

Formal Semantics for Testing

Colin Stebbins Gordon
csgordon@cs.washington.edu

University of Washington

Off the Beaten Track Workshop 2012



Why Care About Testing?

Testing is a huge portion of real development

- Developer:Tester ratio often $\approx 1:1$
- Underrepresented in top PL venues

PL researchers should care about testing!

- Testing \subseteq reasoning about program semantics
- Many testing tasks can be restated as *classic* PL problems
- Devs don't write formal specs; they write tests

Existing Testing Research

- Venues: ICSE, FSE, ISSTA, PASTE, ASE, some in OOPSLA, ...
- Good progress on important problems:
 - ▶ Test generation (many kinds!)
 - ▶ Test suite comparison
 - ▶ Test prioritization
 - ▶ many more!
- Many techniques, varying rigour
- Has limitations PL techniques handle well

Existing Testing Research

- Venues: ICSE, FSE, ISSTA, PASTE, ASE, some in OOPSLA, ...
- Good progress on important problems:
 - ▶ Test generation (many kinds!)
 - ▶ Test suite comparison
 - ▶ Test prioritization
 - ▶ many more!
- Many techniques, varying rigour
- Has limitations PL techniques handle well
 - ▶ Much pays limited attention to precise meanings

Existing Testing Research

- Venues: ICSE, FSE, ISSTA, PASTE, ASE, some in OOPSLA, ...
- Good progress on important problems:
 - ▶ Test generation (many kinds!)
 - ▶ Test suite comparison
 - ▶ Test prioritization
 - ▶ many more!
- Many techniques, varying rigour
- Has limitations PL techniques handle well
 - ▶ Much pays limited attention to precise meanings
 - ▶ Algorithms described informally

Existing Testing Research

- Venues: ICSE, FSE, ISSTA, PASTE, ASE, some in OOPSLA, ...
- Good progress on important problems:
 - ▶ Test generation (many kinds!)
 - ▶ Test suite comparison
 - ▶ Test prioritization
 - ▶ many more!
- Many techniques, varying rigour
- Has limitations PL techniques handle well
 - ▶ Much pays limited attention to precise meanings
 - ▶ Algorithms described informally
 - ▶ Limited understanding of different techniques' relative power

Existing Testing Research

- Venues: ICSE, FSE, ISSTA, PASTE, ASE, some in OOPSLA, ...
- Good progress on important problems:
 - ▶ Test generation (many kinds!)
 - ▶ Test suite comparison
 - ▶ Test prioritization
 - ▶ many more!
- Many techniques, varying rigour
- Has limitations PL techniques handle well
 - ▶ Much pays limited attention to precise meanings
 - ▶ Algorithms described informally
 - ▶ Limited understanding of different techniques' relative power

PL tools & techniques may improve or complement existing testing work

What's a Test?

Definition 1

An ad-hoc check that a program or subroutine produces an expected result.

What's a Test?

Definition 1

An ad-hoc check that a program or subroutine produces an expected result.

Definition 2

A partial specification of part of a program's behavior.

What's a Test?

Definition 1

An ad-hoc check that a program or subroutine produces an expected result.

Definition 2

A partial specification of part of a program's behavior.

My Definition

A specification of the *dynamic semantics* for a subset of a domain-specific (program-specific) language.

A natural test for an increment function might be:
`(assertEqual (add1 2) 3)`

A natural test for an increment function might be:

```
(assertEqual (add1 2) 3)
```

But what is left implicit here?

- The parameters to `assertEqual` are related by computation
- Written as a program, but is really a static assertion about *another program*

Tests Define Small Languages

For testing, `add1` is a language primitive:

- `add1` and its inputs/outputs form a small language
- A test suite specifies the *semantics for a small language*

Tests Define Small Languages

For testing, `add1` is a language primitive:

- `add1` and its inputs/outputs form a small language
- A test suite specifies the *semantics for a small language*

So instead of:

```
(assertEqual (add1 2) 3)
```

we get:

$$e ::= (\text{add1 } e) \mid v \quad v ::= 2 \mid 3 \quad \text{TEST1} \frac{}{(\text{add1 } 2) \Downarrow 3}$$

Tests Define Small Languages

For testing, `add1` is a language primitive:

- `add1` and its inputs/outputs form a small language
- A test suite specifies the *semantics for a small language*

So instead of:

```
(assertEqual (add1 2) 3)
```

we get:

$$e ::= (\text{add1 } e) \mid v \quad v ::= 2 \mid 3 \quad \text{TEST1} \frac{}{(\text{add1 } 2) \Downarrow 3}$$

The latter lets us directly leverage PL techniques.

