Formal Semantics for Testing [*]

Colin S. Gordon
University of Washington
csgordon@cs.washington.edu

Abstract

Software testing constitutes a substantial portion of the real world work of developing software. There has been much research aimed at partially-automating testing (random test generation, etc.). However, relatively little of it pays attention to precise semantics for languages or tests. Algorithms are described informally as pseudocode, and there is limited understanding of the relative power of many of the testing techniques in the literature. We outline a correspondence between tests and formal language semantics, and argue that techniques from programming language research are likely to advance our understanding of existing testing techniques and help to propose new techniques. We also argue that software tests can be a useful aid to formal verification.

1. Overview

Tests are often written on an ad-hoc basis, as short functions that first set up some initial state (precondition), then perform an operation (a test expression), and finally check that certain properties hold of the resulting value and state (a test oracle). For Scheme, consider:

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

But a test is more than an assertion that a certain value is produced. A test is in fact a partial specification of the semantics for some language. The simple test above can be considered to specify the semantics of the language:

$$e ::= (\mathsf{add1}\ 2) \mid v \qquad v ::= 2 \mid 3 \qquad \textsc{Test1}\ \frac{}{(\mathsf{add1}\ 2) \Downarrow 3}$$

With a richer set of such tests — as operational semantics — many testing-related properties and tasks take forms well-known in the language semantics literature. When generalized to classes of tests (for example, tests specifying function behavior over integers in a certain range) this approach edges close to defining contracts or dependent types, which have natural uses in program verification.

2. Traditional Testing Tasks Made Precise

A semantic view of testing allows us to use precise definitions for testing tasks and properties of test suites that are often only informally defined:

Satisfying Specifications A program satisfies the specification encoded in tests if and only if all expressions in the test-generated language produce the same value under both the test-defined semantics and normal execution semantics with the real implementation (possibly with some abstraction from concrete to abstract values to account for heap state, etc.). For parameterized tests, selecting concrete inputs is still necessary [5]; the formal semantics make the satisfaction problem well-defined, but does not improve upon the current approach of approximating specification satisfaction by trying carefully selected test inputs.

Inconsistent Tests A test set is inconsistent if the execution of some expression in the test language does not have a unique normal form (up to some equivalence class) in the test-induced semantics.

Redundant Tests A set of tests has redundancy if some expression in the test language can execute non-deterministically but produce consistent results. Techniques to find non-determinism in operational semantics can be applied to reduce test suite size.

Test Set Coverage A test set T1 is more expressive than a test set T2 if the language L1 defined by T1 contains the language L2 defined by T2 (L1⊇L2).

Test Generation With standard sequencing and composition rules, larger test languages and test cases can be generated automatically. For our increment example, adding another test and a composition rule:

$$\textsc{Test2}\ \frac{}{(\mathsf{add1}\ 3) \Downarrow 4} \qquad \textsc{Compose}\ \frac{(f\ a) \Downarrow v_1 \qquad (g\ v_1) \Downarrow v_2}{(g\ (f\ a)) \Downarrow v_2}$$

We could use a few template rules for sequencing and composition to generate the test

$$\textsc{GeneratedTest1}\ \frac{}{(\mathsf{add1}\ (\mathsf{add1}\ 2)) \Downarrow 4}$$

This test includes not only a new test expression, but also a new test oracle that results from the composition of other test oracles. In a variant with state, the composed oracle could combine the oracles from a sequence of operations. For example, the rule for sequencing in a stateful test language could combine the post-conditions of binary tree operations (e.g. inserting 2 then 3 yields a tree containing both 2 and 3).

