Operational Semantics

- Useful for exploration
- Useful to guide implementation
- Useful to show correctness of implementation
- Derived assertions correspond to individual test cases
- More general statements need to be shown at the meta-level
- Not useful to prove general properties of programs
  - termination
  - correctness

Correctness

- Correctness is always relative to a specification
- A specification is — in general — a logical formula
  — many different logics are used!
- A program is correct iff it satisfies its specification
- Using logical methods to prove correctness is called formal verification
  — Using (normally human-aided) syntactic methods: proving
  — normally necessary for functional requirements
  — Using (exhaustive, automated) semantic methods: model checking
  — most useful for safety & lifeness properties (finite models)
- How do you show a specification is correct?
  - Validation: Are we building the right product?
  - Verification: Are we building the product right?

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\[
\{P\} S \{Q\}
\]

where \(P\) and \(Q\) are formulae denoting conditions on execution states:
- \(P\) is the precondition
- \(S\) is a program fragment (statement)
- \(Q\) is the postcondition

A Hoare triple \(\{P\} S \{Q\}\) has two readings:
- **Total correctness:** If \(S\) starts in a state satisfying \(P\), then it terminates and its terminating state satisfies \(Q\)
  — “\(S\) is totally correct with respect to \(P\) and \(Q\)’
- **Partial correctness:** If \(S\) starts in a state satisfying \(P\) and terminates, then its terminating state satisfies \(Q\)
  — “\(S\) is partially correct with respect to \(P\) and \(Q\)’

(“terminates” means “terminates without run-time error”)

Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Therefore:
- Operational semantics facilitates investigation of examples (“testing”)
- Axiomatic semantics facilitates relating a program with its specification — verification
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “s ⊨ P” means “condition P holds, or is valid, in state s”

For example:
- \{x \mapsto 5, y \mapsto 7\} \models x > 0
- \{x \mapsto 5, y \mapsto 7\} \models \sum_{i=0}^{10} = 55
- \{x \mapsto 5, y \mapsto 7\} \not\models x > y

Proving Partial and Total Correctness

Total correctness of \(\{P\} S \{Q\}\)
is equivalent to

partial correctness of \(\{P\} S \{Q\}\) together with the fact that S terminates when started in a state satisfying P

⇒ usually, separate termination proof!

- For partial correctness, it is relatively easy to give a direct proof calculus
- Proving partial correctness therefore does not need operational semantics
- In the following, we will study and use this calculus
- (Termination proofs use different methods — well-ordered sets)

Unless explicitly mentioned, we read “\(\{P\} S \{Q\}\)” as meaning partial correctness.

Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence:

\[
\frac{P \Rightarrow P'}{\{P'\} S \{Q'\}} \quad \frac{Q' \Rightarrow Q}{\{P\} S \{Q\}}
\]

Sequence:

\[
\frac{\{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}}
\]

Conditional:

\[
\frac{\{P \land b\} S_1 \{Q\} \quad \{P \land \neg b\} S_2 \{Q\}}{\{P\} \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \{Q\}}
\]

while-Loop:

\[
\frac{\{INV \land b\} S \{INV\}}{\{INV\} \text{while } b \text{ do } S \od \{INV \land \neg b\}}
\]
Axiom Schema for Assignments

\[ \{P[x \leftarrow e]\}x := e\{P\} \]

Examples:

- \(\{2 = 2\}x := 2\{x = 2\}\)
- \(\{x + 1 = 2\}x := x + 1\{x = 2\}\)
- \(\{n + 1 = 2\}x := n + 1\{x = 2\}\)

Typically, Hoare triples are derived starting from the postcondition — backward reasoning.

Considering this axiom schema as a way to calculate a precondition from assignment and postcondition, it calculates the weakest precondition that completes a valid Hoare triple.

### Example Verification

\(\{\text{True}\}\) \(k := 0; s := 0; \text{while } k \neq n \text{ do } k := k + 1; s := s + k \text{ od}\{s = \sum_{i=1}^{n} i\}\)

\(\equiv \{\text{True}\}\) \(k := 0; s := 0; \text{while } k \neq n \text{ do } k := k + 1; s := s + k \text{ od}\{s = \sum_{i=1}^{k} i \land k = n\}\)

\(\equiv \{\text{True}\}\) \(k := 0; s := 0\{s = \sum_{i=1}^{k} i\}\)

\(\land \{s = \sum_{i=1}^{k} i\}\) \(\text{while } k \neq n \text{ do } k := k + 1; s := s + k \text{ od}\{s = \sum_{i=1}^{k} i \land k = n\}\)

\(\equiv \{\text{True}\}\) \(k := 0; s := 0\{s = \sum_{i=1}^{k} i\}\)

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it
- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration
- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

**Auxiliary variables used in a loop are usually involved in the invariant!**

Given a loop "**while** b **do** S **od**" and a postcondition Q, use the consequence rule to strengthen Q to \(Q'\), such that

- \(Q' \Rightarrow Q\) (strengthening)
- \(Q'\) involves all auxiliary variables — **generalisation**!
- \(Q'\) is of shape \(INV \land \neg b\)
Simultaneous Assignments

\{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n](x_1, \ldots, x_n) := (e_1, \ldots, e_n)\}\{P\}

Examples:

- \{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\}
- \{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\}

Simultaneous assignments
- shorten code
- save auxiliary variables (for example for swapping)
- make proofs easier
- require simultaneous substitution

Example Problems (with Simultaneous Assignments)

\{n \geq 0\} \ (y, a, b) := (0, 1, 1) ;

\textbf{while} y \neq n \textbf{do} (y, a, b) := (y + 1, b, a + b) \textbf{od} \ \{a = fib_n\}

\iffalse \text{(right consequence)} \fi
\{n \geq 0\} P \ {a = fib_y \land b = fib_{y+1} \land y = n} \land (a = fib_y \land b = fib_{y+1} \land y = n \Rightarrow a = fib_n)

\iffalse \text{(sequence, logic)} \fi
\{n \geq 0\} (y, a, b) := (0, 1, 1) \ \{a = fib_y \land b = fib_{y+1}\} \land
\{a = fib_y \land b = fib_{y+1}\} \textbf{while} y \neq n \textbf{do} \textbf{od} [a = fib_y \land b = fib_{y+1} \land y = n] \land True

\{n \geq 0\} \ (y, a, b) := (0, 1, 1) ;

\textbf{while} y \neq n \textbf{do} (y, a, b) := (y + 1, b, a + b) \textbf{od} \ \{a = fib_n\}

\iffalse \text{(left consequence, while)} \fi
\{n \geq 0\} \ (y, a, b) := (0, 1, 1) ;

\textbf{while} y \neq n \textbf{do} (y, a, b) := (y + 1, b, a + b) \textbf{od} \ \{a = fib_n\}

\iffalse \text{(logic)} \fi
\{n \geq 0\} \ (y, a, b) := (0, 1, 1) ;

\textbf{while} y \neq n \textbf{do} (y, a, b) := (y + 1, b, a + b) \textbf{od} \ \{a = fib_n\}

\iffalse \text{(arithmetic, assignment)} \fi
\{n \geq 0\} \ (y, a, b) := (0, 1, 1) ;

\textbf{while} y \neq n \textbf{do} (y, a, b) := (y + 1, b, a + b) \textbf{od} \ \{a = fib_n\}

What does this program do?