of modularity, too, and therefore might be able to use a technique like type-directed slicing in the future, if we can construct a dependency map for them. The key barrier is that few other analyses have such well-defined formalisms from which it is straightforward to extract a dependency map. For example, heuristic bug-finding tools like FindBugs [48] or the dataflow analyses built into modern IDEs (e.g., [49]) are definitely modular, but typically lack a formal description. And, some sound verification tools that prove user-written specifications, such as OpenJML [50] or KeY [51], are modular because they rely on user-written specifications to communicate across method boundaries; because these tools are based on a core logic (e.g., OpenJML is in the Larch family of specification languages [52]), it might be feasible to derive a dependency map from their formalisms. We leave doing so to future work.

However, it would be implausible to extend our type-directed slicing approach to some analyses, because those analyses are fundamentally non-modular. For example, Facebook's Infer tool [5] is non-modular because its analysis is inter-procedural. Other examples include analyses backed by reduction to graph reachability (i.e., IFDS/IDE [53, 54]), such as FlowDroid [55] or CogniCrypt [56].

9.2 Other Applications

Static program reduction via type-directed slicing might be useful for other software engineering tasks that involve typecheckers besides debugging, which we plan to explore in future work.

For example, when generating code using a large language model (LLM), it may be desirable to use one or more typecheckers to discard incorrect code generated by the LLM. One issue with this general approach is that the context window for LLMs may not be large enough to include the entire program; recent work [57] has proposed using (dynamic) program reduction to help. A type-directed slicer could provide the LLM with the slice of the codebase related to any particular warning from the typechecker, making it easier and quicker for the LLM to fix the warning.

Another possible future application of type-directed slicing is in making it practical to reduce false positives from typecheckers by trying to typecheck semantically-equivalent refactorings. For example, consider the two semantically-equivalent programs in fig. 8. Figure 8b uses a simple nullability check before dereferencing `x`, while fig. 8a stores the nullability of `x` in a boolean and then checks that boolean instead. A nullability typechecker like the Checker Framework's Nullness Checker [10, 58] will warn about the program in fig. 8a, but not the program in fig. 8b, because it does not reason about whether specific booleans are connected to nullability. We can use a type-directed slicer to automate the discovery that a warning like this one is a false positive, using the following steps. First, use TypeSlice to create a small, independently-compilable program that exhibits the warning. Then, apply each refactoring from a library of semantically-equivalent program transformations to this reduced program and re-run the typechecker; if any one refactoring leads the typechecker to no longer issue the warning, then the warning must have been a false positive (assuming the typechecker is sound). We could use a system like this one to avoid showing warnings that are definitely false positives to the developer, making typecheckers more useful.

```
boolean xIsNull = x == null;
if (!xIsNull) { x.m(); }
```

(a) Extant nullability typecheckers would warn here...

```
if (x != null) { x.m(); }
```

(b) ...but not here.

Fig. 8: Two semantically-equivalent programs that dereference `x` if it is non-null.

10 CONCLUSION

We have introduced type-directed slicing, a practical static program reduction technique that can preserve the behavior of a typechecker. Compared to the state-of-the-art dynamic tools based on delta-debugging-like algorithms, our static program reduction technique has predictable performance that is unrelated to the cost of running the typechecker whose results are to be preserved. Our approximate slicing technique builds on this advantage by adding the ability to process incomplete programs, which makes it easy to apply to the bug reports that users of static analysis tools typically provide. Although approximate slicing does not guarantee that typechecker behavior is preserved, our experiments show that on a significant collection of real bugs reported in Java typecheckers, TypeSlice's approximate mode usually preserves the relevant behavior in practice.

DATA AVAILABILITY

All data and code used in this paper is open-source and publicly available. The implementation of the TypeSlice tool is available at https://anonymous.4open.science/r/typeslice-issta25. The scripts and data used to produce the experiments in section 6 are available at https://anonymous.4open.science/r/typeslice-eval-scripts-issta25.

REFERENCES

[1] D. Delmas and J. Souyris, "Astrée: From research to industry," in *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings 14*. Springer, 2007, pp. 437–451.