[*] Off the Beaten Track Workshop, 2012

3. Understanding Prior Testing Research

Much work has already been devoted to completing the tasks in Section 2 using ad-hoc methodologies: a different formalization (if any) in every paper on test generation, oracle generation, etc. With the type of framework we propose, it becomes possible to compare the expressivity of existing work:

Test Generation Different test generation techniques manifest in our framework as different semantic rules for composition and sequencing, which in turn result in more or less expressive or precise semantics. Some intuitively obvious results that could be precisely stated and proven include things like:

- The "does not crash" oracle is less precise (will reveal fewer, or at least no more, bugs in the real semantics) than oracles checking specific predicates (like well-formedness of data structures) over results.
- Type-agnostic test generation schemes can generate tests prohibited by type-directed test generation (which would additionally require all terms to be well-typed).

Deeper results about classes of bugs a given test generation approach cannot expose are certainly possible as well.

4. Enabling New Testing Tasks

Checking Adequate Test Coverage Few testing tools give a way to verify that a test suite covers (specifies) all of the program behavior a tester intends. The language interpretation of tests offers a way to do so. A tool can compare a grammar for the expressions whose behavior should be specified by the test suite to the grammar covered by actual test cases. If the target coverage grammar permits something not covered by the test semantics, either the tester specified too broad a language, or at least one test case is missing. This ensures the test set specifies some semantics for every expression in expected usage, a measure missing from most testing approaches. This non-standard coverage metric is valuable because it precisely indicates what program behavior is addressed by a test suite with respect to what the tester believes has been addressed. This complements code-execution-coverage metrics, which approximate which parts of the code have been at least lightly checked.

Test Suggestion Using the test-defined language grammar as the coverage metric, it becomes straightforward to suggest additional test inputs based on grammar extensions.

5. Integrating Testing and Verification

As others have observed, there is some interplay between verification and testing [1]. Treating tests as formal semantics relates them even more strongly.

Incremental Verification Once tests are formally specified as semantics, they provide a stepping stone towards verification. Note the similarity between the test:

$$\textsc{Test3}\ \frac{n : \mathsf{Nat}}{(\mathtt{add1}\ n) \Downarrow n+1}$$

and one possible dependent (refinement) type for `add1`:

$$\mathtt{add1} : \Pi n : \mathsf{Nat} \to \{r : \mathsf{Nat} | r = n+1\}$$

The test above can be seen as a proposition that `add1` has the type above. This opens the door for mixing testing with formal verification. Some threshold for (concrete) test passage for some component may be treated as sufficient evidence to use a specification in verifying another component. This is useful for example, if a development team is willing to invest the energy to verify one module but not another, but still want to integrate some assurance that the unverified module meets its specification into the verification result.

Effective Dependent Type Inference Postconditions of tests suggest predicates for use when inferring dependent types. This offers an inherently incomplete, but possibly effective approach to inferring precise candidate types for expressions. The functions testers typically write to check important properties of results are typically pure, and generally good candidates for use as predicates in dependent types. This is similar to the use of specification functions in Boogie, Code Contracts, and Dafny. In combination with a well-annotated set of core libraries, it may be possible to verify that the implementation meets the specification from some such candidate type.

6. State and Higher-Order Tests

So far we have only given examples of first-order pure-functional programs. The literature on higher-order contracts [2] suggests semantics that would be useful for specifying higher-order tests. There is also a significant body of work on static representation of heaps and data structures that would be appropriate for use in this context (for example, alias types [4]). Some of that work is already in a setting appropriate to combine with an approach for enriching tests into dependent types [3]. Integrating tests with such heap models would allow composition rules to generate much more precise oracles for composite tests than existing test generation approaches, which typically consider the absence of a program crash to be a passing test.

References

[1] M. Barnett, M. Fähndrich, F. Logozzo, P. de Halleux, and N. Tillmann. Exploiting the Synergy between Automated-Test-Generation and Programming-by-Contract. In *ICSE Companion*, 2009.

[2] R. B. Findler and M. Felleisen. Contracts for Higher-order Functions. In *ICFP*, 2002.

[3] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent Types for Imperative Programs. In *ICFP*, 2008.

[4] F. Smith, D. Walker, and J. G. Morrisett. Alias Types. In *ESOP*, 2000.

[5] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *FSE*, 2005.