Axiomatic Semantics

Stansifer Ch 2.4, Ch. 9

Winskel Ch.6

Slonneger and Kurtz Ch. 11
Semantics: How?

**Operational semantics** in the general mathematical sense: imagine an abstract machine

Some notion of the state of this machine

**Transition function**: given the current state, what is the next state?

**Axiomatic semantics**: general high-level properties of a program, specified with first-order logic

As with axiomatic semantics, we will use inference rules to construct proofs (derivations) of such properties

**Goal**: should be able to prove all true statements about the program, and not be able to prove any false statements
State (of the memory)

**State**: a function $\sigma$ from variable names to values

An abstraction of the contents of the physical memory

For simplicity, we will only consider integer variables

$\sigma$: Variables $\rightarrow \{0, -1, 1, -2, 2, \ldots\}$

Set of states: specify with **assertion** in first-order logic

E.g. the set $x=1, y=2, z=1$ or $x=1, y=2, z=2$ or $x=1, y=2, z=3$

Assertion: $x=1 \land y=2 \land 1 \leq z \leq 3$

An assertion $p$ represents the set of states that satisfy that assertion

We will write $\{p\}$ to denote this set of states
Building Blocks of Assertions

Variables from the program code
In the program they are part of the syntax, here they are part of the assertion

Extra helper variables (examples later; used to represent values)

Standard elements of first-order logic formulas

\[
= \quad < \quad \land \quad \lor \quad \neg \quad \exists x \quad \forall x \quad \text{true} \quad \text{false}
\]

\(\exists x\) and \(\forall x\) range over integer values (\(x\) is a helper variable)
Note: in higher-order logic, \(\exists x\) and \(\forall x\) can range over sets, sets of sets, etc.

Operators from the programming language

\(+ \quad - \quad *\) etc.
First-Order Logic Formulas (formal definition)

Terms
If \( x \) is a variable, \( x \) is a term
If \( n \) is an integer constant, \( n \) is a term
If \( t_1 \) and \( t_2 \) are terms, so are \( t_1 + t_2, t_1 - t_2, \ldots \)

Formulas

true and false
\( t_1 < t_2 \) and \( t_1 = t_2 \) for terms \( t_1 \) and \( t_2 \)
\( f_1 \land f_2, f_1 \lor f_2, \neg f_1 \) for formulas \( f_1, f_2 \)
\( \exists x.f \) and \( \forall x.f \) for a formula \( f \)
Free vs. Bound Variable Occurrences

An occurrence of a variable $x$ is **bound** if it is in the scope of $\exists x$ or $\forall x$

An occurrence is **free** if it is not bound

Example: $\exists i. k = i * j$: $k$ and $j$ are free, $i$ is bound

Example: $(x+1 < y+2) \land (\exists x. x+3 = y+4)$

Could rename bound $x$ without any effect: $\exists z. z+3 = y+4$

Substitution: $f[e/x]$ is the formula $f$ with **all free occurrences** of $x$ replaced by $e$

In an assertion $p$ that represents a set of states $\{ p \}$

Free variables correspond to variables from the code

Bound variables are helper variables which range over integer values
When Does a State Satisfy an Assertion?

**Value of a term** in some state $\sigma$

$\sigma(x)$ for variable $x$, $n$ for constant $n$, the usual arithmetic

for terms $t_1+t_2$, $t_1-t_2$,...

$\sigma$ satisfies the assertion $t_1=t_2$ if and only if $t_1$ and $t_2$

have the same value in $\sigma$

Similarly for assertion $t_1<t_2$

$\sigma$ satisfies $f_1 \land f_2$ if and only if it satisfies $f_1$ and $f_2$

Similarly for $f_1 \lor f_2$ and $\neg f_1$
When Does a State Satisfy an Assertion?

σ satisfies $\forall x. f$ if and only if for every integer value $n$,
σ satisfies $f[n/x]$

Example: which states satisfy $\forall x. (x+y=y+x)$?
That is, which σ satisfies $0+y=y+0, 1+y=y+1, 2+y=y+2, ...$?
Answer: any σ in which $y$ is initialized ($y$ is a variable from the code)

σ satisfies $\exists x. f$ if and only if for some integer value $n$,
σ satisfies $f[n/x]$

Example: which states satisfy $\exists i. k=i*j$? ($k$ and $j$ are variables from the code)
When Does a State Satisfy an Assertion?

