Hoare Logic

Axiomatic Semantics
(AKA program logics)

A system for proving properties about programs

Key idea:

- We can define the semantics of a construct by describing its effect on **assertions** about the program state.

Two components

- A language for stating assertions (“the assertion logic”)
  - Can be First-Order Logic (FOL), a specialized logic such as separation logic, or Higher-Order Logic (HOL), which can encode the others.
  - Many specialized languages developed over the years: Z, Larch, JML, Spec#
- Deductive rules (“the program logic”) for establishing the truth of such assertions

Hoare Triples

```
{A} stmt {B}
Precondition        Postcondition
```

Hoare triple

- If the program state *before* execution satisfies A, and the execution of stmt *terminates*, the program state *after* execution satisfies B
- This is a partial correctness assertion.

- We sometimes use the notation

```
[A] stmt [B]
```

to denote a total correctness assertion

which means you also have to prove termination.

What do assertions mean?

The language of assertions:

- $A \coloneqq \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 \leq e_2 \mid A_1 \land A_2 \mid \neg A \mid \forall x. A$
- $e \coloneqq 0 \mid 1 \mid \cdots \mid x \mid y \mid \cdots \mid e_1 + e_2 \mid e_1 \cdot e_2$

Notation $\sigma \vDash A$ means that the assertion holds on state σ .

- A is interpreted inductively over state σ as a FO structure.
- Ex. $\sigma \vDash x = 1$ iff. $\sigma[x] = 1$
- Ex. $\sigma \vDash A \land B$ iff. $\sigma \vDash A$ and $\sigma \vDash B$

Derivation Rules

Derivation rules for each language construct

$$\frac{}{\vdash \{A[x \to e]\}x := e\ \{A\}}$$

$$\frac{\vdash \{A \wedge b\}c_1\ \{B\} \qquad \vdash \{A \wedge not\ b\}c_2\ \{B\}}{\vdash \{A\}if\ b\ then\ c_1\ else\ c_2\ \{B\}}$$

$$\frac{\vdash \{A\}c_1\ \{C\} \qquad \vdash \{C\}c_2\ \{B\}}{\vdash \{A\}c_1; c_2\ \{B\}}$$

$$\frac{\vdash \{A \wedge b\}c\ \{A\}}{\vdash \{A\}while\ b\ do\ c\ \{A \wedge not\ b\}}$$

Can be combined with the rule of consequence

$$\frac{\vdash A' \Rightarrow A \vdash \{A\}c\ \{B\} \vdash B \Rightarrow B'}{\vdash \{A'\}c\ \{B'\}}$$

Example

The following program purports to compute the square of a given integer n (not necessarily positive).

```
int i, j;
i := 1;
j := 1;
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
return i;
```

Example

```
{true}
int i, j;
i := 1;
j := 1;
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{??}
i := 1;
{??}
j := 1;
{??}
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
{??}
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{true}  //strongest postcondition
i := 1;
{i=1}  //strongest postcondition
j := 1;
{i=1 ∧ j=1}  //strongest postcondition
{??}  //loop invariant
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
{i = n*n} //weakest precondition
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{true}  //strongest postcondition
i := 1;
{i=1}  //strongest postcondition
j := 1;
{i=1 ∧ j=1}  //strongest postcondition
{??}  //loop invariant
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
{i = n*n} //weakest precondition
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{true}  //strongest postcondition
i := 1;
{i=1}  //strongest postcondition
j := 1;
{i=1 ∧ j=1}  //strongest postcondition
{i = j*j}  //loop invariant
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
{i = n*n} //weakest postcondition
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{true}  //strongest postcondition
i := 1;
{i=1}  //strongest postcondition
j := 1;
{i=1 ∧ j=1}  //strongest postcondition
{i = j*j}  //loop invariant
while (j != n) {
   {i = j*j ∧ j != n}
   {i + 2*j + 1 = (j+1)*(j+1)}
   i := i + 2*j + 1;
   {i = (j+1)*(j+1)}
   j := j+1;
   {i = j*j }
}
{i = n*n} //weakest postcondition
return i;
{i = n*n}
```

Soundness and Completeness

What does it mean for our derivation rules to be sound?

What does it mean for them to be complete?

So, are they complete?

{true} x:=x {p}

{true} c {false}

Relative Completeness in the sense of Cook (1974)

Expressible enough to express intermediate assertions, e.g., loop invariants