[2] S. Banerjee, L. Clapp, and M. Sridharan, "NullAway: Practical type-based null safety for Java," in *ESEC/FSE 2019: The ACM 27th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Tallinn, Estonia, Aug. 2019, pp. 740–750.

[3] M. Kellogg, M. Schäf, S. Tasiran, and M. D. Ernst, "Continuous compliance," in *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*, Melbourne, Australia, Sep. 2020, pp. 511–523.
[4] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 598–608.
[5] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods Symp.* Springer, 2015, pp. 3–11.
[6] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 335–346.
[7] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 361–371.
[8] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE TSE*, vol. 28, no. 3, pp. 183–200, Feb. 2002.
[9] M. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. dissertation, University of Michigan, Ann Arbor, 1979.
[10] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 2008, pp. 201–212.
[11] V. Sitnikov, "Regression in 3.7.1: InternalUtils.symbol: tree is null," https://github.com/typetools/checker-framework/issues/3850, 2020, accessed 11 April 2024.
[12] The Calcite Developers, "Apache calcite," https://github.com/apache/calcite, 2024, accessed 11 April 2024.
[13] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: a minimal core calculus for Java and GJ," *ACM TOPLAS*, vol. 23, no. 3, pp. 396–450, May 2001.
[14] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, Jan. 1977, pp. 238–252.
[15] P. Cousot, "Types as abstract interpretations," in *POPL '97: Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997, pp. 316–331.
[16] Anonymous Authors, "TypeSlice: A type-directed slicer for Java," https://anonymous.4open.science/r/typeslice-issta25, 2024.
[17] JavaParser Contributors, "Javaparser: Java 1-17 parser and abstract syntax tree for java with advanced analysis functionalities," 2023. [Online]. Available: https://github.com/javaparser/javaparser
[18] The Vineflower Developers, "Vineflower," https://github.com/Vineflower/vineflower, 2024, accessed 11 April 2024.
[19] Anonymous Authors, "TypeSlice evaluation scripts," https://anonymous.4open.science/r/typeslice-eval-scripts-issta25, 2024.
[20] Z. Xu, Y. Tian, M. Zhang, G. Zhao, Y. Jiang, and C. Sun, "Pushing the limit of 1-minimality of language-agnostic program reduction," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 636–664, 2023.
[21] Z. Xu, Y. Tian, M. Zhang, J. Zhang, P. Liu, Y. Jiang, and C. Sun, "T-rec: Fine-grained language-agnostic program reduction guided by lexical syntax," *ACM Transactions on Software Engineering and Methodology*, 2024.
[22] Google, "google-java-format, v1.25.2," https://github.com/google/google-java-format/releases/tag/v1.25.2, 2024, accessed 25 February 2025.
[23] https://github.com/fabgo, "UnsolvedSymbolException resolving MethocCallExpr using MethodReferenceExpr," https://github.com/javaparser/javaparser/issues/4188, 2023, issue is open as of October 31, 2024.
[24] Perses Contributors, "Perses: Syntax-directed program reduction," 2023. [Online]. Available: https://github.com/uw-pluverse/perses/tree/master
[25] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.
[26] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, USA, July 2014, pp. 437–440, tool demo.
[27] SourceGraph, "Public code search," https://sourcegraph.com/search, 2025, accessed 26 February 2025.
[28] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 142–151.
[29] C. G. Kalhauge and J. Palsberg, "Binary reduction of dependency graphs," in *PLDI 2019: Proceedings of the ACM SIGPLAN 2016 Conference on Programming Language Design and Implementation*, Phoenix, AZ, USA, June 2019, pp. 556–566.
[30] Y. Tian, X. Zhang, Y. Dong, Z. Xu, M. Zhang, Y. Jiang, S.-C. Cheung, and C. Sun, "On the caching schemes to speed up program reduction," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–30, 2023.