Recall that \{ p \} is a set of states satisfying p

Simple properties

\[
\{ p \land q \} = \{ p \} \cap \{ q \}
\]

\[
\{ p \lor q \} = \{ p \} \cup \{ q \}
\]

\[
\{ \neg p \} = U - \{ p \} \text{ where } U \text{ is the universal set (i.e. set of all states)}
\]

if \( p \Rightarrow q \) then \( \{ p \} \subseteq \{ q \} \)

Note: saying “\( p \Rightarrow q \)” is the same as saying “\( \neg p \lor q \) is always true” which is the same as “every state satisfies \( \neg p \lor q \)” which is the same as \( \{ \neg p \lor q \} = U \) which is the same as \( (U - \{ p \}) \cup \{ q \} = U \) which is the same as \( \{ p \} \subseteq \{ q \} \)

Example: \( x=2 \land y=3 \Rightarrow x=2 \), so \( \{ x=2 \land y=3 \} \subseteq \{ x=2 \} \)
Examples of Assertions

Three variables in the code: x, y, z

\{ x = 1 \land 1 \leq y \leq 5 \land 1 \leq z \leq 10 \}: set of size 50

\{ x = 1 \land y = 2 \}: infinite set

\{ x = 1 \land 1 \leq y \leq 5 \}: infinite set

\{ x = y + z \}: all states such that \( \sigma(x) = \sigma(y) + \sigma(z) \)

\{ x = x \}: the set of all states i.e. universal set U

\{ \text{true} \}: the set of all states i.e. universal set U

\{ x \neq x \}: the empty set

\{ \text{false} \}: the empty set
IMP: Simple Imperative Language

<c> ::= skip | id := <ae> | <c> ; <c>
  | if <be> then <c> else <c>
  | while <be> do <c>

<ae> ::= id | int | <ae> + <ae>
  | <ae> - <ae> | <ae> * <ae> | <ae> / <ae>

<be> ::= true | false
  | <ae> = <ae> | <ae> < <ae>
  | ¬ <be> | <be> ∧ <be>
  | <be> ∨ <be>
Hoare Triples


\[ \{ p \} S \{ q \} \]

- S is a piece of code (program fragment)
- \( p \) and \( q \) are assertions
  - \( p \): pre-condition, \( q \): post-condition

Intuitive meaning: if we start executing S from any state \( \sigma \) that satisfies \( p \), and if S terminates, then the resulting state \( \sigma' \) satisfies \( q \)

Terminology: will refer to the triples as results (think “results of proofs”)

Intuition

In \( \{ p \} S \{ q \} \) the relationship between \( p \) and \( q \) captures the essence of the semantics of \( S \).

Abstract description of constraints that any implementation of the language must satisfy: says something about properties of the final state, assuming we know something about the initial state.

Says nothing about how exactly these relationships will be achieved (in contrast to operational semantics): we do not know what intermediate steps are taken.
Operationally Valid Results

{ p } S { q } is valid if and only if, for every state \( \sigma \)
if \( \sigma \) satisfies \( p \) (i.e., \( \sigma \) belongs to set \{p\})
and the execution of \( S \) starting in \( \sigma \) terminates in
state \( \sigma' \)
then \( \sigma' \) must satisfy \( q \) (i.e., \( \sigma' \) belongs to set \{q\})

Example: \{ x=2 \} y:=x \{ y=2 \} is valid

Example: \{ false \} S \{ q \} valid for all possible \( S \) and \( q \)
More Examples

\{ x=1 \} \text{ skip } \{ x=1 \} \quad \text{Valid}

\{ x=1 \land y=1 \} \text{ skip } \{ x=1 \} \quad \text{Valid}

\{ x=1 \} \text{ skip } \{ x=1 \land y=1 \} \quad \text{Invalid}

\{ x=1 \} \text{ skip } \{ x=1 \lor y=1 \} \quad \text{Valid}

\{ x=1 \lor y=1 \} \text{ skip } \{ x=1 \} \quad \text{Invalid}

\{ x=1 \} \text{ skip } \{ \text{true} \} \quad \text{Valid}

\{ x=1 \} \text{ skip } \{ \text{false} \} \quad \text{Invalid}

\{ \text{false} \} \text{ skip } \{ x=1 \} \quad \text{Valid}
More Examples

{ x=1 \land y=2 } x := x+1 \{ x=2 \land y=2 \} \quad \text{Valid}

{ x=1 \land y=2 } x := x+1 \{ x \geq 2 \} \quad \text{Valid}

{ x=1 \land y=2 } x := x+1 \{ x=y \} \quad \text{Valid}

{ x=0 } \text{while } x<10 \text{ do } x:=x+1 \{ x=10 \} \quad \text{Valid}

{ x<0 } \text{while } x<10 \text{ do } x:=x+1 \{ x=10 \} \quad \text{Valid}

{ x \geq 0 } \text{while } x<10 \text{ do } x:=x+1 \{ x=10 \} \quad \text{Invalid}