Precision for testing tasks becomes easier.

Testing Tasks Made Precise

Restating testing tasks precisely as properties of semantics:

Testing Tasks Made Precise

Restating testing tasks precisely as properties of semantics:

- **Satisfying Specs:** Test-semantics and real-semantics are equivalent

Testing Tasks Made Precise

Restating testing tasks precisely as properties of semantics:

- **Satisfying Specs:** Test-semantics and real-semantics are equivalent
- **Inconsistent Tests:** Test-semantics lack a normal form

Restating testing tasks precisely as properties of semantics:

- **Satisfying Specs:** Test-semantics and real-semantics are equivalent
- **Inconsistent Tests:** Test-semantics lack a normal form
- **Redundant/Overlapping Tests:** Test-semantics are (internally) non-deterministic
 - ▶ e.g. admissible transitions, overlapping non-conflicting transitions

Testing Tasks Made Precise

Restating testing tasks precisely as properties of semantics:

- **Satisfying Specs:** Test-semantics and real-semantics are equivalent
- **Inconsistent Tests:** Test-semantics lack a normal form
- **Redundant/Overlapping Tests:** Test-semantics are (internally) non-deterministic
 - ▶ e.g. admissible transitions, overlapping non-conflicting transitions
- **Test Set Coverage:** Grammars for covered use cases

Restating testing tasks precisely as properties of semantics:

- **Satisfying Specs:** Test-semantics and real-semantics are equivalent
- **Inconsistent Tests:** Test-semantics lack a normal form
- **Redundant/Overlapping Tests:** Test-semantics are (internally) non-deterministic
 - ▶ e.g. admissible transitions, overlapping non-conflicting transitions
- **Test Set Coverage:** Grammars for covered use cases
- **Test Suite Comparison:** Comparing semantics

Restating testing tasks precisely as properties of semantics:

- **Satisfying Specs:** Test-semantics and real-semantics are equivalent
- **Inconsistent Tests:** Test-semantics lack a normal form
- **Redundant/Overlapping Tests:** Test-semantics are (internally) non-deterministic
 - ▶ e.g. admissible transitions, overlapping non-conflicting transitions
- **Test Set Coverage:** Grammars for covered use cases
- **Test Suite Comparison:** Comparing semantics

and likely more!

Automatic Test Generation

With more than one test:

$$\text{TEST1} \frac{}{(\text{add1 } 2) \Downarrow 3}$$

$$\text{TEST2} \frac{}{(\text{add1 } 3) \Downarrow 4}$$

Automatic Test Generation

With more than one test:

$$\text{TEST1} \frac{}{(\text{add1 } 2) \Downarrow 3} \quad \text{TEST2} \frac{}{(\text{add1 } 3) \Downarrow 4}$$

And a *composition rule*:

$$\text{COMPOSE} \frac{(f \ a) \Downarrow v_1 \quad (g \ v_1) \Downarrow v_2}{(g \ (f \ a)) \Downarrow v_2}$$

Automatic Test Generation

With more than one test:

$$\text{TEST1} \frac{}{(\text{add1 } 2) \Downarrow 3} \quad \text{TEST2} \frac{}{(\text{add1 } 3) \Downarrow 4}$$

And a *composition rule*:

$$\text{COMPOSE} \frac{(f \ a) \Downarrow v_1 \quad (g \ v_1) \Downarrow v_2}{(g \ (f \ a)) \Downarrow v_2}$$

Then we can *generate additional tests*, like

$$\text{GENERATEDTEST1} \frac{}{(\text{add1 } (\text{add1 } 2)) \Downarrow 4}$$

Automatic Test Generation

With more than one test:

$$\text{TEST1} \frac{}{(\text{add1 } 2) \Downarrow 3} \quad \text{TEST2} \frac{}{(\text{add1 } 3) \Downarrow 4}$$

And a *composition rule*:

$$\text{COMPOSE} \frac{(f \ a) \Downarrow v_1 \quad (g \ v_1) \Downarrow v_2}{(g \ (f \ a)) \Downarrow v_2}$$

Then we can *generate additional tests*, like

$$\text{GENERATEDTEST1} \frac{}{(\text{add1 } (\text{add1 } 2)) \Downarrow 4}$$

With semantics, we can *generate testing oracles*.

Understanding Prior Testing Research

Many test generation techniques are really different composition rules.

