Analyzing Test Completeness for Dynamic Languages

Anders Møller

joint work with Christoffer Quist Adamsen and Gianluca Mezzetti

CENTER FOR ADVANCED SOFTWARE ANALYSIS

http://cs.au.dk/CASA

Languages with dynamic or optional typing are popular!

- Dart
- JS JavaScript — TypeScript
- Racket — Typed Racket
- python™ — Reticulated Python
- Ruby *A Programmer's Best Friend* — DRuby
- php — hack
- …

```
dynamic cross(dynamic x, dynamic y,
              [dynamic out=null]) {
  if (x is vec3 && y is vec3) {
    return x.cross(y, out);
  } else if (x is vec2 && y is vec2) {
    assert(out == null);
    return x.cross(y);
  } else if (x is num && y is vec2) {
    x = x.toDouble();
    if (out == null) {
      out = new vec2.zero();
    }
    ...
    return out;
  } else if (x is vec2 && y is num) {
    ...
  } else {
    assert(false);
  }
  return null;
}

...
// solve the linear system
dp2perp = cross(dp2, normal);
dp1perp = cross(normal, dp1);
tangent = dp2perp * duv1.x + dp1perp * duv2.x;
```

overloaded – the behavior and return type depend on runtime types of parameters

return type is either `vec3`, `vec2`, `double`, or the type of `out`

assertion failure if unexpected combination of types

runtime type error if values have unexpected types

(code from the Dart libraries *vector_math* and *box2d*)

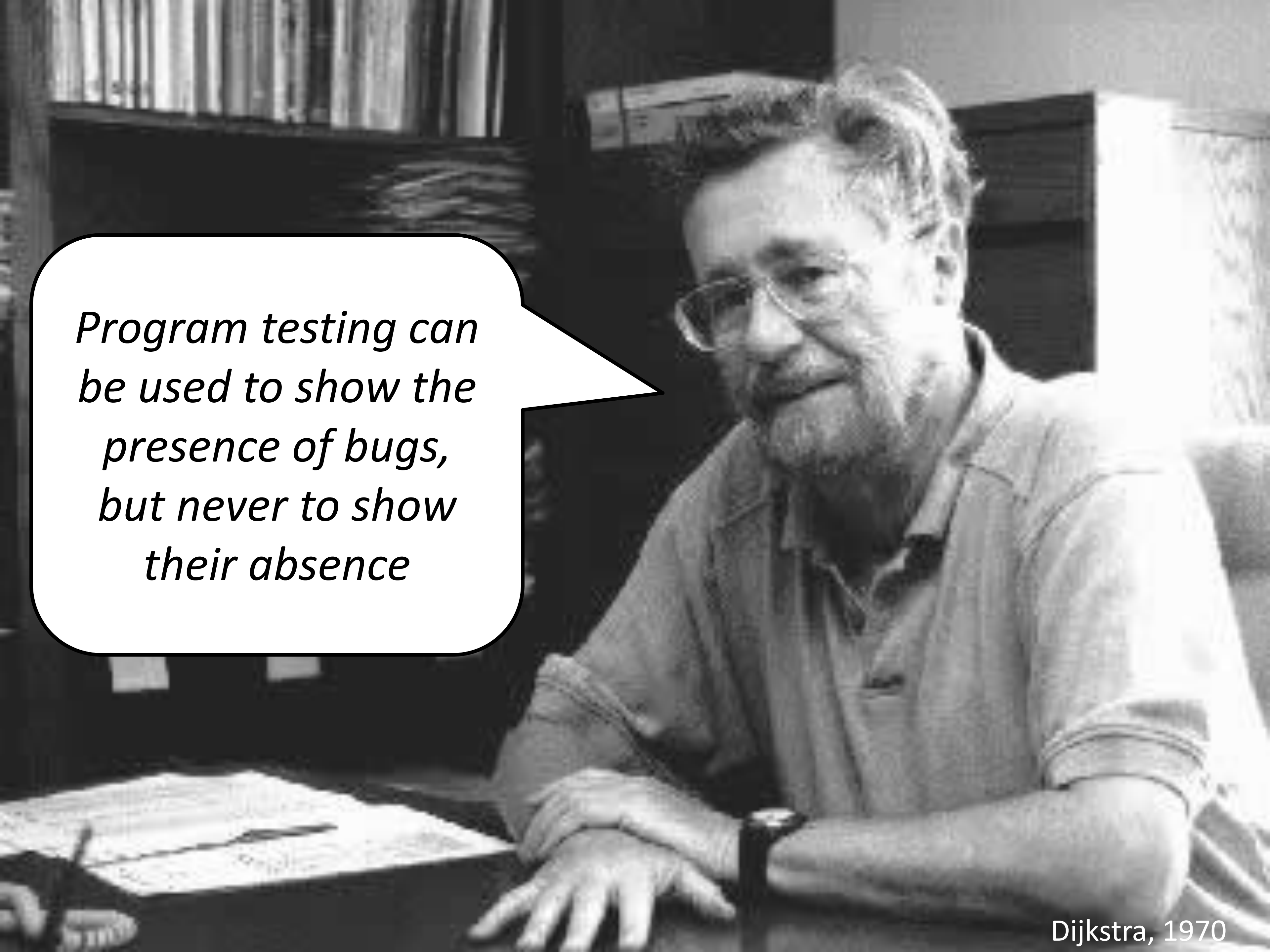
How to ensure absence of runtime type errors in dynamically typed languages?