{ x \geq 0 } \text{while } x<10 \text{ do } x:=x+1 \{ x \geq 10 \} \quad \text{Valid}
Termination

A result says: ... if S terminates ...

What if S does not terminate? The result is valid

We are concerned only with initial states for which S terminates

\{ x=3 \} while x \neq 10 do x:=x+1 \{ x=10 \}
\{ x \geq 0 \} while x \neq 10 do x:=x+1 \{ x=10 \}
\{ \text{true} \} while x \neq 10 do x:=x+1 \{ x=10 \}

All of these results are valid
Observations

What exactly does “valid result” mean?

We had an operational model of how the code would operate, and we “executed” the code in our heads using this model.

The result is valid with respect to the model.

The operational model can be formalized by defining an operational semantics (as we have already done):

- \(<c_1, \sigma_1> \rightarrow <c_2, \sigma_2> \rightarrow \ldots \rightarrow \sigma_n\) for small-step
- \(<c, \sigma> \rightarrow \sigma'\) for big-step

Goal: derive valid results without using operational reasoning (i.e., without saying “based on the operational semantics, we can tell how the state will change and what the final state will be”)

Purely formally, using a proof system (a set of inference rules)
Proofs

Proof = set of applications of inference rules
Starting from one or more axioms
Conclusions are subsequently used as premises
Can be represented by a derivation tree

The conclusion of the last rule is proved (derived)
If a proof exists, the result is provable (derivable)

We have seen this before: inference rules for
\[ t : T \] (typing relation)
\[ <ae,\sigma> \rightarrow <ae',\sigma> \quad <c,\sigma> \rightarrow <c',\sigma'> \] (small-step operational)
\[ <ae,\sigma> \rightarrow \text{int} \quad <c,\sigma> \rightarrow \sigma' \] (big-step operational)
Terminology

**Assertion**: may be **satisfied** or **not satisfied** by a particular state

**Result**: may be **valid** or **invalid** in a particular operational model

**Result**: may be **derivable** or **not derivable** in a given proof system (set of inference rules)

Some meaningless statements (do not use)

“\{ p \} S \{ q \} is true”, “\{ p \} S \{ q \} is valid for some states”, “assertion p is not valid”
Soundness and Completeness

Properties of a proof system (axiomatic semantics) $A$ with respect to an operational model $M$

**Soundness (consistency):** every result we can prove (derive) in $A$ is valid in $M$

**Completeness:** every result that is valid in $M$ can be derived (proven) in $A$

Example: suppose $A$ had only one rule: $\{ p \} \text{skip} \{ p \}$

Everything we can derive is valid: *sound* system

Not everything valid is derivable: *incomplete* system
Inference Rules

\[
\{ p \} \text{skip} \{ p \}
\]

*skip axiom*, for any assertion \( p \)

\[
\{ x=1 \} \text{skip} \{ x=1 \}
\]
is derivable (provable)

\[
\{ x=1 \land y=2 \} \text{skip} \{ x=1 \}
\]
is valid but not derivable (provable)

\[
p' \Rightarrow p \quad \{ p \} \text{S} \{ q \} \quad q \Rightarrow q'
\]

\[
\{ p' \} \text{S} \{ q' \}
\]

*rule of consequence*

Recall that \( x \Rightarrow y \) means \( \{ x \} \subseteq \{ y \} \)

\[
 x=1 \land y=2 \Rightarrow x=1 \quad \{ x=1 \} \text{skip} \{ x=1 \}
\]

\[
\{ x=1 \land y=2 \} \text{skip} \{ x=1 \}
\]
Exercise: show that the following rule will make the proof system inconsistent (unsound) – i.e. it will be possible to derive something that is not operationally valid (so, this is a “bad” rule)
Substitution in Assertions

