Analyzing Test Completeness for Dynamic Languages

Christoffer Quist Adamsen
Aarhus University, Denmark
quist@cs.au.dk

Gianluca Mezzetti
Aarhus University, Denmark
mezzetti@cs.au.dk

Anders Møller
Aarhus University, Denmark
amoeller@cs.au.dk

ABSTRACT

In dynamically typed programming languages, type errors can occur at runtime. Executing the test suites that often accompany programs may provide some confidence about absence of such errors, but generally without any guarantee. We present a program analysis that can check whether a test suite has sufficient coverage to prove a given type-related property, which is particularly challenging for program code with overloading and value dependent types. The analysis achieves a synergy between scalable static analysis and dynamic analysis that goes beyond what can be accomplished by the static analysis alone. Additionally, the analysis provides a new coverage adequacy metric for the completeness of a test suite regarding a family of type-related properties.

Based on an implementation for Dart, we demonstrate how such a hybrid static/dynamic program analysis can be used for measuring the quality of a test suite with respect to showing absence of type errors and inferring sound call graph information, specifically for program code that is difficult to handle by traditional static analysis techniques.

CCS Concepts

•Theory of computation → Program analysis;

Keywords

Program testing; static analysis; type checking; Dart

1. INTRODUCTION

A well-known quote from Dijkstra is: "Program testing can be used to show the presence of bugs, but never to show their absence" [13]. This is of course true in general, but, as also observed decades ago [19], there are situations where test suites are *complete* in the sense that they allow verification of correctness properties that hold in any execution. As a trivial example, for a straight-line Java program that takes a string as input, a single test execution suffices to detect any null pointer bug that may exist. More generally, test suites may be complete with respect to local properties at specific program points. In this paper, we explore how such a notion of test completeness can be exploited for soundly type checking challenging programming patterns in dynamically typed languages.

One of the advantages of programming languages with dynamic or optional typing, such as, Dart [14], TypeScript [33], Typed Racket [43], and Reticulated Python [45], is flexibility: they allow dynamic programming patterns, such as, value-based overloading [44, 1] and other forms of value dependent types, that do not fit into traditional static type systems. The cost of this flexibility is that type-related errors are not detected until runtime and that type information is not available statically for guiding auto-completion and navigation in IDEs or optimizers in compilers. Our goal is to provide program analysis techniques that can automatically infer sound and precise type information for such code. More specifically, we focus on Dart and aim to ensure absence of runtime type errors and to infer precise call graph information. Two kinds of runtime type errors are particularly interesting in Dart programs: subtype-violation errors at implicit downcasts and message-not-understood errors at field and method lookup operations. Programmers can benefit from knowing to what extent their tests guarantee absence of such runtime errors. Likewise, call graphs are useful for enabling dead code elimination and other optimizations, as well as for various kinds of IDE support [16].

Example Figure 1 shows some Dart code from a sample application shipped with the *box2d* library[1] that uses the *vector_math* library,[2] which illustrates a style of programming that is common with dynamically typed languages and that is challenging for static type analysis. The `cross` function is overloaded in the sense that its behavior and return type depend on the runtime types of its parameters: the branch in line 4 returns type `vec3` or the runtime type of the parameter `out`, line 7 returns type `double`, and line 16 returns type `vec2` or the type of `out`. Moreover, if the types of the parameters at a call do not match any of the cases, then a failure occurs in line 20 if assertions are enabled and otherwise `null` is returned, which will likely trigger a runtime error later in the program. The `cross` function is called, for example, in lines 32 and 33. In Dart's checked mode execution, the assignments to `dp2perp` and `dp1perp` will fail if the values being returned are not of type `vec3`. In production mode execution, the assignments will succeed, but the applications of `*` in line 34

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...
http://dx.doi.org/10.1145/2931037.2931059

[1]`https://github.com/financeCoding/dartbox2d`
[2]`https://github.com/google/vector_math.dart`

```
 1 dynamic cross(dynamic x, dynamic y,
 2               [dynamic out=null]) {
 3   if (x is vec3 && y is vec3) {
 4     return x.cross(y, out);
 5   } else if (x is vec2 && y is vec2) {
 6     assert(out == null);
 7     return x.cross(y);
 8   } else if (x is num && y is vec2) {
 9     x = x.toDouble();
10     if (out == null) {
11       out = new vec2.zero();
12     }
13     var tempy = x * y.x;
14     out.x = -x * y.y;
15     out.y = tempy;
16     return out;
17   } else if (x is vec2 && y is num) {
18     ...
19   } else {
20     assert(false);
21   }
22   return null;
23 }
24 class vec2 {
25   ...
26   vec2 operator+(vec2 other) =>
27     new vec2.raw(x + other.x, y + other.y);
28   ...
29 }
30 ...
31 // solve the linear system
32 vec3 dp2perp = cross(dp2, normal);
33 vec3 dp1perp = cross(normal, dp1);
34 vec3 tangent = dp2perp * duv1.x + dp1perp * duv2.x;
35 ...
```

Figure 1: Code from the *vector_math* library (lines 1–29) and the *box2d* library (lines 31–34).

will fail if `dp2perp` or `dp1perp` is of type `double` and the other argument is of type `vec2` or `vec3`, and `+` will fail if the left-hand argument and the right-hand argument do not have the same type. (Lines 26–27 show the implementation of `+` for the case where the left-hand argument is of type `vec2`.) How can the programmer be certain that none of these potential type-related errors can occur in the application? Running a few tests may give some confidence, but it is difficult to know when enough testing has been done.

Traditional static analysis is not suitable for reasoning about such code since very high precision (e.g. context sensitivity and path sensitivity) would be needed, which is difficult to achieve together with sufficient scalability. For the example described above, a context-insensitive analysis may be able to show that `cross` can return values of type `vec3`, `vec2`, or `double` (assuming that `out` is always also of one of those types), but that information is not precise enough to rule out the various type-related errors. Two-phase typing [44] has been proposed to address similar challenges but is not yet practical for realistic programs. Conversely, traditional dynamic analysis is also not suitable as it does not give any soundness guarantees. One notable exception, which has inspired our work, is the constraint-based dynamic type inference technique by An et al. [2]. It can infer sound type information for Ruby programs using dynamic analysis, however, it requires full path coverage of every function in the program being analyzed, which is rarely realistic.

This work We propose a hybrid of lightweight static analysis and dynamic execution of test suites. Our key insight is that such a combination of static and dynamic techniques can determine when test suites have sufficient coverage to guarantee type-related correctness properties as in the example.

The static analysis has two parts: a dependence analysis and a type analysis (technically, a points-to analysis). It is context- and path-insensitive and thereby scales to large programs, and it is relatively easy to implement; notably it requires simpler modeling of native functions than what would be required by a fully static analysis approach.

In summary, the contributions of this paper are:

- We define a notion of test completeness as a sufficient condition for a test suite to have enough coverage to guarantee a given type-related correctness property. Using a lightweight static analysis to approximate test completeness, we then demonstrate how a hybrid static/dynamic analysis can produce sound type information for challenging dynamic programming patterns that resist traditional static analysis.
- Based on an implementation, GOODENOUGH,[3] for the Dart programming language, we evaluate the ability to ensure absence of runtime type errors and to produce precise call graphs, compared to a more traditional technique that uses the static analysis information only. Across a variety of Dart programs, we find numerous examples where precision is improved. Specifically, the analysis is able to guarantee for 81 % of the expressions that all types that can possibly appear at runtime are in fact observed by execution of the test suite. The experiments additionally show that the limiting factor of the precision in some cases is the test coverage and in other cases the precision of the dependence analysis, which suggests opportunities for future improvements of the technique.

