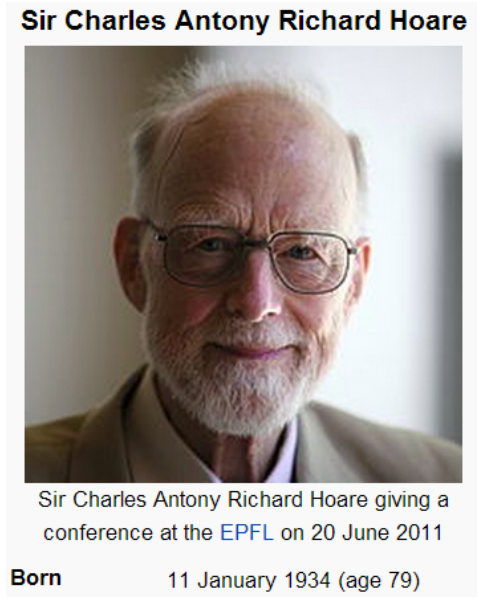
Lecturecise 10
Hoare Logic continued

2013

Hoare Triple and Friends



$P, Q \subseteq S \qquad r \subseteq S \times S$

Hoare Triple

$$\{P\}\ r\ \{Q\} \iff \forall s, s' \in S.\ \big(s \in P \wedge (s, s') \in r \rightarrow s' \in Q\big)$$

Strongest postcondition:

$$sp(P, r) = \{s' \mid \exists s.\ s \in P \wedge (s, s') \in r\}$$

Weakest precondition:

$$wp(r, Q) = \{s \mid \forall s'.(s, s') \in r \rightarrow s' \in Q\}$$

Exercise: Prove wp Distributivity

$$wp(r, Q) = \{s \mid \forall s'.(s, s') \in r \rightarrow s' \in Q\}$$

$wp(r_1 \cup r_2, Q) =$

Rules for WP

Rules for Computing Weakest Preconditions

We derive the rules below from the definition of weakest precondition on sets and relations

$$wp(r, Q) = \{s \mid \forall s'.(s, s') \in r \rightarrow s' \in Q\}$$

Assume Statement

Suppose we have one variable x, and identify the state with that variable. Note that $\rho(assume(F)) = \Delta_{F_s}$. By definition

$$\begin{aligned} wp(\Delta_{F_s}, Q_s) &= \{x \mid \forall x'.(x, x') \in \Delta_{F_s} \rightarrow x' \in Q_s\} \\ &= \{x \mid \forall x'.(x \in F_s \wedge x = x') \rightarrow x' \in Q_s\} \\ &= \{x \mid x \in F_s \rightarrow x \in Q_s\} = \{x \mid F \rightarrow Q\} \end{aligned}$$

Changing from sets to formulas, we obtain the rule for *wp* on formulas:

$$wp_F(\text{assume(F)}, Q) = (F \rightarrow Q)$$

Rules for Computing Weakest Preconditions

Assignment Statement

Consider the case of two variables. Recall that the relation associated with the assignment $x = e$ is

$$x' = e \wedge y' = y$$

Then we have, for formula Q containing x and y:

$$\begin{aligned} wp(\rho(x = e), \{(x, y) \mid Q\}) &= \{(x, y) \mid \forall x'.\forall y'.\ x' = e \wedge y' = y \rightarrow \\ &\qquad Q[x := x', y := y']\} \\ &= \{(x, y) \mid Q[x := e]\} \end{aligned}$$

From here we obtain a justification to define:

$$wp_F(x = e, Q) = Q[x := e]$$

Rules for Computing Weakest Preconditions

Havoc Statement

$$wp_F(\text{havoc(x)}, Q) = \forall x.Q$$

Sequential Composition

$$wp(r_1 \circ r_2, Q_s) = wp(r_1, wp(r_2, Q_s))$$

Same for formulas:

$$wp_F(c_1 \ ; \ c_2, Q) = wp_F(c_1, wp_F(c_2, Q))$$

Nondeterministic Choice (Branches)