Notation: $p[e/x]$  
Other notations: $p^x_e$ and $p[x:=e]$  

$p[e/x]$ is the assertion $p$ with all free occurrences of $x$ replaced by $e$

Examples

$(x=y)[5/x]$ is $5=y$

$(x=y \land x=2)[5/x]$ is $5=y \land 5=2$
Assignment Axiom

\[
\{ p[e/x] \} \ x := e \ \{ p \}
\]

assignment axiom, for any assertion \( p \)

\[
\{ x+1 = y+z \} \ x := x+1 \ \{ x = y+z \}
\]

\[
\{ y+z > 0 \} \ x := y+z \ \{ x > 0 \}
\]

\[
\{ y+z = y+z \} \ x := y+z \ \{ x = y+z \}
\]
due to \( \text{true} \Rightarrow y+z = y+z \) and the consequence rule:

\[
\{ \text{true} \} \ x := y+z \ \{ x = y+z \}
\]

Why not just use \( \{ \text{true} \} \ x := e \ \{ x = e \} \) ? Is this valid for any possible \( x \) and \( e \)?
Intuition

The initial state must satisfy the same assertion except for e playing the role of x

Operational intuition: you cannot use it in an axiomatic derivation

Only allowed to use the axioms and rules

E.g. \{ x > 0 \} x := 1 \{ x = 1 \}

Not: “After assigning 1 to x, we end up in a state in which the value of x is 1”

But: “This can be proved using the assignment axiom and the rule of consequence”
Rule of Composition

\[
\begin{array}{cccc}
{p} \quad S_1 \quad {q} \quad {q} \quad S_2 \quad {r} \\
\hline
\{ p \} S_1 ; S_2 \{ r \}
\end{array}
\]

\[\text{rule of composition}\]

\[
\begin{array}{cccc}
\{ x+1=y+z \} \text{ skip} \quad \{ x+1=y+z \} \\
\hline
\{ x+1=y+z \} x:=x+1 \quad \{ x=y+z \}
\end{array}
\]

\[
\begin{array}{cccc}
\{ x+1=y+z \} \text{ skip; } x:=x+1 \quad \{ x=y+z \}
\end{array}
\]
Input/Output

Suppose we extend the language

\[ <c> ::= \text{skip} \mid \ldots \mid \text{write } <e> \mid \text{read id} \]

\(<e>\) is just shorthand for \(<ae>\) or \(<be>\)

Idea: treat input and output streams as variables

Use the assignment axiom

**write** modifies the output stream (\(^\) is “append”)

“write \(e\)” is \(\text{OUT} := \text{OUT} ^\ e\)

**read** modifies the input variable and the input stream

“read \(x\)” is \(x := \text{head(IN)}; \text{IN} := \text{tail(IN)}\)

Here we use primitive operations on lists: **head** is like CAR and **tail** is like CDR
Write and Read Axioms

\[
\{ p[OUT^e / OUT] \} \text{write } e \{ p \}
\]

\[
\text{OUT=<> } \Rightarrow \text{OUT^4=<>} \quad \{ \text{OUT^4=<>} \} \text{write } 4 \{ \text{OUT=<>} \}
\]

\[
\{ \text{OUT=<>} \} \text{write } 4 \{ \text{OUT=<>} \}
\]

\[
\{ (p[tail(IN)/IN]) \text{[head(IN)/x]} \} \text{read } x \{p\}
\]

\[
\{ \text{tail(IN)=<> } \land \text{head(IN)=3} \} \text{read } x \{ \text{IN=<> } \land \text{x=3} \}
\]

\[
\text{IN=<3,4} \Rightarrow \text{tail(IN)=<> } \land \text{head(IN)=3}
\]

\[
\{ \text{IN=<3,4} \} \text{read } x \{ \text{IN=<> } \land \text{x=3} \}
\]
Example

Prove

\{ \text{IN} = <3,4> \land \text{OUT} = <> \} \\
read x; \\
read y; \\
write x+y; \\
\{ \text{OUT} = <7> \} \\

Here <> denotes an empty sequence
Example

1. Using the write axiom and the desired post-condition:
   \[ \{ \text{OUT}^{x+y} = <7> \} \text{ write } x+y \{ \text{OUT} = <7> \} \]