2. THE DART LANGUAGE

In this paper we use Dart as subject for presenting and evaluating our technique. The Dart programming language was introduced by Google in 2013 and has later been standardized by Ecma [14]. The language is now widely used by Google and elsewhere, primarily as an alternative to JavaScript but also for server-side high performance computation and, more recently, embedded devices. Dart supports both object oriented and functional programming using a Java-like syntax.

From our perspective, the most interesting aspect of the language is its type system. Type annotations are optional (the default is called `dynamic`). The static type checker can warn about likely type errors in the parts of the program where type annotations have been provided, while ignoring the parts without type annotations. Warnings produced by the type checker do not preclude executing the program. Moreover, even for fully annotated programs, the type checker is deliberately unsound. These design choices give Dart the flexibility of dynamically typed programming languages, but they also mean that type errors can occur at runtime.

Dart programs run in either *production mode* where type annotations are entirely ignored, or *checked mode* where implicit type casts are performed at assignments to non-`dynamic` variables or fields, checking that the runtime type is a subtype of the annotated type. Two kinds of runtime type errors can occur: a *subtype-violation* error occurs if a

[3]Named in honor of J. B. Goodenough [19].

type cast fails, and a *message-not-understood* error occurs if attempting to access a field or method that does not exist. We here use the terminology from Ernst et al. [15] who recently presented a formalization of a core of Dart with a focus on its type system and the causes of unsoundness.

The example code in Figure 1 shows how programmers use optional types in practice. Some variables and methods have type annotations (e.g. `dp2persp` in line 32 and `operator+` in line 26), whereas in other parts (e.g. `cross`) the programmer chose to use `dynamic` (that annotation could in fact be omitted without changing the meaning of the code).

3. OVERVIEW

Given a Dart program with a test suite, we wish to know for any expression e whether the test suite has sufficient coverage to explore all possible types e may evaluate to. In our setting, a test suite is simply a finite set of program inputs. To simplify the discussion we assume that program execution is deterministically determined by the input.

Definition 1 (Test completeness)**.** A test suite T is *complete* with respect to the type of an expression e, written $\textsc{complete}_T(e)$, if execution of T covers all possible types e may have at runtime.

The analysis we present conservatively approximates completeness (which is clearly undecidable): if the analysis reports that the completeness condition is satisfied, then this is indeed the case, but the analysis may be unable to prove completeness although the condition semantically holds.

Our approach involves four components: 1) a dynamic execution of the test suite, 2) a static dependence analysis, 3) a static type analysis, and 4) a test completeness analysis. In this section we present an overview of the approach and explain the purpose of each component, while the subsequent sections give more details about the components.

3.1 Combining Over- and Under-approximation

Our starting point is a well-known fact about dynamic execution and static analysis: executing a test suite constitutes an under-approximation of the program behavior, whereas a static analysis (in the abstract interpretation style) can provide an over-approximation. If the two agree, for example regarding the possible types of an expression e, then the completeness condition is trivially satisfied. Obviously, no further tests can then possibly reveal new types of e.

For the program code shown in lines 36–37, a simple static analysis is able to show that the only possible value of `x` in line 37 is the `A` object created in line 36. That information tells us not only that line 37 cannot fail with a message-not-understood runtime error, but also that the only possible callee is the `m` method in class `A`.

```
36 x = new A();
37 x.m();
```

For scalability reasons we wish to use only context-insensitive and path-insensitive static analysis. (An analysis is *context-insensitive* if it does not distinguish between different calling contexts inter-procedurally [39], and it is *path-insensitive* it if does not distinguish between the different paths that may lead to a given program point intra-procedurally [5].) The precision of such an analysis may suffice for large parts of a program, but programs written in dynamically typed languages often contain code that require more heavyweight program analysis techniques, such as, refinement typing [44] or context sensitivity [3]. The following example shows a typical case of *value-dependent types.*

```
38 class A {
39   m() { ... }
40 }
41 class B {}
42
43 f() {
44   var t = 42;
45   A x = g(t);
46   x.m();
47 }
```

```
48 g(a) {
49   var r;
50   if (a > 0) {
51     r = new A();
52   } else {
53     r = new B();
54   }
55   return r;
56 }
```

Here, `A` has a `m` method and `B` does not. The function `g` is overloaded as its argument determines whether it returns an object of type `A` or `B` (assume that some other code not shown here contains a call to `g` with argument `0`). The call in line 45 clearly always returns an `A` object, but this fact cannot be obtained by a context-insensitive static analysis alone (it would infer that the type is either `A` or `B`). Nor is it obvious by executing a test suite covering `f` that `A` is the only possibility. If the call to `g` instead returned a `B` object, then the program would fail at runtime, in checked mode with a subtype-violation error in line 45 and in production mode with a message-not-understood error in line 46.

3.2 Exploiting Tests and Dependencies

Our key insight is that it is possible through a *combination* of lightweight static analysis and execution of a test suite to obtain completeness guarantees for the kind of code shown in the previous example.

The dependence analysis component One component in our system is a context-insensitive and path-insensitive *dependence analysis* that over-approximates the dependencies of each expression in the given program. Unlike traditional dependence analyses, this one considers both value and type dependencies. (This dependence analysis is described in more detail in Section 4.) For example, it infers that the *type* of `r` in line 55 depends (only) on the *value* of the parameter `a`. It also tells us that the parameter passed to `g` in line 45 has no dependencies (it is a constant). By combining these pieces of information, we see that *a single concrete execution of line 45 suffices* for learning all the possible types of the return value at that call. Thus, we run the test suite, and if it covers line 45 we learn that the only possible type is `A`—in other words, the test suite is complete with respect to the type of the return value of this particular call. Notice that the static analysis alone does not have this information; we need the concrete execution too.

The type analysis component The `bar` function shown in lines 57–69 is a Dart version of a Ruby example by An et al. [2]. Assume that elsewhere in the program, there are calls to `bar` with arguments `true` and `false`. A purely static analysis would require path sensitivity to be able to prove that `y` is always a number in line 65 (so that the + operation is guaranteed to succeed) and a string in line 67 (so that it has a `length` field).

```
57 bar(p) {
58   var y;
59   if (p) {
60     y = 3;
61   } else {
62     y = "hello";
63   }
64   if (p) {
65     y + 6;
66   } else {
67     y.length;
68   }
69 }
```

We now show how we can obtain the necessary precision without path sensitive static analysis. The dependence analysis gives us that the type of `y` in line 65 and the type of `y` in line 67 can only depend on the value of the parameter `p`. As the actual value of `p` is unknown to the dependence analysis, we need more information to prove type safety of lines 65 and 67. For this reason, we include another component in our system: a context-insensitive and path-insensitive *type analysis* that provides an over-approximation of the possible types of all expressions. For the `bar` example, the type analysis tells us that the value of `p` can only be `true` or `false`. Now, notice that by combining this information with the dependence information we see that if executing the test suite includes call to `bar` with both these two values then the test suite is complete with respect to the type of `y` in lines 65 and 67. We thereby know that runtime type errors cannot occur in those lines.

The Rubydust technique by An et al. [2] is able to infer sound types for the `bar` function if all possible paths inside the function are executed by the tests. For this particular example, Rubydust can therefore, like our technique, infer sound and precise types using only two executions of the function. However, our technique differs in several important aspects: (1) Rubydust infers sound types *if* all possible paths are executed, but it does not know whether or not that condition is satisfied (in this example, the control-flow of the function suggests that there are four paths, but only two are possible because of the branch correlation); in contrast, our technique is able to tell that the two executions suffice. (2) In this example, the fact that the two branches are correlated is quite obvious and could be inferred by a very simple static analysis, and that information could easily be incorporated into Rubydust. However, our technique is capable of reaching the conclusion about the type of `y` without reasoning explicitly about branch correlations. (3) If `bar` contained additional branches with code not affecting `y`, then the Rubydust soundness criterion would require more than two executions, whereas our technique would still only require two, to show type safety for the operations on `y`.