In terms of sets and relations

$$wp(r_1 \cup r_2, Q_s) = wp(r_1, Q_s) \cap wp(r_2, Q_s)$$

In terms of formulas

$$wp_F(c_1 [] c_2, Q) = wp_F(c_1, Q) \wedge wp_F(c_2, Q)$$

Summary of Weakest Precondition Rules

c	$wp(c, Q)$
$x = e$	$Q[x := e]$
$havoc(x)$	$\forall x.Q$
$assume(F)$	$F \rightarrow Q$
$c_1 \,[\!]\, c_2$	$wp(c_1, Q) \wedge wp(c_2, Q)$
$c_1; c_2$	$wp(c_1, wp(c_2, Q))$

Size of Generated Verification Conditions

Because of the rule

$$wp_F(c_1 [] c_2, Q) = wp_F(c_1, Q) \wedge wp_F(c_2, Q)$$

which duplicates Q, the size can be exponential.

$wp_F((c_1 \,[\!]\, c_2); (c_3 \,[\!]\, c_4), Q) =$

Avoiding Exponential Blowup

Propose an algorithm that, given an arbitrary program c and a formula Q, computes in polynomial time formula equivalent to $wp_F(c, Q)$

Syntactic Rules for Hoare Logic

Summary of Proof Rules

We next present (one possible) summary of proof rules for Hoare logic.

Weakening and Strengthening

Strengthening precondition:

$$\frac{\models P_1 \to P_2 \quad \{P_2\}c\{Q\}}{\{P_1\}c\{Q\}}$$

Weakening postcondition:

$$\frac{\{P\}c\{Q_1\} \quad \models Q_1 \to Q_2}{\{P\}c\{Q_2\}}$$

Loop Free Blocks

We can directly use the rules we derived for basic loop-free code. Either through weakest preconditions or strongest postconditions.

$$\{wp(c, Q)\}c\{Q\}$$

or,

$$\{P\}c\{sp(P, c)\}$$

For example, we have:

$$\{Q[x := e]\}\ (x = e)\ \{Q\}$$

$$\{\forall x.Q\}\ havoc(x)\ \{Q\}$$

$$\{(F \rightarrow Q)\}\ assume(F)\ \{Q\}$$

$$\{P\}\ assume(F)\ \{P \wedge F\}$$

Rules continued

Loops

$$\frac{\{I\}c\{I\}}{\{I\}\ while(*)c\{I\}}$$

Sequential Composition

$$\frac{\{P\}\,c_1\{Q\} \quad \{Q\}\,c_2\,\{R\}}{\{P\}\ c_1; c_2\ \{R\}}$$

Non-Deterministic Choice

$$\frac{\{P\}c_1\{Q\} \quad \{P\}c_2\{Q\}}{\{P\}c_1[]c_2\{Q\}}$$

While Loops

Knowing that the while loop: **while** (F) c;

is equivalent to: **while**$(*)\{$assume(F); c$\}$;
assume($\neg$ F);

Question: What is the rule for while loops?
Hint

$$\frac{(\models P \rightarrow ?);\ \{?\}c\{?\};\ (\models ? \rightarrow Q)}{\{P\}\ \mathit{while}\{I\}(F)(c)\ \{Q\}}$$

While Loops

Knowing that the while loop: **while** (F) c;

is equivalent to: **while**$(*)\{$assume(F); c$\}$;
assume($\neg$ F);

Question: What is the rule for while loops?
Hint

$$\frac{(\models P \rightarrow ?);\ \{?\}c\{?\};\ (\models ? \rightarrow Q)}{\{P\}\ while\{I\}(F)(c)\ \{Q\}}$$

It follows that the rule for while loops is:

$$\frac{(\models P \rightarrow I);\ \{I \wedge F\}c\{I\};\ (\models (I \wedge \neg F) \rightarrow Q))}{\{P\}\ while\{I\}(F)(c)\ \{Q\}}$$

