Program Verification with
Hoare Logic

Anders Møller

University of Aarhus

BRICS

`http://www.brics.dk/~amoeller/talks/hoare.pdf`

Using Assertions in Programming

- **Assertion**: invariant at specific program point
- dynamic checks, runtime errors
  (e.g. Java 1.4 `assert(exp)`)
- Floyd, 1967:
  - use assertions as foundation for **static correctness proofs**
  - specify assertions at *every* program point
  - correctness reduced to **reasoning about individual statements**

Hoare Logic

Hoare, 1969: use Floyd’s ideas to define **axiomatic semantics**
(i.e., define the programming language semantics as a **proof system**)

program statement

Hoare triple: {P} S {Q}

precondition postcondition - using some predicate logic

- ***partial correctness***: if **S** is executed in a store initially satisfying **P** and it terminates, then the final store satisfies **Q**
- ***total correctness***: as partial, but also requires termination
- (we ignore termination and definedness...)

Hoare Logic for miniTIP

miniTIP: as TIP, but without

- functions
- pointers
- input/output

i.e., a core while-language with only *pure* expressions

An Axiom for Assignment

$$\frac{}{\{Q[E/id]\}\ \ id=E;\ \{Q\}}$$

Example:

{y+7>42} x=y+7; {x>42}

- *the most central aspect of imperative languages is reduced to simple syntactic formula substitution!*
- *this axiom is "backwards" - it allows the precondition to be inferred automatically from the statement and the postcondition*

A Proof Rule for Sequence

$$\frac{\{P\}\ S_1\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1\ S_2\ \{Q\}}$$

(Apparently) R *must be created manually...*

A Proof Rule for Conditional

$$\frac{\{P \wedge E\}\ S_1\ \{Q\} \qquad \{P \wedge \neg E\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if (E) \{}S_1\texttt{\} else \{}S_2\texttt{\}}\ \{Q\}}$$

A Proof Rule for Iteration

$$\frac{\{P \wedge E\}\ \mathtt{S}\ \{P\}}{\{P\}\ \mathtt{while\ (E)\ \{S\}}\ \{P \wedge \neg E\}}$$

- P *is the loop invariant - this is where the main difficulty is!*
- *This rule can be extended to handle total correctness...*

Pre-Strengthening and Post-Weakening

$$\frac{P \Rightarrow P' \qquad \{P'\}\ S\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Intuitively, $A \Rightarrow B$ *means that* A *is* ***stronger*** *than* B

Soundness and Completeness

- **Soundness**: if {P} S {Q} can be proven, then it is certain that executing S from a store satisfying P will only terminate in stores satisfying Q

- **Completeness**: the converse of soundness

- Hoare logic is both sound and complete, *provided that the underlying logic is!*
- often, the underlying logic is sound but *incomplete* (e.g. Peano arithmetic)

Example: `factorial`

a logical variable, remembers the initial value

$\{n \geq 0 \wedge t = n\}$

```
r=1;   {P1}
while (n≠0){P2} {
  r=r*n;
                {P3}
  n=n-1;
}
```

$\{r = t!\}$

$P_1 \equiv n \geq 0 \wedge t = n \wedge r = 1$

$P_2 \equiv r = t!/n! \wedge t \geq n \geq 0$

$P_3 \equiv r = t!/(n-1)! \wedge t \geq n > 0$

- Peano arithmetic can be used in the assertions

Proof Obligations in the Example

- $\{n \geq 0 \wedge t=n\}$ `r=1;` $\{P_1\}$
- $P_1 \Rightarrow P_2$
- $\{P_2 \wedge n \neq 0\}$ `r=r*n;` $\{P_3\}$
- $\{P_3\}$ `n=n-1;` $\{P_2\}$
- $(P_2 \wedge \neg(n \neq 0)) \Rightarrow r=t!$

Hoare Logic for the full TIP language?

- Input/Output expressions?
  - just convert to separate statements

- Functions?
  - require pre/post-conditions at function declaration
  - the **frame** problem: to be useful, the pre/post-conditions also need to specify which things do *not* change

- Pointers?
  - the heap-as-array trick: model *x=y as H[x]=y
  - the **global reasoning** problem: in the proofs, each heap write appears to affect *every* heap read

Dijkstra's Weakest Precondition Technique

Dijkstra, 1975:

Given a statement **S** and a postcondition **Q**, the **weakest precondition** **WP(S,Q)** denotes the **largest set of stores** for which **S** terminates and the resulting store satisfies **Q.**

- WP(`id=E;`, Q) = Q[E/id]
- WP(S_1 S_2, Q) = WP(S_1,WP(S_2,Q))
- WP(`if (E) {`S_1`} else {`S_2 `}`, Q) = $E \Rightarrow WP(S_1,Q) \land \neg E \Rightarrow WP(S_2,Q)$
- WP(`while (E) {S}`, Q) = $\exists k \geq 0: H_k$ where
  $H_0 = \neg E \land Q$
  $H_{k+1} = H_0 \lor WP(S, H_k)$

this shows that the intermediate assertion comes for free in the sequence rule in Hoare Logic

inductive definition, calls for inductive proofs

Strongest Postcondition

- WP is a **backward** *predicate transformer*
- **SP** (strongest postcondition) is **forward**:

  SP(P, id=E;) = ∃v: P[v/id] ∧ id=E[v/id]

  ...