As these examples suggest, we use two lightweight static analyses: a dependence analysis and a type analysis. A central part of our approach for inferring test completeness facts is combining the information provided by these analyses with the information obtained from executing the test suite. In Section 6 we explain how this can be achieved, a key step being a mechanism for substituting type properties according to dependencies.

3.3 Using the Inferred Completeness Facts

Program correctness For Dart programs we are particularly interested in test completeness at assignments, calls, and field lookup operations. As mentioned in Section 2, running Dart programs may encounter message-not-understood errors at property lookup operations and subtype-violation errors at implicit casts, even in programs that are accepted without warnings by the static type checker. For a programmer, knowing that his test suite is complete for such properties gives confidence about the correctness of the program.

Test adequacy The notion of test completeness with respect to type properties directly gives rise to a new metric for test adequacy alongside statement coverage, branch coverage, path coverage, etc. [49]:

Definition 2 (Type coverage)**.** For a given set of expressions X in a program and a test suite T, the *type coverage* of T, denoted $\mathcal{C}_T(X)$, is computed as

$$\mathcal{C}_T(X) = \frac{|\{x \in X \mid \textsc{complete}_T(x)\}|}{|X|}$$

As X, one may select, for example, the set of expressions that are subjected to implicit casts in a unit of code of interest, e.g. a method, class, or library. A type coverage of 100% would ensure that the test suite is adequate for revealing all cast errors that are possible in that unit of code.

Traditional coverage metrics are not suitable for giving such guarantees. For example, full statement coverage or full branch coverage is not always sufficient, and full path coverage is impossible to achieve whenever loops are involved [49]. Other techniques that focus on branches and paths, such as, basis path testing using cyclomatic complexity [32], have similar limitations.

By selecting X as the set of receiver expressions of method calls (e.g. `x` in line 46) and the function expressions at function calls (e.g. `g` in line 45) in a unit of code, then 100% type coverage implies that the call graph has been fully exercised in that code. A programmer may use such information to guide further testing. For example, if a test suite has full statement coverage for two classes `C` and `D`, maybe only 30% of the part of the call graph that involves `C` has been covered, while the number is 95% for `D`, in which case the programmer should perhaps prioritize adding tests targeting `C` rather than `D`. We leave it to future work to evaluate the practical usefulness of reporting such type coverage numbers to developers and how the type coverage metric correlates with other metrics and with errors; for the rest of this paper we focus on the use of test completeness in checking type safety and inferring call graphs.

Type filtering A well-known trick in points-to analysis and dataflow analysis is to use type tests (e.g., casts) that appear in the program as *type filters* [40, 3, 20]. The type-related completeness facts inferred by our analysis can of course be exploited for such type filtering: if we have inferred that, for example, `y` is definitely a string in line 67 then a subsequent analysis may use that fact to filter away spurious values that are not strings. As part of our evaluation in Section 7, we demonstrate that performing type filtering based on the completeness facts inferred by our analysis can have a substantial impact on the precision of type analysis and call graph analysis.

The lightweight type analysis we use as one of our components can directly use this mechanism itself, and increasing precision of this component may lead to more completeness facts being inferred. It therefore makes sense to repeat the entire analysis, boosting precision using type filtering based on type information inferred in the previous iteration. For now, this remains an intriguing observation, however; although it is possible to construct examples where the completeness analysis becomes more precise by such a feedback mechanism, we have not yet encountered much need in practice.

Optimizations The inferred completeness facts may also be used for optimizations, for example, removal of runtime type checks, dead-code elimination, replacement of indirect calls by direct calls, inlining, and type specialization [8, 30, 20, 23, 26]; we leave such opportunities for future work.

4. DEPENDENCE ANALYSIS

As motivated in the preceding section, a key component of our technique is a static dependence analysis. Unlike traditional dependence analyses as used in, for example, optimization [17], information flow analysis [36] and program slicing [42] we are interested in both value and type dependencies. We therefore introduce a general *dependence relation* denoted $\lhd$, which is a binary relation over abstractions of runtime states at different program points (inspired by the notion of abstract dependence by Mastroeni and Zanardini [31]). For example, we have seen that the type of `y` at the program point after line 65 depends on the value of `p` in line 57, which we write as $\text{TYPE}_\mathtt{y}[65] \lhd \text{VAL}_\mathtt{p}[57]$. The dependence relation is computed using a whole-program analysis, but all dependencies are intra-procedural in the sense that they relate variables and parameters within the same function.

More generally, the dependence information we need can be expressed via different abstractions of runtime states:

Definition 3 (Type, value, and top abstraction)**.** The *type* abstraction $\text{TYPE}_\mathtt{x}$ for a variable `x` maps a state σ to the runtime type of `x` in σ. The *value* abstraction $\text{VAL}_\mathtt{x}$ instead maps σ to the value of `x` in σ. The *top* abstraction $\top$ is the identity function on program states (we use this abstraction later to express dependencies that are unknown due to analysis approximations).

The dependence relation can now be expressed as a relation of the form $\pi[c] \lhd \pi'[c']$ such that $\pi, \pi' \in \Pi$ and $c \in \mathbb{C}$ where Π is a family of state abstractions and $\mathbb{C}$ is a set of program points. (The program point associated with a given line is the one immediately after that line.) We want the dependence relation being computed to conservatively approximate all dependencies in the following sense.

Property 1 (Valid dependence relation)**.** Given any two executions of the program that both reach a program point c inside a function f with entry program point c_0, let σ and σ' be the states at c for the two executions, respectively, and similarly let σ_0 and σ'_0 be the states at c_0 when f was entered. If there exists some state abstraction $\pi \in \Pi$ where $\pi(\sigma) \neq \pi(\sigma')$ then there must exist some $\pi' \in \Pi$ where $\pi[c] \lhd \pi'[c_0]$ and $\pi'(\sigma_0) \neq \pi'(\sigma'_0)$.

Intuitively, if two executions disagree on π at program point c, then they must also disagree on π' at the entry of the function containing c, where π' is some state abstraction that π depends on. For example, if two executions disagree on the type of `r` in line 55 then they must also disagree on the value of `a` in line 48, so for our choice of state abstractions, the dependence relation must include the fact that the type of `r` depends on the value of `a`.

Program representation To concisely explain how the dependence analysis works we represent a program as a control flow graph $(\mathbb{C}, \hookrightarrow)$ where $\mathbb{C}$ is now a set of nodes corresponding to primitive instructions of the different kinds as shown in Figure 2 and $\hookrightarrow \subseteq \mathbb{C} \times \mathbb{C}$ is the intra-procedural control flow relation between nodes. We let $\hookrightarrow^+$ be the transitive closure of $\hookrightarrow$.

We assume nested expressions have been flattened, so all primitive instructions operate directly on local variables (which include function parameters) and nodes are uniquely identified by the line number. Every call is represented by two nodes: a `call` node and an associated `aftercall` node.

$\mathtt{entry}[\mathtt{f}, \mathtt{p}_1, \ldots, \mathtt{p}_n]$	the entry of function `f` with parameters $\mathtt{p}_1, \ldots, \mathtt{p}_n$
$\mathtt{call}[\mathtt{f}, \mathtt{x}_1, \ldots, \mathtt{x}_n]$	calls function `f` with arguments $\mathtt{x}_1, \ldots, \mathtt{x}_n$
$\mathtt{aftercall}[\mathtt{x}]$	writes the returned value to `x`
$\mathtt{const}[\mathtt{x}, p]$	writes constant value p to `x`
$\mathtt{new}[\mathtt{x}, \mathtt{D}]$	writes new `D` instance to `x`
$\mathtt{assign}[\mathtt{x}, \mathtt{y}]$	writes value of `y` to `x`
$\mathtt{load}[\mathtt{x}, \mathtt{y}, \mathtt{f}]$	writes value of `y.f` to `x`
$\mathtt{store}[\mathtt{x}, \mathtt{f}, \mathtt{y}]$	writes value of `y` to `x.f`
$\mathtt{is}[\mathtt{x}, \mathtt{D}, \mathtt{y}]$	writes *true* to `x` if value of `y` is a subtype of `D`, *false* otherwise
$\mathtt{binop}[\mathtt{x}, \mathtt{y}, \oplus, \mathtt{z}]$	writes the result of the operation $\mathtt{y} \oplus \mathtt{z}$ to `x`
$\mathtt{if}[\mathtt{x}]$	a branch on condition `x`
$\mathtt{phi}[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, B]$	writes $\phi(\mathtt{x}_1, \mathtt{x}_2)$ to `x` after branch
$\mathtt{return}[\mathtt{x}]$	returns the value stored in `x`

Figure 2: Primitive instructions.