Standard Composition Rule in Test Generation Literature

$$\text{DOESNOTCRASH} \frac{(f \ a) \Downarrow v_1 \quad (g \ b) \Downarrow v_2 \quad R \neq \frac{1}{2}}{(g \ (f \ a)) \Downarrow R}$$

- Used to generate regression tests

Understanding Prior Testing Research

Many test generation techniques are really different composition rules.

Standard Composition Rule in Test Generation Literature

$$\text{DOESNOTCRASH} \frac{(f a) \Downarrow v_1 \quad (g b) \Downarrow v_2 \quad R \neq \frac{1}{2}}{(g (f a)) \Downarrow R}$$

- Used to generate regression tests
- This makes for very strange language semantics.

Understanding Prior Testing Research

Many test generation techniques are really different composition rules.

Standard Composition Rule in Test Generation Literature

$$\text{DOESNOTCRASH} \frac{(f a) \Downarrow v_1 \quad (g b) \Downarrow v_2 \quad R \neq \frac{1}{2}}{(g (f a)) \Downarrow R}$$

- Used to generate regression tests
- This makes for very strange language semantics.
- But this works reasonably well in practice!

Understanding Prior Testing Research

Many test generation techniques are really different composition rules.

Standard Composition Rule in Test Generation Literature

$$\text{DOESNOTCRASH} \frac{(f a) \Downarrow v_1 \quad (g b) \Downarrow v_2 \quad R \neq \frac{1}{2}}{(g (f a)) \Downarrow R}$$

- Used to generate regression tests
- This makes for very strange language semantics.
- But this works reasonably well in practice!

How much better can we do?

Using traditional PL techniques:

Using traditional PL techniques:

- We can prove that many composition rules are more precise than DOESNOTCRASH.

Using traditional PL techniques:

- We can prove that many composition rules are more precise than `DOESNOTCRASH`.
- We can prove type-agnostic test generation creates more type-incorrect tests than type-directed generation.

Using traditional PL techniques:

- We can prove that many composition rules are more precise than `DOESNOTCRASH`.
- We can prove type-agnostic test generation creates more type-incorrect tests than type-directed generation.
- We can prove type-directed test generation only creates well-typed tests
 - ▶ Klein, Flatt, and Findler.
Random Testing for Higher-Order, Stateful Programs. OOPSLA'10.

Using traditional PL techniques:

- We can prove that many composition rules are more precise than `DOESNOTCRASH`.
- We can prove type-agnostic test generation creates more type-incorrect tests than type-directed generation.
- We can prove type-directed test generation only creates well-typed tests
 - ▶ Klein, Flatt, and Findler.
Random Testing for Higher-Order, Stateful Programs. OOPSLA'10.
- Other propositions are sure to follow

Enabling New Testing Tasks

A language perspective on tests can give us new ways to:

- **Check Adequate Test Coverage:** Developer writes a grammar for the syntactic uses he believes are tested; tools can check this.

Enabling New Testing Tasks

A language perspective on tests can give us new ways to:

- **Check Adequate Test Coverage:** Developer writes a grammar for the syntactic uses he believes are tested; tools can check this.
 - ▶ A coverage metric related to *behavior!*

Enabling New Testing Tasks

A language perspective on tests can give us new ways to:

- **Check Adequate Test Coverage:** Developer writes a grammar for the syntactic uses he believes are tested; tools can check this.
 - ▶ A coverage metric related to *behavior!*
- **Suggest New Tests:** Using grammars as a coverage metric, possible grammar extensions/simplifications suggest expressions with no test

For verification, how different are these?

A Test

$$\text{TEST3} \frac{n : \text{Nat}}{(\text{add1 } n) \Downarrow n + 1}$$

A Dependent Type

$$\text{add1} : \prod n : \text{Nat} \rightarrow \{r : \text{Nat} \mid r = n + 1\}$$

For verification, how different are these?

A Test

$$\text{T}_{\text{EST}3} \frac{n : \text{Nat}}{(\text{add1 } n) \Downarrow n + 1}$$

A Dependent Type

$$\text{add1} : \prod n : \text{Nat} \rightarrow \{r : \text{Nat} \mid r = n + 1\}$$

Can we use the first to infer the second?

- i.e., *infer dependent type predicates* from test postconditions

- Tests are important
 - ▶ Huge portion of real development effort
- PL techniques have the potential to improve testing
 - ▶ Decades of PL tools and techniques can apply to testing, via semantics
 - ▶ A fresh lense to look at testing
 - ▶ New approaches may improve or complement existing work
- Tests can help address PL problems
 - ▶ Scale to real code
 - ▶ An often-ignored source of information