2. Using (1) and the read axiom:
   \[ \{ \text{OUT}^{x+\text{head(IN)}} = <7> \} \text{ read } y \{ \text{OUT}^{x+y} = <7> \} \]

3. Using (2) and the read axiom again:
   \[ \{ \text{OUT}^{\text{head(IN)}+\text{head(tail(IN))}} = <7> \} \text{ read } x \]
   \[ \{ \text{OUT}^{x+\text{head(IN)}} = <7> \} \]

4. Using sequential composition:
   \[ \{ \text{OUT}^{\text{head(IN)}+\text{head(tail(IN))}} = <7> \} \]
   \[ \text{read } x; \text{ read } y; \text{ write } x+y \{ \text{OUT} = <7> \} \]

5. Rule of consequence: \( \text{IN} = <3,4> \land \text{OUT} = <> \Rightarrow \text{OUT}^{\text{head(IN)}+\text{head(tail(IN))}} = <7> \)
Proof Strategy

For any sequence of assignments and input/output operations:

Start with the last statement
Apply the assignment/read/write axioms working backwards
At the end, apply the rule of consequence to change the pre-condition to be the desired one

e.g. want \{ p \} s_1 ; \ldots ; s_n \{ q \}
First derive \{ q' \} s_n \{ q \} using appropriate axiom
Then do \{ q'' \} s_{n-1} \{ q' \} etc. and eventually \{ p' \} s_1 \{ \ldots \}
Using composition rule: \{ p' \} s_1 ; \ldots ; s_n \{ q \}
With \( p \Rightarrow p' \) and rule of consequence: \{ p \} s_1 ; \ldots ; s_n \{ q \}
If-Then-Else Rule

\[
\{ p \land b \} S_1 \{ q \} \quad \{ p \land \lnot b \} S_2 \{ q \}
\]

\[
\{ p \} \text{ if } b \text{ then } S_1 \text{ else } S_2 \{ q \}
\]

Example:

\[
\{ y = 1 \}
\]

if \( y = 1 \) then \( x := 1 \) else \( x := 2 \)

\[
\{ x = 1 \}
\]
If-Then-Else Example

\[
y = 1 \land y = 1 \implies 1 = 1 \quad \{1 = 1\} \ x := 1 \quad \{x = 1\}
\]

\[
\{ y = 1 \land y = 1 \} \ x := 1 \quad \{x = 1\}
\]

\[
y = 1 \land \neg(y = 1) \implies 2 = 1 \quad \{2 = 1\} \ x := 2 \quad \{x = 1\}
\]

\[
\{ y = 1 \land \neg(y = 1) \} \ x := 2 \quad \{x = 1\}
\]

\{y = 1\} \text{ if } y = 1 \text{ then } x := 1 \text{ else } x := 2 \quad \{x = 1\}
Can We Use a Simplified If-Then-Else Rule?

Why not simply

\[
\begin{array}{c}
\{ p \} S_1 \{ q \} & \{ p \} S_2 \{ q \} \\
\hline
\{ p \} \text{if } b \text{ then } S_1 \text{ else } S_2 \{ q \}
\end{array}
\]

Works for

\{\text{true}\} \text{ if } y=1 \text{ then } x:=1 \text{ else } x:=2 \{x=1 \lor x=2\}

Easy to prove that

\{ true \} x:=1 \{ x = 1 \lor x = 2 \}
\{ true \} x:=2 \{ x = 1 \lor x = 2 \}

with assignment axiom and consequence rule
Can We Use a Simplified If-Then-Else Rule?

Does **not** work for

\[ \{ y=1 \} \text{ if } y=1 \text{ then } x:=1 \text{ else } x:=2 \{ x=1 \} \]

Attempt for a proof: we need

\[ \{ y=1 \} x:=1 \{ x=1 \} \text{ and } \{ y=1 \} x:=2 \{ x=1 \} \]

The second result cannot be proven using axioms and rules

With the simplified rule, the proof system becomes **incomplete**

i.e. it becomes impossible to prove something that is, in fact, operationally valid
While Loop Rule

Problem: proving

\{ p \} \textbf{while} b \textbf{do} S \textbf{end} \{ q \}

for arbitrary \( p \) and \( q \) is undecidable

Need to encode the knowledge that went into constructing the loop

For each loop, we need an invariant \( I \) – an assertion that must be satisfied by

1. the state immediately before we enter the loop
2. the state at the end of each iteration
3. the state immediately after we exit the loop