Applying Proof Rules given Invariants

Let us treat $\{P\}$ as a new kind of statement, written

assert(P)

For the moment the purpose of assert is just to indicate preconditions and postconditions. When we write

assert(P)
c1;
assert(Q)
c2;
assert(R)

we expect that these Hoare triples hold:

$$\{P\}c1\{Q\}$$
$$\{Q\}c2\{R\}$$

Sufficiently annotated program

Consider the control-flow graph of a program with statements assert, assume, x=e and with graph edges expressing "[]" and ";".
We will say that the program c is *sufficiently annotated* iff

- the first statement is assert(Pre)
- the last statement is assert(Post)
- every cycle in the control-flow graph contains at least one assert

Assertion path

An assertion path is a path in its control-flow graph that starts and ends with *assert*. Given the assertion path

assert(P)
c1
...
cK
assert(Q)

we omit any assert statements in the middle, obtaining from c1,...,cK statements d1,...,dL. We call

$$\{P\}d1\ ;\ \ldots\ ;\ dL\{Q\}$$

the Hoare triple of the assertion path.

Proving Hoare triple for entire program

A *basic path* is an assertion path that contains no *assert* commands other than those at the beginning and end. Each sufficiently annotated program has finitely many basic paths.

Theorem: If the Hoare triple for each basic path is valid, then the Hoare triple $\{Pre\}c\{Post\}$ is valid.
Proof: If each basic path is valid, then each path is valid, by induction and Hoare logic rule for sequential composition. Each program is union of (potentially infinitely many) paths, so the property holds for the entire program. (Another explanation: consider any given execution and corresponding path in the control-flow graph. By induction on the length of the path we prove that all assert statements hold, up to the last one.)

Verification recipe

The verification condition of a basic path is the formula whose validity expresses the validity of the Hoare triple for this path.
Simple verification conditions for a sufficiently annotated program is the set of verification conditions for each each basic path of the program.
One approach to verification condition generation is therefore:

- start with sufficiently annotated program
- generate simple verification conditions
- prove each of the simple verification conditions

In a program of size n, what is the bound on the number of basic paths?

Verification recipe

The verification condition of a basic path is the formula whose validity expresses the validity of the Hoare triple for this path.
Simple verification conditions for a sufficiently annotated program is the set of verification conditions for each each basic path of the program.
One approach to verification condition generation is therefore:

- start with sufficiently annotated program
- generate simple verification conditions
- prove each of the simple verification conditions

In a program of size n, what is the bound on the number of basic paths?
It can be $2^{O(n)}$.

Handling the path explosion

In a program of size n, the number of basic paths can be $2^{O(n)}$.
Remedies:

- require more annotations (e.g. at each merge point)
- extreme case: assertion on each CFG vertex - this gives classical Hoare logic proof
- merge subgraphs without annotations: perform sequential composition and disjunction of formulas on edges
- generate correctness formulas for multiple paths in an acyclic subgraph at once, using propositional variables to encode the existence of paths

Exercise

Give a complete Hoare logic proof for the following program:

$\{\mathsf{n} >= 0 \;\&\&\; \mathsf{d} > 0\}$

```
    q = 0
    r = n
    while ( r >= d ) {
        q = q + 1
        r = r - d
    }
```

$\{\mathsf{n} == \mathsf{q} * \mathsf{d} + \mathsf{r} \;\&\&\; 0 <= \mathsf{r} < \mathsf{d}\}$

The proof should be step-by-step as in the example proof in the lecture on Hoare Logic. To prove each step you can use the syntactic rules for Hoare Logic.