[31] M. Zhang, Z. Xu, Y. Tian, Y. Jiang, and C. Sun, “PPR: Pairwise Program Reduction,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 338–349.
[32] G. Gharachorlu and N. Sumner, “Type batched program reduction,” in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 398–410.
[33] M. Zhang, Y. Tian, Z. Xu, Y. Dong, S. H. Tan, and C. Sun, “Lpr: Large language models-aided program reduction,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 261–273.
[34] K. Ferles, V. Wüstholz, M. Christakis, and I. Dillig, “Failure-directed program trimming,” in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 174–185.
[35] R. Brunner, R. Dyer, M. Paquin, and E. Sherman, “PAClab: a program analysis collaboratory,” in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1616–1620.
[36] F. Tip, “A survey of program slicing techniques,” *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.
[37] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, “A brief survey of program slicing,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.
[38] M. Harman and R. Hierons, “An overview of program slicing,” *software focus*, vol. 2, no. 3, pp. 85–92, 2001.
[39] J. Silva, “A vocabulary of program slicing-based techniques,” *ACM computing surveys (CSUR)*, vol. 44, no. 3, pp. 1–41, 2012.
[40] J. Slabý, J. Strejček, and M. Trtík, “Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution,” in *Formal Methods for Industrial Critical Systems: 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings 17*. Springer, 2012, pp. 207–221.
[41] T. A. Wagner, *Practical algorithms for incremental software development environments*. University of California, Berkeley, 1997.
[42] T. A. Wagner and S. L. Graham, “Incremental analysis of real programming languages,” *ACM SIGPLAN Notices*, vol. 32, no. 5, pp. 31–43, 1997.
[43] E. R. Van Wyk and A. C. Schwerdfeger, “Context-aware scanning for parsing extensible languages,” in *Proceedings of the 6th international conference on Generative programming and component engineering*, 2007, pp. 63–72.
[44] N. Gunasinghe and N. Marcus, *Language Server Protocol and Implementation*. Springer, 2021.
[45] The Roslyn Developers, “Roslyn: The .NET Compiler Platform,” https://github.com/dotnet/roslyn, 2023.
[46] The TypeScript Developers, “TypeScript,” https://github.com/microsoft/TypeScript/, 2023.
[47] E. Lippert, “Answer to "what are some techniques for faster, fine-grained incremental compilation and static analysis?",” https://langdev.stackexchange.com/a/2880, 2023.
[48] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, “Using static analysis to find bugs,” *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, Sep. 2008.
[49] Eclipse developers, “Avoiding resource leaks,” https://help.eclipse.org/2020-12/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-avoiding_resource_leaks.htm&cp%3D1_3_9_3, 2020, accessed 3 February 2021.
[50] The OpenJML Developers, “OpenJML,” https://www.openjml.org/, 2022.
[51] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich, “The KeY platform for verification and analysis of Java programs,” in *VSTTE 2014: 6th Working Conference on Verified Software: Theories, Tools, Experiments*, Vienna, Austria, July 2014, pp. 55–71.
[52] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing, *Larch: Languages and Tools for Formal Specification*, ser. Texts and Monographs in Computer Science. New York, NY: Springer-Verlag, 1993.
[53] T. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in *POPL ’95: Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, Jan. 1995, pp. 49–61.
[54] M. Sagiv, T. Reps, and S. Horwitz, “Precise interprocedural dataflow analysis with applications to constant propagation,” *Theoretical Computer Science*, vol. 167, no. 1-2, pp. 131–170, 1996.
[55] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps,” in *PLDI 2014: Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation*, Edinburgh, UK, June 2014, pp. 259–269.
[56] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, “CrySL: An extensible approach to validating the correct usage of cryptographic APIs,” in *ECOOP 2018 — Object-Oriented Programming, 32nd European Conference*, Amsterdam, Netherlands, July 2018, pp. 10:1–10:27.

[57] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, "DeepCode AI fix: Fixing security vulnerabilities with large language models," *arXiv preprint arXiv:2402.13291*, 2024.

[58] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller, "Building and using pluggable type-checkers," in *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, Waikiki, Hawaii, USA, May 2011, pp. 681–690.