The `call` node is labeled by the name of the function being called (we explain in Section 5 how indirect function/method calls are resolved), and for simplicity we assume every call has a single callee. The program representation is in SSA form [12], and every ϕ-node $\mathtt{phi}[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, B]$ is associated with the set B of all `if`-nodes that dominate $\mathtt{x}_1$ or $\mathtt{x}_2$. Due to the use of SSA, each variable name uniquely corresponds to a node where the variable is assigned its value (where parameters are considered to be assigned at the function entry node), and we sometimes use the variable name to denote the program point after that node. Furthermore, we let entry_c denote the entry program point of the function containing node c. The meaning of the other kinds of nodes is straightforward.

The dependence analysis We compute the dependence relation $\lhd$ as the smallest fixpoint of the rules shown in Figure 3 (in addition, $\lhd$ is reflexive and transitive).

The ASSIGN rule for an assignment $\mathtt{assign}[\mathtt{x}, \mathtt{y}]$ establishes that the type and value of `x` depend on the type and value of `y`, respectively. More precisely, for $\pi = \text{TYPE}_\mathtt{x}$ we see that the type of `x` at the program point where `x` has been defined, i.e. $\text{TYPE}_\mathtt{x}[\mathtt{x}]$, depends on the type of `y` at the program point where `y` has been defined, i.e. $\text{TYPE}_\mathtt{y}[\mathtt{y}]$, and similarly for the value. The BINOP rule shows that the value of the result depends on the values of the two operands; however, the *type* of the result is always fixed by the operator so it does not depend on the operands. The IS rule for the instruction $\mathtt{is}[\mathtt{x}, \mathtt{D}, \mathtt{y}]$ shows that the *value* of `x` depends on the *type* of `y`. (The *type* of `x` is always boolean.) A rule for $\mathtt{const}[\mathtt{x}, p]$ can be entirely omitted, since neither the type nor the value of constants depend on anything.

To keep the analysis lightweight we choose to entirely avoid tracking dependencies on the heap. This is modeled using the $\top$ abstraction. In the LOAD rule for $\mathtt{load}[\mathtt{x}, \mathtt{y}, \mathtt{f}]$, the type and value of `x` conservatively depend on the entire state at the entry of the function, i.e. $\top[\mathit{entry}_c]$. With this coarse model of the heap, it is safe to omit a rule for $\mathtt{store}[\mathtt{x}, \mathtt{f}, \mathtt{y}]$. The NEW rule captures the fact that the value being produced by $\mathtt{new}[\mathtt{x}, \mathtt{D}]$ is unknown to the analysis (whereas the type is trivially `D`, without any dependencies).

The PHI rule models dependencies that are due to the data *and* control flow at branches, which are represented by $\mathtt{if}[\mathtt{y}]$ and $\mathtt{phi}[\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2, B]$ nodes. First, the type and value of `x` depend on the type and value, respectively, of both $\mathtt{x}_1$ and $\mathtt{x}_2$

(Assign)
$$\frac{\texttt{assign}[\mathtt{x},\mathtt{y}] \in \mathbb{C} \qquad \pi \in \{\text{TYPE}, \text{VAL}\}}{\pi_{\mathtt{x}}[\mathtt{x}] \lhd \pi_{\mathtt{y}}[\mathtt{y}]}$$

(BinOp)
$$\frac{\texttt{binop}[\mathtt{x},\mathtt{y},\oplus,\mathtt{z}] \in \mathbb{C} \qquad l \in \{\mathtt{y},\mathtt{z}\}}{\text{VAL}_{\mathtt{x}}[\mathtt{x}] \lhd \text{VAL}_{l}[l]}$$

(Is)
$$\frac{\texttt{is}[\mathtt{x},\mathtt{D},\mathtt{y}] \in \mathbb{C}}{\text{VAL}_{\mathtt{x}}[\mathtt{x}] \lhd \text{TYPE}_{\mathtt{y}}[\mathtt{y}]}$$

(Load)
$$\frac{c = \texttt{load}[\mathtt{x},\mathtt{y},\mathtt{f}] \in \mathbb{C} \qquad \pi \in \{\text{TYPE}, \text{VAL}\}}{\pi_{\mathtt{x}}[\mathtt{x}] \lhd \top[\mathit{entry}_c]}$$

(New)
$$\frac{c = \texttt{new}[\mathtt{x},\mathtt{D}] \in \mathbb{C}}{\text{VAL}_{\mathtt{x}}[\mathtt{x}] \lhd \top[\mathit{entry}_c]}$$

(Def)
$$\frac{c \in \mathbb{C} \qquad \pi \in \{\text{TYPE}, \text{VAL}\}}{\pi_{\mathtt{x}}[c] \lhd \pi_{\mathtt{x}}[\mathtt{x}] \qquad \top[c] \lhd \top[\mathit{entry}_c]}$$

(Phi)
$$\frac{\begin{array}{cc} \texttt{phi}[\mathtt{x},\mathtt{x}_1,\mathtt{x}_2,B] \in \mathbb{C} & \pi \in \{\text{TYPE}, \text{VAL}\} \\ \texttt{if}[\mathtt{y}] \in B & l \in \{\mathtt{x}_1,\mathtt{x}_2,\mathtt{y}\} \end{array} \qquad d = \begin{cases} \pi_l[l] & \text{if } l \in \{\mathtt{x}_1,\mathtt{x}_2\} \\ \text{VAL}_l[l] & \text{if } l = \mathtt{y} \end{cases}}{\pi_{\mathtt{x}}[\mathtt{x}] \lhd d}$$

(Call)
$$\frac{\begin{array}{ccc} c = \texttt{call}[\mathtt{f},\mathtt{x}_1,\ldots,\mathtt{x}_n] \in \mathbb{C} & c_e = \texttt{entry}[\mathtt{f},\mathtt{p}_1,\ldots,\mathtt{p}_n] \in \mathbb{C} & c \hookrightarrow c_a \\ c_a = \texttt{aftercall}[\mathtt{x}] \in \mathbb{C} & c_r = \texttt{return}[\mathtt{y}] \in \mathbb{C} & c_e \hookrightarrow^{+} c_r \end{array} \qquad d = \begin{cases} \text{VAL}_{\mathtt{x_i}}[x_i] & \text{if } \pi_{\mathtt{y}}[\mathtt{y}] \lhd \text{VAL}_{\mathtt{p_i}}[c_e] \\ \text{TYPE}_{\mathtt{x_i}}[x_i] & \text{if } \pi_{\mathtt{y}}[\mathtt{y}] \lhd \text{TYPE}_{\mathtt{p_i}}[c_e] \\ \top[\mathit{entry}_c] & \text{if } \pi_{\mathtt{y}}[\mathtt{y}] \lhd \top[c_e] \end{cases}}{\pi_{\mathtt{x}}[\mathtt{x}] \lhd d}$$

Figure 3: Rules for type and value dependence.