static analysis?

common programming patterns require
very high analysis precision and/or annotations
(not practical)

examples:

- static determinacy analysis [Andreasen & Møller, OOPSLA 2014],
- refinement types [Vekris et al., ECOOP 2015]


Program testing can be used to show the presence of bugs, but never to show their absence
Dijkstra, 1970

ACM SIGPLAN Notices, 1975

TOWARD A THEORY OF TEST DATA SELECTION*

John B. Goodenough
Susan L. Gerhart**
SofTech, Inc., Waltham, Mass.

Keywords and Phrases:

testing, proofs of correctness

Abstract

This paper examines the theoretical and practical role of testing in ...

properly structured tests are capable of demonstrating the absence of errors in a program. Th

...tually reliable. We explain what makes tests unreliable (for example, we show by example why testing all program statements, predicates, or paths is not usually sufficient to insure test reliability), and we outline a possible approach to developing reliable tests. We also show how the analysis required to define reliable tests can help in checking a program's design and specifications as well as in preventing and detecting implementation errors.

1. Introduction

The purpose of this paper is:

- to survey the ...

- What are the possible sources of failure in a program?
- What test data should be selected to demonstrate that failures do not arise from these sources?

... succeeds only when a program contains no errors. In this paper, one of our goals is to define the characteristics of an ideal test in a way that gives insight into problems of testing. We begin with some basic definitions.

Consider a program F whose input domain is the set of data D. F(d) denotes the result of executing F with input $d \in D$. OUT(d, F(d)) specifies the output requirement for F, i.e., OUT(d, F(d)) is true if and only if F(d) is an acceptable result. We will write OK(d) as an abbreviation for OUT(d, F(d)). Let T be a subset of D. T constitutes an ideal test if OK(t) for all $t \in T$ implies OK(d) for all $d \in D$, i.e., if from successful execution of a sample of the ...

Test completeness

Many programs have manually written or auto-generated test suites

A test suite ***T*** is ***complete*** *with respect to the type of an expression* ***e*** if execution of ***T*** covers all possible types ***e*** may have at runtime

Example of test completeness

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

a single execution of this piece of code suffices to cover all possible types **x** may have at the call site

Deciding test completeness

How can we (conservatively) decide whether a given test suite *T* is complete with respect to the type of an expression *e*?

A hybrid approach

1) execute program test suite

2) lightweight static **dependence** analysis

3) lightweight static **type** analysis

type safety facts

4) test completeness analysis

test completeness facts

1) Execution of test suite

Simply observe which values and types appear at each expression…

(generally an *under*-approximation of which values and types may appear in *any* execution)

2) Static dependence analysis

```
class A {
  m() { ... }
}
class B {}

f(v) {
  var t = 42;
  var x = g(t,v);
  x.m();
}
```

```
g(a,b) {
  var r;
  ...
  if (a*a > 100) {
    r = new A();
  } else {
    r = new B();
  }
  return r;
}
```

an overloaded function, the **type** of `r` depends (only) on the **value** of `a`

the **type** of `x` depends on the **value** of `t`, which depends on nothing (it's a constant)

- Over-approximates value <u>and type</u> dependencies
  (considers both *data* and *control* dependence)
- **Lightweight** analysis: context- and path-insensitive

3) Static type analysis

```
bar(p) {
  var y;
  if (p) {
    y = 3;
  } else {
    y = "hello";
  }
  if (p) {
    print(y + 6);
  } else {
    print(y.length);
  }
}
```

(example from An et al. , POPL 2011)

from calls, **p** is always true or false

how to prove type safety here?