\textit{In homeworks/exams, always check these 3 conditions!}

Finding a loop invariant is the hard part
While Loop Rule

\[
\begin{align*}
  \{ I \land b \} & \quad S \quad \{ I \} \\
  \{ I \} \quad & \text{while b do S end} \quad \{ I \land \neg b \}
\end{align*}
\]

In practice often combined with the rule of consequence

\[
\begin{align*}
p \Rightarrow I & \quad \{ I \land b \} \quad S \quad \{ I \} \quad (I \land \neg b) \Rightarrow q \\
\{ p \} \quad & \text{while b do S end} \quad \{ q \}
\end{align*}
\]
Example: Division

Prove

\[ \{ (x \geq 0) \land (y > 0) \} \]

\begin{verbatim}
q := 0;
r := x;
while (r - y) \geq 0 do
    q := q + 1;
r := r - y
end
\end{verbatim}

\[ \{ (x = q \times y + r) \land (0 \leq r < y) \} \]

q: quotient
r: remainder

Note: what if \( y > 0 \) was not in the precondition?
Is the result valid?
Is it derivable?
Example: Division

Loop invariant

Should state relationship between variables used in loop

\[(x=q\cdot y+r)\]

Needs a boundary condition to make the proof work

\[(x=q\cdot y+r) \land (0 \leq r)\]

Because we need \((I \land \neg b) \Rightarrow q\) which in this case means we need

\[(I \land (r-y<0)) \Rightarrow (x=q\cdot y+r) \land (0 \leq r<y)\]
Example: Division

\{ (x \geq 0) \land (y > 0) \}

q := 0;

r := x;

\{ (x=q*y+r) \land (0 \leq r) \}

while \((r - y) \geq 0\) do

\begin{align*}
q &:= q + 1; \\
r &:= r - y
\end{align*}

end

\{ (x=q*y+r) \land (0 \leq r) \land (r-y<0)\}

And of course \((x=q*y+r) \land (0 \leq r) \land (r-y<0)\) implies \((x=q*y+r) \land (0 \leq r<y)\)
Example: Division

Code before the loop

\{ (x \geq 0) \land (y > 0) \} \\
q := 0; \\
r := x; \\
\{ (x = q \cdot y + r) \land (0 \leq r) \} – the invariant

Proof: backward application of the assignment axiom, then the composition rule, and finally the consequence rule using

\( (x \geq 0) \land (y > 0) \Rightarrow (x = 0 \cdot y + x) \land (0 \leq x) \)
Example: Division

Need: \{ I \land b \} S \{ I \}

\{ (x=q\ast y+r) \land (0\leq r) \land (r-y\geq 0) \}

q := q + 1;

r := r - y

\{ (x=q\ast y+r) \land (0\leq r) \}

Similar process, ending with rule of consequence
for \( (x=q\ast y+r) \land (0\leq r) \land (r-y\geq 0) \Rightarrow \)

\( (x=(q+1)\ast y+r-y) \land (r-y\geq 0) \)

which is always true
Example: Division

At exit: need the implication \((I \land \neg b) \Rightarrow q\)

\((x=q*y+r) \land (0 \leq r) \land (r-y<0) \Rightarrow (x=q*y+r) \land (0 \leq r<y)\)

Trivially true
Example: Fibonacci Numbers

{ n > 0 } 

i := n;

f := 1;

h := 1;

while i > 1 do
    h := h + f;
    f := h - f;
    i := i - 1
end

{ f = fib(n) }

Math definition:

fib(1) = 1
fib(2) = 1

...

fib(i+1) = fib(i) + fib(i-1)

...
Example: Fibonacci Numbers

Invariant: \( f = \text{fib}(n-i+1) \land h = \text{fib}(n-i+2) \land i > 0 \)

Steps of the proof, presented informally

\( n > 0 \Rightarrow 1 = \text{fib}(n-n+1) \land 1 = \text{fib}(n-n+2) \land n > 0 \)

\( i := n; f := 1; h := 1 \)

\{ \( f = \text{fib}(n-i+1) \land h = \text{fib}(n-i+2) \land i > 0 \) \} [\text{loop invariant}] \n
For loop body:

\( f = \text{fib}(n-i+1) \land h = \text{fib}(n-i+2) \land i > 0 \land i > 1 \) [this is \( I \land b \)] \( \Rightarrow \)