(as reflected by the case $l \in \{\mathtt{x}_1, \mathtt{x}_2\}$). Second, the type and value of `x` also depend on the value of the branch conditions in B (corresponding to the case where $l = \mathtt{y}$). It is standard to compute control flow dependence using post-dominance information [12]. A node $c \in \mathbb{C}$ is control flow dependent on $c' \in \mathbb{C}$ if c' determines whether c is executed, i.e. (*i*) there is a path from c to c' where every node in the path is post-dominated by c', and (*ii*) c is not post-dominated by c'. Since we assume the control flow graph to be on SSA form, control flow dependence for variables is already explicit in the ϕ-nodes.

The Call rule exploits the dependencies computed between the return value and the parameters of the callee, to obtain dependencies between the resulting value and the arguments at the call site. In this rule, c is a **call** node and c_a is the associated **aftercall** node at the call site, while c_e is the callee **entry** node and c_r is the **return** node. The dependencies for the result `x` are found simply by substituting the dependencies of the return value `y` according to the parameter binding.

Finally, the Def rule has the consequence that the type or value of a variable `x` at any program point c can always be determined from its type or value at the definition of `x`, and the top abstraction at c depends on the top abstraction at the entry of the function. This rule is strictly not necessary, but it simplifies the use of $\lhd$ in Section 6.

Notice that even though type abstractions are conservative approximations of value abstractions, all combinations of dependencies between types and values can occur; in particular, it is possible that the *value* of a variable `x` depends on the *type* of another variable `y` (due to the Is rule), and that the *type* of `x` depends on the *value* of `y` (due to the Phi rule).

Proposition 1 (Soundness of computed dependencies)**.** For any program, the relation $\lhd$ defined by the least fixpoint of the dependence rules satisfies Property 1.

Example For the example in lines 38–56, the dependence analysis infers, in particular, that $\text{TYPE}_{\mathtt{r}}[55] \lhd \text{VAL}_{\mathtt{a}}[48]$ but *not* $\text{TYPE}_{\mathtt{r}}[55] \lhd \top[48]$, and that there is no π such that $\text{TYPE}_{\mathtt{t}}[45] \lhd \pi[43]$. Therefore the Call rule gives that there is also no π' such that $\text{TYPE}_{\mathtt{x}}[\mathtt{x}] \lhd \pi'[43]$, i.e., the type of `x` does not depend on the input to `f`.

5. TYPE ANALYSIS

The static type analysis component serves two purposes: it resolves indirect calls for the dependence analysis in Section 4, and it computes an over-approximation of the possible types of every expression, which we use in the test completeness analysis in Section 6. The type analysis is simply a context-insensitive and path-insensitive subset-based analysis that simultaneously tracks functions/methods and types. This is a well-known analysis technique (see e.g. the survey by Sridharan et al. [40]), so due to the limited space we omit a detailed description of how this component works. The heap is modeled using allocation-site abstraction [9], and we use flow-sensitivity only for local variables. We choose a lightweight analysis for the reasons given in Sections 1 and 3.

In Dart, all values are objects, including primitive values and functions. The analysis abstracts primitive values by their type (e.g. `bool` or `String`) and treats each function and method as having its own type. As output, for every program point c the analysis returns an abstract state $\hat{\sigma}_c$ that over-approximates all concrete states that may appear at c when the program is executed. We define $\hat{\sigma}_c(e)$ to be the set of types the expression e may evaluate to according to $\hat{\sigma}_c$.

Example Consider a call to the `cross` function from Figure 1

```
70  x = cross(y,z);
```

in a context where the type analysis finds that the type of `y` is either `vec3` or `vec2` and the type of `z` is `vec3`. That is, $\hat{\sigma}_c(\mathtt{y}) = \{\mathtt{vec3}, \mathtt{vec2}\}$ and $\hat{\sigma}_c(\mathtt{z}) = \{\mathtt{vec3}\}$ where c is the program point at the call. (This example uses a direct call, so the connection between the call and the callee is trivial.) Assuming there are other calls to `cross` elsewhere in the program, the context-insensitive type analysis has only imprecise information about the possible return types for this function. However, the dependence analysis has inferred that

(Base)

$$\frac{\begin{array}{c}\forall(a_1,\ldots,a_n)\in\pi_1(\hat{\sigma}_c)\times\cdots\times\pi_n(\hat{\sigma}_c):\\ \exists t\in T,\sigma\in[\![t]\!]_c:\ \forall i=1,\ldots,n:\ \pi_i(\sigma)=a_i\end{array}}{T\vdash\pi_1,\ldots,\pi_n[c]}$$

(Inductive-ToEntry)

$$\frac{\{\pi'_1,\ldots,\pi'_m\}=\{\pi'\mid\pi_i[c]\lhd\pi'[\mathit{entry}_c]\}\qquad T\vdash\pi'_1,\ldots,\pi'_m[\mathit{entry}_c]}{T\vdash\pi_1,\ldots,\pi_n[c]}$$

(Inductive-ToCall)

$$\frac{\begin{array}{llll}c=\mathtt{call}[\mathtt{f},\mathtt{x}_1,\ldots,\mathtt{x}_\mathtt{n}]\in\mathbb{C} & c_e=\mathtt{entry}[\mathtt{f},\mathtt{p}_1,\ldots,\mathtt{p}_\mathtt{n}]\in\mathbb{C} & c\hookrightarrow c_a & \{\pi'_1,\ldots,\pi'_m\}=\\ c_a=\mathtt{aftercall}[\mathtt{x}]\in\mathbb{C} & c_r=\mathtt{return}[\mathtt{y}]\in\mathbb{C} & c_e\hookrightarrow^+ c_r & \quad\{\textsc{val}_{\mathtt{x_i}}\mid\pi_\mathtt{y}[\mathtt{y}]\lhd\textsc{val}_{\mathtt{p_i}}[c_e]\}\ \cup\\ & & & \quad\{\textsc{type}_{\mathtt{x_i}}\mid\pi_\mathtt{y}[\mathtt{y}]\lhd\textsc{type}_{\mathtt{p_i}}[c_e]\}\ \cup\\ \pi\in\{\textsc{type},\textsc{val}\} & T\vdash\pi'_1,\ldots,\pi'_m[c] & & \quad\{\top\mid\pi_\mathtt{y}[\mathtt{y}]\lhd\top[c_e]\})\end{array}}{T\vdash\pi_\mathtt{x}[c_a]}$$

Figure 4: Rules for test completeness.

the return type only depends on the types of the parameters. This allows the test completeness analysis, presented in the following section, to conclude that two executions suffice to cover all possible types of `x`: one where `y` has type `vec3` and one where it has type `vec2`. This example demonstrates the power of combining dependence analysis and type analysis.

Now consider a slight modification of the example, where the type of both `y` and `z` is either `vec3` or `vec2`. Our technique then requires four executions of the call to ensure that all relevant combinations are covered. This suggests that *relational* dependence and type information may be valuable: since we only obtain test completeness guarantees at a call when all combinations of dependencies have been exercised, it may be beneficial to know that only some combinations are relevant. This is an opportunity to explore in future work.

6. TEST COMPLETENESS

In this section we present a proof system for test completeness, i.e., for proving $\textsc{complete}_T(\mathtt{x})$ for a given variable `x` in some program with a test suite T (see Definition 1). (We use the program representation from Section 4, so we assume without loss of generality that the expression of interest is simply a variable `x`.)

The proof rules, shown in Figure 4, rely on three ingredients: the execution of T, the dependence relation $\lhd$ from Section 4, and for each program point c the abstract state $\hat{\sigma}_c$ produced by the type analysis from Section 5. Each test input $t \in T$ gives rise to a program execution, which can be seen as a sequence of concrete states at different program points. We write $[\![t]\!]_c$ for the set of states encountered at program point c when t is executed.