1) path-sensitive static analysis
2) cover all paths [An et al., POPL 2011]
3) cover all values of **p**, exploiting lightweight static analyses:
   – the type of **y** depends only on the value of **p**

- Flow analysis to over-approximate types/values
  - also used to infer call graph for the dependence analysis
- **Lightweight** analysis: context- and path-insensitive

4) Test completeness analysis

Two ways to show that a test suite *T*
is complete for the type of an expression *e*:

- *T* has covered all the possible types/values of *e* (according to the static type analysis)
- *T* is complete for all dependencies of *e* (according to the static dependence analysis) ← recursive

Combine these rules into a proof system…

Boosting precision using *type filters*

1) execute program test suite

2) lightweight static **dependence** analysis

3) lightweight static **type** analysis

type safety facts

4) test completeness analysis

test completeness facts

Type filtering in action

```
class A {
  m() { ... }
}
class B {}

f(v) {
  var t = 42;
  var x = g(t,v);
  x.m();
}
```

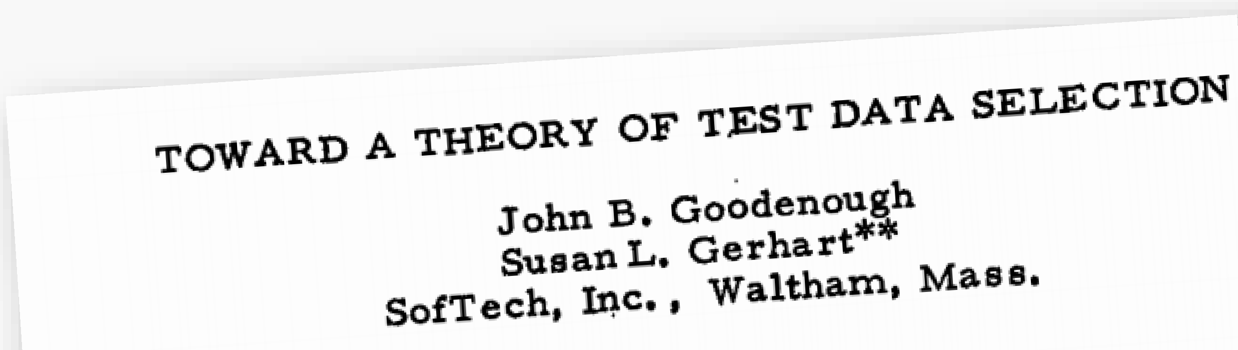
```
g(a,b) {
  var r;
  ...
  if (a*a > 100) {
    r = new A();
  } else {
    r = new B();
  }
  return r;
}
```

- First run of the type analysis infers that `x` has type `A` or `B`
- Second run can filter away `B`
  and thereby prove type safety for `x.m()`

Implementation: Goodenough

- finds out whether your test suite is *good enough*
- for the Dart language
  (developed by Google and ecma INTERNATIONAL)

- tested on 27 programs with test suites



TOWARD A THEORY OF TEST DATA SELECTION

John B. Goodenough
Susan L. Gerhart**
SofTech, Inc., Waltham, Mass.

Experiments

Research questions:

Q1) *To what extent can this technique show* ***test completeness*** *for realistic programs and test suites?*

Q2) *How important are the* ***test suites*** *for showing absence of runtime type errors?*

Q3) *How important is the* ***dependence analysis****?*

Q4) *In situations where test completeness is* not *shown, is the reason typically inadequate test coverage or inadequate precision of the static analysis components?*

Experiments

Research questions:

Q1) *To what extent can this technique show **test completeness** for realistic programs and test suites?*

Q2) *How* … *absence of ru*…

Q3) *How* …

Q4) *In sit*… *own, is th*… *e or inadequate precision of the static analysis components?*

For (at least) 81% of the expressions, all types that can possibly appear at runtime are observed by execution of the test suite

Experiments

Research questions:

Q1) *To what extent can this technique show* ***test completeness*** *for realistic programs and test suites?*

Q2) *How important are the* ***test suites*** *for showing absence of runtime type errors?*

Q3) *How i*

Q4) *In situ*
is the
or ina   *nts?*

Incorporating the test suites leads to improvements in 19 out of 27 benchmarks (in code with value-dependent types and branch correlations)

Experiments

Research q[...]

