Formal verification of algorithms

Jeremy Avigad

Department of Philosophy and Department of Mathematical Sciences
Carnegie Mellon University
(currently visiting the INRIA-MSR Joint Research Centre in Orsay)

February, 2010

Formal verification

"Formal verification," in computer science, refers to the use of formal methods to prove correctness, for example, of:

- a circuit description, a program, or a network or security protocol; or
- a proof of a mathematical theorem.

Let's shift focus from "algorithms" (something abstract and vague) to "programs" (syntactic objects – texts – in a particular language).

What does it mean to say that a program is correct?

An example

Bowl1 ← butter
Cream(Bowl1)
Bowl1 ← whiteSugar, brownSugar
Do 2:
    Bowl1 ← egg
    Beat
Bowl1 ← Vanilla
Bowl2 ← Flour, BakingSoda, Salt
Mix(Bowl2)
Bowl1 ← Bowl2
Bowl1 ← ChocChips, Nuts
...

Fundamentals

Talk about "proving the correctness of a program" presupposes:

- A language to describe programs
- An understanding of what the programming language "means"
- A language to express specifications and properties
- An understanding of what the specification language "means"
- An understanding of the rules by which one can reason about programs and specifications.

Outline

- Models of computation
  - Turing machines
  - General recursion
  - The lambda calculus
- Styles of programming languages:
  - Imperative
  - Functional (typed, untyped)
- Specifications and specification languages
- Ways of providing semantics:
  - Operational
  - Denotational
  - Axiomatic

Models of computation

Traditional models of computation:

- Turing machines (Turing, 1937), Register machines
- General recursive functions (Gödel 1934, Kleene 1936)
- Lambda calculus (Church, 1936)
- Production systems, grammars, abacus computability, ...

Today:

- Imperative programming languages
- Functional programming languages
- Declarative programming languages

Turing machines

Start with: a set of symbols, states

Instructions of the form:

If in state i, scanning symbol q, do x, and go to state j

where "x" is move left, move right, or write q'.

Semantics: need to talk about halting computation sequences.

General recursive functions

These are a collection of partial functions on $\mathbb{N}$.

Start with basic functions: 0, $S(x)$, $P_i^n(x_1, \ldots, x_n) = x_i$.

Close under:

- Composition: $f(\vec{x}) \simeq g(h_1(\vec{x}), \ldots, h_k(\vec{x}))$
- Primitive recursion: $f(0, \vec{y}) \simeq g(\vec{y})$,
  $f(x+1, \vec{y}) = h(x, f(x, \vec{y}), \vec{y})$
- Unbounded search: $f(\vec{y}) \simeq \mu x\, g(x, \vec{y}) = 0$.

Semantics: these denote the corresponding mathematical objects.

Lambda calculus

The idea: $(\lambda x.x + 3)4 = 4 + 3 = 7$.

The pure lambda calculus: represent 3 as $\lambda xy.x(x(xy))$.

Examples: $S = \lambda uxy.x(uxy)$, $A = \lambda xy.xSy$

Theorem: normalization is confluent.

Semantics: normalize.

Imperative programming languages

```
class SelectionSortAlgorithm extends SortAlgorithm {
    void sort(int a[]) throws Exception {
        for (int i = 0; i < a.length; i++) {
            int min = i; int j;
            for (j = i + 1; j < a.length; j++) {
                if (a[j] < a[min]) {
                min = j;
                }
            }
            int T = a[min];
            a[min] = a[i];
            a[i] = T;
        }
    }
}
```

Functional programming languages

```
<<ssort.hs>>=
min1:: (a->a->Bool)->[a]->a
min1 (<=) [] = undefined
min1 (<=) [x] = x
min1 (<=) (x:xs)
  | x <= (min1 (<=) xs) = x
  | otherwise = min1 (<=) xs

<<ssort.hs>>=
delete:: (Eq a) => a->[a]->[a]
delete a [] = []
delete a (x:xs)
  | a==x = xs
  | otherwise = x:(delete a xs)

<<ssort.hs>>=
ssort:: (Eq a) => (a->a->Bool)->[a]->[a]
ssort (<=) [] = []
ssort (<=) xs = [x] ++ ssort (<=) (delete x xs)
  where x = min1 (<=) xs
```

Semantics

At the bare minimum: a program determines a function from input to output.

May also care about:

- what happens in between
- interactions with the environment

Three types of semantics:

- Operational semantics: describe behavior on an abstract machine
- Denotational semantics: assign a mathematical object as meaning
- Axiomatic semantics: state properties that are necessarily satisfied

Wrinkes: asynchronous behavior; nondeterminism; parallelism; hybrid systems

Specifications and specification languages

Types of specifications:

- Input / output. Examples:
  - Input: list of integers; output: sorted list.
  - Input: graph; output: yes/no (is it connected?)
  - Input: an integer; output: the factorial
  - Input: a real number, r (with some fixed precision?); output, $\ln r$.
- Avoidance of errors:
  - Never writes to memory locations out of range.
  - Always terminates / never freezes.
  - Always responds to user requests.

Specifications and specification languages

Writing specifications:

- Sometimes one uses ordinary mathematical language (formal or informal).
- Often one uses ordinary mathematical language, with additional operations and notations.
- Sometimes one uses *restricted* languages.
- Sometimes a proof a mathematical statement can be viewed as constructing an algorithm that meets a specification!

Two major approaches to verification:

- Model checking.
- Interactive theorem proving.

Operational semantics

Need to specify the notion of a machine "state."

Typically, one assumes that the state assigns values to memory "locations."

Parts of a program change the state. The rules for the semantics explain how.

Denotational semantics

Here, programs denote a mathematical objects in an appropriate "domain."

For example, one can assign to programs (partial) functions on the natural numbers.

The semantics is typically *compositional*: the denotation of a complex program is determined from the denotation of its parts.

One can model operational semantics in this way, using functions to act on the state.

Axiomatic semantics

In practice, this is the most useful.

Rather than assign a precise meaning, simply explain what properties a program is assumed to satisfy.

One typically uses "Hoare triples": $\{P\}\ S\ \{Q\}$.