A simple judgment $T \vdash \pi[c]$ intuitively means that the test suite T is complete with respect to the state abstraction π (see Definition 3) at program point c. In other words, the judgment holds if for every abstract value in $\pi(\sigma_c)$ where σ_c is a runtime state at c in *some* execution of the program, there exists a test in T that also encounters that abstract value at c. In particular, we are interested in the type abstraction $\textsc{type}_x$ and the program point where `x` is defined. We therefore aim for the following connection between proof judgments and test completeness for type properties.

Proposition 2 (Soundness of test completeness analysis)**.**

$$T \vdash \textsc{type}_\mathtt{x}[\mathtt{x}] \text{ implies } \textsc{complete}_T(\mathtt{x})$$

To show $\textsc{complete}_T(\mathtt{x})$ for some variable `x`, we thus attempt to derive $T \vdash \textsc{type}_\mathtt{x}[\mathtt{x}]$.

More generally, judgments may involve *multiple* state abstractions: $T \vdash \pi_1, \ldots, \pi_n[c]$. The intuitive interpretation of such a judgment is that T is complete with respect to the product of the state abstractions $\pi_1, \ldots, \pi_n$ at program point c.

We now briefly describe the rules from Figure 4.

The Base rule corresponds to the observation in Section 3.1 that completeness sometimes can be shown using the information from the type analysis. To understand the rule in its full generality, consider first this special case:

$$\frac{\forall a \in \hat{\sigma}_c(\mathtt{x}) : \exists t \in T, \sigma \in [\![t]\!]_c : \textsc{type}_\mathtt{x}(\sigma) = a}{T \vdash \textsc{type}_\mathtt{x}[c]}$$

This rule states that a test T is complete for the type of `x` if for all the types $a \in \hat{\sigma}_c(\mathtt{x})$ that can be observed according to the type analysis (Section 5), there exists an execution that reaches program point c with a concrete state σ where `x` has type a.

A first step from this special case toward the general Base rule is to generalize it from using the type abstraction $\textsc{type}$ to use any state abstraction $\pi \in \Pi$. For this, we introduce notation for lifting a state abstraction π to operate on abstract states produced by the type analysis: we define $\pi(\hat{\sigma}_c) = \{\pi(\sigma_c) \mid \hat{\sigma}_c \text{ is the type abstraction of } \sigma_c\}$. Note that $\pi(\hat{\sigma}_c)$ can be infinite, for example if $\pi = \top$. This corresponds to a completeness condition that requires an infinite set of executions, so in that situation we can simply give up. Another interesting case is when $\pi = \textsc{val}_\mathtt{x}$ for some variable `x`. This occurs when our analysis is required to prove $T \vdash \textsc{val}_\mathtt{p}[c]$ in the example in lines 57–69 in Section 3 (where c is the program point at the entry of `bar`). Because `p` is a boolean according to the type analysis, $\textsc{val}_\mathtt{p}(\hat{\sigma}_c)$ contains only the two values `true` and `false`, so two executions suffice to prove $T \vdash \textsc{val}_\mathtt{p}[c]$.

The last step to the general Base rule is to account for judgments with multiple state abstractions $\pi_1, \ldots, \pi_n$. For this case, we simply require observations of all combinations of abstract values in $\pi_1(\hat{\sigma}_c) \times \cdots \times \pi_n(\hat{\sigma}_c)$.

The Inductive-ToEntry rule uses the dependence relation $\lhd$ to prove a completeness property at a program point c by a completeness property at the function entry entry_c.

The Inductive-ToCall rule mimics the Call rule from the dependence analysis (Figure 3), substituting completeness properties at calls according to the dependencies of the callee. This corresponds to the reasoning used for the example in Section 3.2 for the call in line 45.

Completeness proofs can be obtained by proving ground facts using the BASE rule, and then deriving all the others using the inductive rules whenever their premises are satisfied. This procedure terminates, as it works intra-procedurally and the induction proceeds in program order.

Using test completeness for type filtering As suggested in Section 3.3, the completeness facts being inferred can be plugged in as type filters in a type analysis. For example, we can run the type analysis described in Section 5 a second time, now using type filtering.

Let X be the set of observed runtime types for a variable `x` where $\text{COMPLETE}_T(\texttt{x})$. By Proposition 2 this is the set of all the possible types `x` may have in any execution. During the type analysis, for every abstract state, we can filter out those types of `x` that are not in X. Removing those spurious types may improve precision throughout the program.

7. EXPERIMENTAL EVALUATION

Our experimental evaluation addresses the following four research questions.

Q1 To what extent is the technique capable of showing *test completeness* for realistic Dart programs and test suites? More specifically, what is the *type coverage* being computed for such programs and test suites?

Q2 Does the hybrid static/dynamic approach result in better precision for soundly checking absence of runtime type errors (message-not-understood and subtype-violation errors) and producing precise call graph information (outdegree of calls, reachable functions), compared to an analysis that does not exploit the test suites?

Q3 How important is the dependence analysis, which is a central component in our technique, for the precision of the computed type coverage?

Q4 In situations where the analysis cannot show test completeness, is the cause likely to be insufficient tests or lack of precision in the static analysis components?

Implementation and benchmarks Our implementation, GOODENOUGH, consists of the four components listed in Section 3, including the type filtering technique that uses completeness facts to improve the type analysis. For logging the runtime types of expressions we use a small runtime instrumentation framework inspired by Jalangi [38]. We keep the runtime overhead low by only observing types at variable reads and calls in the application code, and we do not instrument libraries.

The evaluation is based on 27 benchmarks, ranging from small command-line and web applications to sample applications that demonstrate the use of libraries. The benchmarks are listed is in the first column of Table 1. The second column shows the size of each benchmark in number of lines of code (excluding unreachable library code). We use the existing test suites available for the command-line applications; for the web applications we obtain test suites by manually exercising the applications via a browser for one minute each. To obtain some degree of confidence that the static analysis components are sound, we check that all runtime observations agree with the static analysis results.

The implementation, benchmarks, test suites, and details of the experimental results are available online.[4]

Table 1: Results.

>0 %	1 %	5 %	30 %	100 %

Program	LOC	$C_T(X)$	MNU	UIC	RF	CGS	PI
a_star	1 338	83.6 %	-	-	0.31 %	7.53 %	77.27 %
archive	2 139	87.8 %	11.11 %	-	-	0.17 %	3.45 %
bad_aliens	1 716	77.6 %	4.26 %	18.18 %	-	0.16 %	22.22 %
dartbox2d	8 732	70.9 %	4.13 %	10.10 %	3.87 %	3.32 %	22.52 %
csslib	6 972	82.4 %	6.45 %	3.08 %	1.16 %	1.85 %	8.95 %
Dark	14 506	78.4 %	25.88 %	1.41 %	-	0.55 %	10.63 %
dart_regexp_tester	3 095	84.8 %	-	-	-	-	-
Dartrix	1 594	80.2 %	-	-	0.50 %	1.49 %	66.67 %
dartrocket	11 651	74.7 %	-	-	-	-	-
frappe	5 712	77.8 %	-	-	-	-	-
graphviz	3 275	80.7 %	-	6.25 %	-	-	-
markdown	1 901	89.9 %	-	-	1.98 %	8.57 %	63.72 %
qr	6 118	87.3 %	-	-	-	0.88 %	16.00 %
petitparser	3 882	69.0 %	-	-	0.31 %	3.68 %	33.62 %
pop_pop_win	14 590	71.9 %	-	-	0.04 %	0.86 %	17.45 %
rgb_cube	1 079	86.1 %	-	-	-	-	-
solar	657	85.1 %	-	-	-	-	-
solar3d	7 813	86.5 %	71.74 %	6.90 %	2.89 %	4.65 %	17.65 %
spaceinvaders	1 292	75.9 %	-	-	-	-	-
speedometer	357	91.0 %	-	-	-	-	-
spirodraw	1 220	86.5 %	4.76 %	5.56 %	-	4.54 %	-
sunflower	526	86.4 %	75.00 %	33.33 %	-	-	-
todomvc_vanilladart	3 967	78.4 %	2.08 %	5.00 %	0.15 %	0.93 %	7.58 %
vote	7 867	76.7 %	17.29 %	7.69 %	15.16 %	18.09 %	31.10 %
three	7 432	83.1 %	42.05 %	5.29 %	20.93 %	7.91 %	8.00 %
stats	1 560	84.2 %	35.71 %	9.09 %	-	-	-
tutorial_kanban	940	70.3 %	-	-	-	-	-

Q1: Type coverage

To answer the first question we run GOODENOUGH on each benchmark with its test suite T and measure the type coverage $C_T(X)$, where X is chosen as the set of all the expressions in the program. (Due to Proposition 2, the type coverage numbers we report are safe in the sense that they underapproximate the semantic definition of type coverage given by Definition 2.) The results are shown in the third column of Table 1.