Exercise

```
// {n >= 0 && d > 0}
q = 0
// {n >= 0 && d > 0 && q == 0}
r = n
// {n >= 0 && d > 0 && q == 0 && r == n}
while // {d > 0 && n == q * d + r && 0 <= r}
   (r >= d) {
// {d > 0 && n == q * d + r && d <= r}
    q = q + 1
// {d > 0 && n == (q-1) * d + r && d <= r}
    r = r - d
// {d > 0 && n == (q-1) * d + r + d && 0 <= r}
// {d > 0 && n == q * d + r && 0 <= r}
}
// {d > 0 && n == q * d + r && 0 <= r && r < d}
// {n == q * d + r && 0 <= r < d}
```

What can be omitted to still have sufficiently annotated program?

...back to Algebraic Data Types

Unification Algorithm

A set of equations is in *solved form* if it is of the form $\{x_1 \doteq t_1, \ldots, x_n \doteq t_n\}$ where variables x_i do not appear in terms t_j, that is $\{x_1, \ldots, x_n\} \cap (FV(t_1) \cup \ldots FV(t_n)) = \emptyset$
We obtain a solved form in finite time using the algorithm that applies the following rules in any order as long as no clash is reported and as long as the equations are not in solved form.

- **Orient**: Select $t \doteq x$ where t is not x, and replace it with $x \doteq t$.
- **Delete**: Select $x \doteq x$, remove it.
- **Eliminate**: Given $x \doteq t$ where x does not occur in t, substitute x with t in all remaining equations.
- **Occurs Check**: Given $x \doteq t$ where x occurs in t, report clash.
- **Decomposition**: Given $f(t_1, \ldots, t_n) \doteq f(s_1, \ldots, s_n)$, replace it with $t_1 \doteq s_1, \ldots, t_n \doteq s_n$.
- **Clash**: Given $f(t_1, \ldots, t_n) \doteq g(s_1, \ldots, s_m)$ for f not g, report clash

Run Unification Algorithm

$\Sigma = \{h, f, a, b\}$ with arities $2, 2, 0, 0$

$$h(x, f(x, y)) = h(f(a, v), f(f(u, b), f(u, u)))$$

$$h(x, f(x, x)) = h(f(a, v), f(f(u, b), f(u, u)))$$

$$h(x, f(x, y)) = h(f(u, v), v)$$

Conjunctions of Equations and Disequations

Represent each disequation $t_1 \neq t_2$ as

$$x_1 = t_1 \wedge x_2 = t_2 \wedge x_1 \neq x_2$$

where x_1, x_2 are fresh variables.
If the most general unifier does not have same values for x_1 and x_2, then there is a substitution that assigns x_1 and x_2 different terms.

Theorem

Let E be a set of equations and disequations where disequations are only between variables. Let $E^+ = \{(t_1 \doteq t_2) \mid (t_1 = t_2) \in E\}$. Then

- *if E^+ has no unifier, the set E is unsatisfiable over ground terms;*
- *if E^+ has a most general unifier σ such that for all $(x \neq y) \in E$ we have $\sigma(x) \neq \sigma(y)$, then if the language contains at least one function symbol of arity at least one (i.e. the set of ground terms is infinite) then E is satisfiable.*

Why do we need infinite language? $\Sigma = \{a, b\}$, $x \neq y \wedge x \neq z \wedge y \neq z$

Example from Verification

$\Sigma = \{Leaf, Node\}$, $ar(Leaf) = 0, ar(Node) = 2$
Consider 'flip' of a tree invoked twice $z_1 \rightsquigarrow z_2 \rightsquigarrow z_3$
Show that the following implication holds for all variables
$z_1, z_2, z_3, x_1, y_1, x_2, y_2$ whose values range over $Terms_\Sigma$

$$(((z_1 = Leaf \wedge z_2 = Leaf) \vee (z_1 = Node(x_1, y_1) \wedge z_2 = Node(y_1, x_1)))$$
$$\wedge((z_3 = Leaf \wedge z_3 = Leaf) \vee (z_2 = Node(x_2, y_2) \wedge z_3 = Node(y_2, x_3))))$$
$$\rightarrow z_3 = z_1$$

Decision Proceudures for Term Algebras

How to handle arbitrary quantifier-free formulas?

How to handle selectors and tests?

Can we also support quantifiers?