Program verification

19 October 2016

Example revisited

```
// assume(n>2);
void partition(int a[], int n) {
  int pivot = a[0];
  int lo = 1, hi = n-1;
  while (lo <= hi) {
    while (lo < n && a[lo] <= pivot)
      lo++;
    while (a[hi] > pivot)
      hi--;
    if (lo < hi)
      swap(a,lo,hi);
  }
}
```

How can we reason about this program (fragment) ?

The beginnings of program verification

Goal: formalizing programming language semantics

Robert W. Floyd. Assigning Meanings to Programs (1967)

"*an adequate basis for formal definitions of the meanings of programs [...] in such a way that a rigorous standard is established for proofs*"

"*If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2.*"

Floyd: Assigning Meanings to Programs

Floyd's method: annotating a program (flowchart) with assertions

verification condition: a formula $V_c(P; Q)$ such that
if P is true before executing c, then Q is true on termination

strongest verifiable consequent (for a program + an initial condition)
= strongest property true after after program execution

Formulas/assertion: expressed in *first order logic* (predicate logic)

Floyd's work:
develops general rules for combining verification conditions
and specific rules to combine different instruction types
introduces *invariants* for reasoning about cycles
handles *termination* using a positive decreasing measure

The work of Hoare

C.A.R. Hoare. An Axiomatic Basis for Computer Programming (1969)
– works with program text, not flowcharts
– like Floyd, uses preconditions and postconditions for statements, but the *Hoare triple* notation better highlights the relation between statement and the two assertions

– Notation *partial correctness* $\{P\}\ S\ \{Q\}$
If S is executed in a state that satisfies P, and S terminates, the resulting state satisfies Q

– Similar statements for *total correctness* $[P]\ S\ [Q]$
If S is executed in a state that satisfies P, then S terminates and the resulting state satisfies Q

Rigorous example: C.A.R. Hoare. Proof of a Program: FIND (1971)

Hoare's rules (axioms)

Are defined for each individual statement
by combining them, we can reason about whole programs

Assignment: $\overline{\{Q[x/E]\}\ x := E\ \{Q\}}$ where $Q[x/E]$ substitutes E for x in Q

e.g.: $\{x = y - 2\}\ x := x + 2\ \{x = y\}$ (in the result, $x = y$, we substitute x with the assigned expression, $x + 2$ and get $x + 2 = y$, so $x = y - 2$)
Note: the "backwards" writing (P as a function of Q) simplifies the rule

Sequencing: $$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Decision: $$\frac{\{P \wedge E\}\ S_1\ \{Q\} \qquad \{P \wedge \neg E\}\ S_2\ \{Q\}}{\{P\}\ \text{if } E \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

Hoare's rules (cont.)

Loop (with initial test): is key in reasoning about programs
– we must find an *invariant* I = a property preserved by every execution of the cycle (true each time between iterations)
– if cycle is entered (E), invariant is maintained after one iteration S
– if cycle is not entered ($\neg E$), invariant implies postcondition Q

Hoare rule for `while`

$$\frac{\{I \wedge E\}\ S\ \{I\} \qquad I \wedge \neg E \Rightarrow Q}{\{I\}\ \text{while } E \text{ do } S\ \{Q\}}$$

Example of applying Hoare rules

Find `n` knowing it's initially between `lo` and `hi`:

```
while (lo < hi) {    // binary search; I: lo <= n && n <= hi
  m = (lo + hi) / 2;
  if (n > m)         // both cases maintain lo<=n && n<=hi
    lo = m+1;        // n > m => n >= m+1 => n >= lo
  else hi = m;       // !(n > m) => n <= m => n <= hi
}                    // I stays true
// lo<=n && n<=hi && !(lo<hi) => lo==n && n==hi
assert(n == lo && n == hi);
```

Hoare rules with pointers (aliasing)

Consider $\{P\}\ *x = 2\ \{v + *x = 4\}$
What is the precondition P ? Right answer: $v = 2 \lor x = \&v$.
But applying assignment rule $(v + *x = 4)[*x/2]$ loses the second case

We must model memory. m = memory, a = address, d = data
Consider the functions $rd(m, a)$ return d and $wr(m, a, d)$ return m'
Rule: $rd(wr(m, a_1, d), a_2) = \begin{cases} d & \text{if } a_2 = a_1 \\ rd(m, a_2) & \text{if } a_2 \neq a_1 \end{cases}$
We must derive a property of memory m from the relation:
$rd(wr(m, x, 2), \&v) + rd(wr(m, x, 2), x) = 4$
$rd(wr(m, x, 2), \&v) + 2 = 4$
$rd(wr(m, x, 2), \&v) = 2$
$x = \&v \land 2 = 2 \lor x \neq \&v \land rd(m, \&v) = 2$
$x = \&v \lor v = 2$

Dijkstra's *weakest precondition* operator

E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs (1975)
– for a statement S and given postcondition Q there can be several preconditions P such that $\{P\}\ S\ \{Q\}$ or $[P]\ S\ [Q]$.
– Dijkstra establishes a *necessary and sufficient* precondition $wp(S, Q)$ for successful termination of S with postcondition Q.
– necessary (*weakest*): if $[P]\ S\ [Q]$ then $P \Rightarrow\ wp(S, Q)$
– *wp* is a *predicate transformer* (transforms post- into precondition) precondiie)
– allows defining a *calculus* with such transformations

Dijkstra's preconditions (cont.)

Assignment: $wp(x := E, Q) = Q[x/E]$ (see Hoare's rule)
Sequencing: $wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$
Decision:
$wp(\text{if } E \text{ then } S_1 \text{ else } S_2, Q) = (E \Rightarrow wp(S_1, Q)) \wedge (\neg E \Rightarrow wp(S_2, Q))$

For loops, we need a recurrent computation
Define wp_k, assuming loop finishes in at most k iteration:
$wp_0(\text{while } E \text{ do } S, Q) = \neg E \Rightarrow Q$ (loop not entered)

$$wp_{k+1}(\text{while}\, E\, \text{do}\, S, Q)) = (E \Rightarrow wp(S, wp_k(\text{while}\, E\, \text{do}\, S, Q))) \\ \wedge(\neg E \Rightarrow Q)$$

($\leq k+1$ iterations $\Leftrightarrow$ one iteration followed by $\leq k$, or no iteration; equivalent with decomposing the first `while` into an `if`)
$\Rightarrow$ can be written as a fixpoint formula

Recap: verification by theorem proving

1. Write Hoare triples / Dijkstra's preconditions
2. Check the chain of implications (with a decision procedure / theorem prover) Examples:

with Hoare's sequencing rule
check $Pre \Rightarrow wp(Prog, Post)$
check $I \wedge E \Rightarrow wp(LoopBody, I)$ for loops