Type coverage is generally high, with an average of 81 %. In other words, the analysis is able to guarantee for 81 % of the expressions that all types that can possibly appear at runtime are in fact observed by execution of the test suite.

Q2: Type errors and call graphs

To investigate whether our hybrid static/dynamic approach to test completeness can be useful for showing absence of type errors and inferring sound call graph information, we compare GOODENOUGH with a variant that does not exploit the runtime observations from the test suite. We select five metrics for measuring the precision:

MNU (*message-not-understood*): number of warnings about possibly failing access to a field or a method;

UIC (*unsafe implicit cast*): number of warnings about potential subtype violations at implicit downcasts (only relevant for checked mode execution);

CGS (*call-graph size*): number of edges in the call graph;

RF (*reached functions*): number of possibly reached functions;

PI (*polymorphic invocation*): number of call sites with multiple potential callees.

The columns MNU, UIC, RF, CGS, and PI in Table 1 show the percentage change of the resulting numbers for the hybrid analysis compared to the fully-static analysis.

We see that the hybrid analysis is able to greatly improve the precision for some benchmarks, while it gives marginal or no improvements in others. This is not surprising, con-

[4] `http://www.brics.dk/goodenough/`

sidering the different nature of the benchmarks and their programming style. For a few benchmarks, the naive fully-static analysis already obtains optimal precision for MNU or UIC, leaving no room for improvements with the hybrid technique. Interestingly, 19 out of 27 benchmarks improve for at least one metric, meaning that the hybrid approach shows advantages across a variety of programs. This confirms that type filters based on inferred completeness facts can have a substantial impact on the precision of the type analysis, as discussed in Section 3.

An important design decision has been that the static analysis components are context- and path-insensitive to keep the system lightweight. By manually studying some of the examples where our hybrid approach obtains high precision, we see that a fully-static alternative would require a high degree of context sensitivity to reach the same conclusions, and it is well known that such analyses do not scale well.

For example, a context-insensitive static analysis is insufficient to reason precisely about the return type of functions similar to `cross` from the *vector_math* library discussed in Section 1. In contrast, GOODENOUGH finds 19 calls to `cross` where the test suite is complete, which enables the static type analysis to filter the inferred types for the return of the calls and thereby avoid several spurious warnings.

Another example is found the *Dark* benchmark:

```
var canvas;
GL.RenderingContext gl;
void startup() {
  canvas = querySelector("#game");
  gl = canvas.getContext("webgl");
  if (gl==null)
    gl = canvas.getContext("experimental-webgl");
  if (gl==null) {
    crash("No webgl", "Go to [...]");
    return;
  }
  ...
}
void start(Level _level) {
  ...
  GL.Texture colorLookupTexture = gl.createTexture();
  ...
}
```

According to the type analysis, `canvas` can be *any* subtype of the HTML `Element` class, and the calls to `getContext` return objects of type `CanvasRenderingContext2D` or `RenderingContext`. With this information, one would conclude that in checked mode execution subtype-violation errors may occur at the two assignments to `gl`, and in production mode the `createTexture` invocation may result in a message-not-understood error since `createTexture` is not declared by `CanvasRenderingContext2D`. Although `canvas` can be an arbitrary HTML element, the call is monomorphic. The results of the calls to `getContext` only depend on the value of the arguments, which are constant. Executing the two branches dynamically is enough to prove type completeness for both calls, which reveals that both calls return objects of type `RenderingContext`, not `CanvasRenderingContext2D`. Type filtering uses this fact to remove spurious dataflow. This example also shows the importance of test coverage: a single execution does *not* suffice to cover all types at both calls.

While lightweight type analysis (e.g. the one from Section 5) cannot reach the conclusion that only `RenderingContext` objects may be returned by `getContext`, a more precise analysis could. In this case, 2-CFA context sensitivity would be needed if we did not exploit the dynamic executions, since the argument to `getContext` is passed to the native function `getContext_Callback_2_`. In other situations, 5-CFA would be necessary to reach the same precision as our hybrid approach.

To correctly handle the example above with a fully static analysis, it would additionally be necessary to precisely model the implementation of `getContext_Callback_2_`: it returns a `RenderingContext` object when the argument is `"webgl"` or `"experimental-webgl"`, and a `CanvasRenderingContext2D` object when the argument is `"2d"`. The Dart SDK contains more than 10 000 external functions, many of which are highly overloaded in such ways. As an example, the method `getParameter` of `RenderingContext` from the WebGL API returns 15 (unrelated) types depending on which of 87 constants the argument matches.[5] Another example is the library function `Element.tag`, which calls a native function that has more than 200 cases involving different types, which a fully static analysis would need detailed models for. Thus, our hybrid approach avoids a massive task of modeling such details, as it only needs the dependence information.

We also observe that test completeness is useful for reasoning about the types of numeric expressions without the need for context sensitivity, constant folding and constant propagation. Indeed, the type of many numerical expressions only depends on the types of the operands. The following simple example from *Dark* is one of many where cast failures may occur in checked mode execution according to a naive type analysis:

```
void move(double iX, double iY, double passedTime) {
  ...
  double frictionXZ = pow(0.0005, passedTime);
  double frictionY = pow(0.2, passedTime);
```

The declared return type of `pow` is `num`, which has subtypes `int` and `double`. Covering the two calls to `bar` with a single execution is enough to ensure completeness.

Q3: Dependence analysis

The dependence analysis finds that 73 % of all function return expressions have no type dependencies on the function entry state. This means that a single execution of those expressions is enough to cover all their possible types. At other 16 % of the return expressions, the return type depends on the types or values of one or more parameters but not on the heap, and for the remaining 11 % the dependence analysis finds dependencies on the heap.

To investigate the importance of the dependence analysis, we have repeated the experiments from Q1 and Q2 using a weaker dependence analysis that does not exploit the call graph produced by the type analysis. Instead, the type and value dependencies of the return value at the call site of all indirect calls are conservatively set to top. This single restriction causes a reduction in the number of completeness facts being proven and a significant degradation in the precision metrics: type coverage drops from 81 % to 77 %, and the precision of type error detection and call graph construction drops to almost the same as the naive fully-static analysis (leaving only a 0.7 % improvement of the UIC metric for the Dark benchmark). These results demonstrate that the dependence analysis is indeed a critical component.

[5] `https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/getParameter`

Q4: Reasons for inability to show test completeness

To answer Q4, we have investigated the behavior of our analysis in selected places where GOODENOUGH is not able to prove test completeness. It is naturally difficult to quantify the reasons, yet some patterns are clear. Our preliminary investigation has identified two main reasons: missing coverage by the test suites, and too coarse heap modeling in the dependence analysis. On the other hand, imprecision in the type analysis does not appear to be significant.