Q1) *To wha[...]*
for rea[...]

Q2) *How im[...]*
of runti[...]

Ability to prove absence of type errors and precision of inferred call graphs drops significantly if using a weaker dependence analysis

Q3) *How important is the **dependence analysis**?*

Q4) *In situations where test completeness is* not *shown,*
is the reason typically inadequate test coverage
or inadequate precision of the static analysis components?

Experiments

Research questions:

Q1) *To w* ... ***eness***
for ...

Q2) *How* ... *e*
of ru ...

Q3) *Hov* ...

Typical reasons:

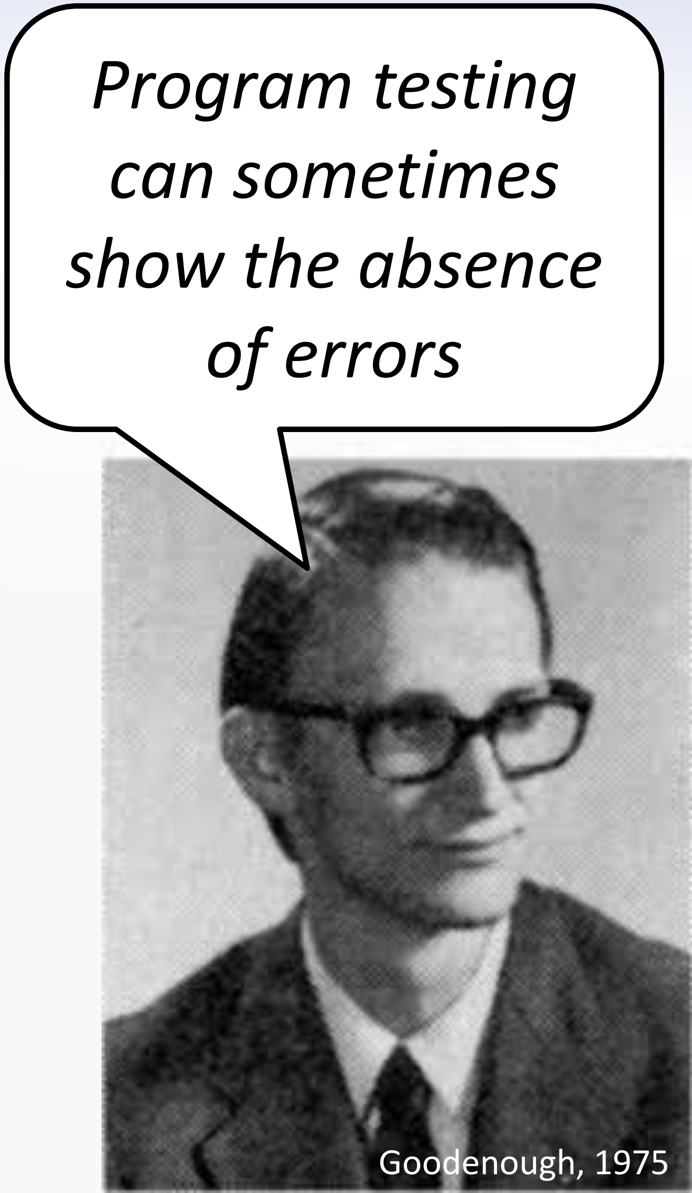
- inadequate test coverage
- imprecise heap modeling in dependence analysis

Q4) *In situations where test completeness is* not *shown,*
is the reason typically inadequate test coverage
or inadequate precision of the static analysis components?

Conclusion

- Hybrid static/dynamic analysis can show absence of type errors (and infer sound call graphs) in Dart code that is challenging for fully-static analysis

- Future work:
  - explore variations of the static analysis components
  - apply to program optimization, and to other languages
  - use test completeness as coverage metric for guiding test effort



Goodenough, 1975

Center for Advanced Software Analysis

http://cs.au.dk/CASA