{P} S {Q} iff P⇒WP(S,Q) iff SP(P,S)⇒Q

(if using the total correctness variant)

The Pointer Assertion Logic Engine

- **PALE**: a tool for verifying pointer intensive programs, e.g., datatype operations
  - no memory leaks or dangling pointers
  - no null pointer dereferences
  - datatype invariants preserved
- Uses **M2L-Tree** (Monadic 2nd-order Logic on finite Trees)
  - a **decidable** but very expressive logic
  - MONA: a decision procedure based on tree automata
  - suitable for modeling many heap structures
    - heap ~ universe
    - pointer variable x ~ unary predicate x(p)
    - pointer field f ~ binary predicate f(p,q)

Example: Red-Black Search Trees

A **red-black tree** is

1. a binary tree whose nodes are red or black and have parent pointers
2. a red node cannot have a red successor
3. the root is black
4. the number of black nodes is the same for all direct paths from the root to a leaf

Goal: verify correctness of the `insert` procedure

Example: `red_black_insert.pale`

```
proc redblackinsert(data t,root:Node):Node
{ pointer y,x:Node;
  x = t;
  root = treeinsert(x,root);
  x.color = false;
  while (x!=root & x.p.color=false) {
    if (x.p=x.p.p.left) {
      y = x.p.p.right;
      if (y!=null & y.color=false) {
        x.p.color = true;
        y.color = true;
        x.p.p.color = false;
        x = x.p.p;
      }
      else {
        if (x=x.p.right) {
          x = x.p;
          root = leftrotate(x,root);
        }
        x.p.color = true;
        x.p.p.color = false;
        root = rightrotate(x.p.p,root);
        root.color = true;
    }}
```

```
    else {
      y = x.p.p.left;
      if (y!=null & y.color=false) {
        x.p.color = true;
        y.color = true;
        x.p.p.color = false;
        x = x.p.p;
      }
      else {
      if (x=x.p.left) {
        x = x.p;
        root = rightrotate(x,root);
      }
      x.p.color = true;
      x.p.p.color = false;
      root = leftrotate(x.p.p,root);
      root.color = true;
  }}
  root.color = true;
  return root;
}
```

\+ auxiliary procedures `leftrotate`, `rightrotate`, and `treeinsert` (total ~135 lines of program code)

Using Hoare Logic in PALE

1. Require **invariants** at all while-loops and procedure calls (extra assertions are allowed)

2. Split the program into **Hoare triples**: {P} S {Q}

3. Verify each triple separately (only loop/call-free code left)
   - including check for null-pointer dereferences and other memory errors

Note: highly modular, no fixed-point iteration, but requires invariants!

Verifying the Hoare Triples

Reduce everything to M2L-Tree and use the MONA tool.

Use ***transductions*** to encode loop-free code:

- ***Store predicates*** (for program variables and record fields) model the store at each program point
- ***Predicate transformation*** models the semantics of statements
  Example: `x = y.next;` $\rightarrow$ $x'(p) = \exists q.\ y(q) \wedge next(q,p)$
- ***Verification condition*** is constructed by expressing the pre- and post-condition using store predicates from end points

– Looks like an interpreter, but is essentially Weakest Precondition

– Sound *and complete* for individual Hoare triples!

Example: Red-Black Search Trees

1. Insert **invariants** and **pre- and post-conditions**, expressing **correctness requirements** for `red_black_insert` and the auxiliary procedures
2. Run the **PALE** tool

Result: after 9000 tree automaton operations and 50 seconds, PALE replies that

- **all assertions are valid**
- there can be **no memory-related errors**

If verification fails, a **counterexample** is returned!

PALE Experiments

Benchmark	Lines of code	Invariants	Time (sec.)
`reverse`	16	1	0.52
`search`	12	1	0.25
`zip`	33	1	4.58
`delete`	22	0	1.36
`insert`	33	0	2.66
`rotate`	11	0	0.22
`concat`	24	0	0.47
`bubblesort_simple`	43	1	2.86
`bubblesort_boolean`	43	2	3.37
`bubblesort_full`	43	2	4.13
`orderedreverse`	24	1	0.46
`recreverse`	15	2	0.34
`doublylinked`	72	1	9.43
`leftrotate`	30	0	4.62
`rightrotate`	30	0	4.68
`treeinsert`	36	1	8.27
`redblackinsert`	57	7	35.04
`threaded`	54	4	3.38

References

- *An Axiomatic Basis for Computer Programming*
  C.A.R. Hoare, CACM 12(10), 1969

- *The Science of Programming*
  D. Gries, Springer-Verlag, 1981

- *The Pointer Assertion Logic Engine*
  A. Møller and M.I. Schwartzbach, PLDI 2001