As we have already seen in the *Dark* example in the answer to Q2, coverage is indeed important to prove test completeness. In that case two executions on specific browser versions were needed. In many other cases, we see that simply improving the statement coverage of the test suite would likely improve the type coverage significantly.

For this first design of our technique, we have chosen a simplistic approach to dependence analysis, in particular, by using the top abstraction at all field load operations. Consider the following example, which uses the `parse` function from a Dart HTML5 parser[6].

```
f() {
  DivElement div = parse("<div></div>").firstChild;
}
Document parse(input, ...) {
  var p = new HtmlParser(input, ...);
  return p.parse();
}
```

The `parse` function always returns a tree of HTML elements whose structure is determined solely by the input to `parse`. A dependence analysis that is able to track dependencies on the heap could in principle determine that the lookup of `firstChild` always has the same type, or equivalently, that the expression has no type dependencies. We see similar cases in many of the libraries being used in the benchmarks. This observation suggests that extending the dependence analysis to also track dependencies involving the heap may be a promising direction for future work.

8. RELATED WORK

Types in dynamic languages Several techniques have been proposed to statically reconstruct precise type information for dynamic languages, for example, for optimization purposes [26, 23]. The type inference approach by An et al. [2] is discussed in Section 3.2. Dependent types and flow-based type checking can deal with common dynamic patterns, such as, value-based overloading [10, 29, 35, 44, 41, 21]. These techniques require programmers to provide detailed type annotations. Advanced static analysis has been used to precisely infer and check types in JavaScript [3], however, this has not yet been proven to scale to realistic programs.

Test adequacy and coverage metrics Numerous criteria for deciding whether a test suite is adequate and metrics for measuring coverage have been proposed (the key concepts being described by Zhu et al. [49]), so due to the limited space we can only mention the high-level relations to our work. The focus in the literature is typically on using coverage metrics to guide the effort of testing programs. Common to most of those techniques, including the seminal work by Goodenough and Gerhart [19], is that they do not come with tool support for checking whether a test suite has adequate coverage to guarantee properties like absence of runtime type errors. The general idea of "test completeness" has a long history [19, 25, 24, 7], but until now without the connection to types in dynamic languages.

Hybrid analysis Other hybrids of static and dynamic analysis have been developed to combine the strengths of both parts [48, 11, 6, 22, 27], or to use dynamic executions to guide static analysis [46, 37, 34]. Notably, the method by Yorsh et al. [48] uses automated theorem proving to check that a generalized set of concrete states obtained by dynamic execution covers all possible executions. This involves a notion of abstraction-based adequacy, which is reminiscent of our notion of type coverage. Predicate-complete testing is another related idea [4]. Most hybrid analysis create tests on-the-fly and do not exploit preexisting test suites, but for the dynamic programming patterns that are our primary target, automatically creating useful tests is by itself a considerable challenge. One approach that does use test suites and is designed for a dynamic programming language is blended analysis [46], however, it is unsound in contrast to our technique.

Dependence analysis The concept of dependence used in Section 4 appears in many variations in the literature. From a theoretical point of view, our definitions fit nicely into the framework proposed by Mastroeni and Zanardini [31]. In the field of information flow [18], dependence plays an important role in reasoning about non-interference. In program slicing [47] and compiler optimization (e.g. [17]), program dependence graphs [28] model the dependencies between values used in different statements. The novelty of the dependence analysis in Section 4 is to capture dependencies not only between values but also between types.

9. CONCLUSION

We have presented the notions of test completeness and type coverage together with a hybrid program analysis for reasoning about adequacy of test suites and proving type-related properties. Moreover, we have demonstrated using our implementation GOODENOUGH how this analysis technique is suitable for showing absence of type errors and inferring sound call graph information, in Dart code that is challenging for fully-static analysis.

Many interesting opportunities for future work exist. We plan to explore how better heap modeling in the dependence analysis and relational dependence and type analysis can improve precision further. In another direction, we intend to perform an empirical study of how the type coverage metric correlates with other metrics and with programming errors, and to use type coverage to guide automated test generation. It may also be interesting to use the type analysis results for program optimizations and to apply our approach to other dynamic languages.

10. ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation program (grant agreement No 647544).

[6]`https://github.com/dart-lang/html`

11. REFERENCES

[1] B. Åkerblom and T. Wrigstad. Measuring polymorphism in Python programs. In *Proc. 11th Symposium on Dynamic Languages (DLS)*, 2015.

[2] J. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for Ruby. In *Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[3] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014.

[4] T. Ball. A theory of predicate-complete test coverage and generation. In *Proc. Formal Methods for Components and Objects, 3rd International Symposium (FMCO)*, 2004.

[5] T. Ball and S. K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, 2001.

[6] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. Tetali, and A. V. Thakur. Proofs from tests. *IEEE Transactions on Software Engineering*, 36 (4):495–508, 2010.

[7] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1): 31–45, 1982.

[8] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[9] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[10] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012.

[11] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *Proc. 27th International Conference on Software Engineering (ICSE)*, 2005.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[13] E. W. Dijkstra. Notes on structured programming. Technical Report EWD249, Technological University Eindhoven, 1970.

[14] Ecma International. *Dart Programming Language Specification, ECMA-408, 3rd Edition*, June 2015.

[15] E. Ernst, A. Møller, M. Schwarz, and F. Strocco. Message safety in Dart. In *Proc. 11th Symposium on Dynamic Languages (DLS)*, 2015.

[16] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proc. 35th International Conference on Software Engineering (ICSE)*, 2013.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[18] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.

[19] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *ACM SIGPLAN Notices – International Conference on Reliable Software*, 10(6): 493–510, 1975.

[20] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and K. T. Tekle. Alias analysis for optimization of dynamic languages. In *Proc. 6th Symposium on Dynamic Languages (DLS)*, 2010.

[21] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proc. Programming Languages and Systems - 20th European Symposium on Programming (ESOP)*, 2011.

[22] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. *International Journal on Software Tools for Technology Transfer*, 15(4):291–303, 2013.

[23] B. Hackett and S. Guo. Fast and precise hybrid type inference for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[24] W. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–215, Sept 1976.

[25] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, (4):371–379, 1982.

[26] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf. Improved type specialization for dynamic scripting languages. In *Proc. 9th Symposium on Dynamic Languages (DLS)*, 2013.

[27] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2), 2010.

[28] B. Korel. The program dependence graph in static program testing. *Information Processing Letters*, 24(2): 103–108, 1987.

[29] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: retrofitting type systems for JavaScript. In *Proc. 9th Symposium on Dynamic Languages (DLS)*, 2013.

[30] F. Logozzo and H. Venter. RATA: rapid atomic type analysis by abstract interpretation - application to JavaScript optimization. In *Proc. Compiler Construction, 19th International Conference (CC)*, 2010.

[31] I. Mastroeni and D. Zanardini. Data dependencies and program slicing: from syntax to abstract semantics. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, 2008.

[32] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[33] Microsoft. TypeScript language specification, February 2015. `http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf`.

[34] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. In *Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

[35] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *Proc. 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015.

[36] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[37] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[38] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proc. Symposium on the Foundations of Software Engineering (FSE)*, 2013.

[39] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[40] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 196–232. Springer, 2013.

[41] A. Takikawa, D. Feltey, E. Dean, M. Flatt, R. B. Findler, S. Tobin-Hochstadt, and M. Felleisen. Towards practical gradual typing. In *Proc. 29th European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[42] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[43] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed Scheme. In *Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

[44] P. Vekris, B. Cosman, and R. Jhala. Trust, but verify: Two-phase typing for dynamic languages. In *Proc. 29th European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[45] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *Proc. 10th ACM Symposium on Dynamic Languages (DLS)*, 2014.

[46] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2013.

[47] M. Weiser. Program slicing. In *Proc. 5th International Conference on Software Engineering (ICSE)*, 1981.

[48] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2006.

[49] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.