\( h = \text{fib}(n-i+2) \land h + f = \text{fib}(n-i+3) \land (i-1) > 0 \) \( \Rightarrow \)

\( h + f - f = \text{fib}(n-(i-1)+1) \land h + f = \text{fib}(n-(i-1)+2) \land (i-1) > 0 \)

Strictly speaking, also need \( n-i+1 \geq 1 \) so that \( \text{fib}(\ldots) \) is defined. But let’s ignore this for simplicity.
Example: Fibonacci Numbers
\[
\begin{align*}
\{ & h+f-f=\text{fib}(n-(i-1)+1) \land h+f=\text{fib}(n-(i-1)+2) \land (i-1)>0 \\
& h:=h+f; \\
& \{ & h-f=\text{fib}(n-(i-1)+1) \land h=\text{fib}(n-(i-1)+2) \land (i-1)>0 \\
& f:=h-f; \\
& \{ & f=\text{fib}(n-(i-1)+1) \land h=\text{fib}(n-(i-1)+2) \land (i-1)>0 \\
& i:=i-1 \\
& \{ & f=\text{fib}(n-i+1) \land h=\text{fib}(n-i+2) \land i>0 \} \text{ [loop invariant]} \\
\end{align*}
\]

end of loop: \( f=\text{fib}(n-i+1) \land h=\text{fib}(n-i+2) \land i>0 \land i\leq 1 \Rightarrow f=\text{fib}(n) \)
Example: I/O [try this at home]

\{ \text{IN}=\langle 1,2,\ldots,100 \rangle \land \text{OUT}=\langle \rangle \} \\
\text{read } x; \\
\text{while } x \neq 100 \text{ do} \\
\quad \text{write } x; \\
\quad \text{read } x \\
\text{end} \\
\{ \text{OUT} = \langle 1,2,\ldots,99 \rangle \} \\
\text{Loop invariant: } \text{OUT}^x \text{^IN} = \langle 1,2,\ldots,100 \rangle \\
\text{At loop exit: } \text{OUT}^x \text{^IN} = \langle 1,2,\ldots,100 \rangle \land x=100 \Rightarrow \text{OUT} = \langle 1,2,\ldots,99 \rangle
Class Invariants

The idea of *invariant* is also used in object-oriented programming. Consider an object \( o \) of class \( C \)

**The internal state of \( o \) must satisfy \( C \)'s invariant before & after each call to a public method**

Example: bank account class with lists *in* and *out* of deposits and withdrawals and a value *balance*

Invariant: \( balance = \text{in.total} - \text{out.total} \) where *total* is the accumulated value of the list

Method \( \text{withdraw}(x) \): invariant is part of both the pre-condition and the post-condition

See, for example, Bertrand Meyer, “Object-Oriented Software Construction” and “Class invariants: concepts, problems and solutions”

**Note:** remember “assert” in Java and many other languages? It can be used at run time to find violations of class invariants (i.e., bugs)
Class Invariants

Very simple example: class Point, with internal field
int \( x \) and limit int \( L \)

Invariant: \( 0 \leq x \leq L \)

Method \textit{moveLeft}: \( x := x - 1; \) if \( x < 0 \) then \( x := L \)

Method \textit{moveRight}: \( x := x + 1; \) if \( x > L \) then \( x := 0 \)

\textbf{Note}: private methods may violate the invariant

private \textit{goBack}: \( x := x - 1 \)

public \textit{moveLeft}: \textit{goBack}(); if \( x < 0 \) then \( x := L \)
Soundness and Completeness

This set of rules is **sound (consistent) & complete** for IMP

Sound: Anything that is provable is operationally valid
Complete: anything operationally valid can be proven

Ensuring **soundness/completeness** is hard

One approach: start with a known system A and make changes to obtain system A’

If A is complete and all results derivable in A are also derivable in A’: A’ is complete
If A is sound and all results derivable in A’ are also derivable in A: A’ is sound

E.g.: if we replace if-then-else rule with the simplified one, A’ is sound:
If \( \{ p \} \text{if } b \text{ then } S_1 \text{ else } S_2 \{ q \} \) is derivable in A’, \( \{ p \} S_1 \{ q \} \) and \( \{ p \} S_2 \{ q \} \) must be derivable in A’. Inductively assume \( \{ p \} S_1 \{ q \} \) and \( \{ p \} S_2 \{ q \} \) are derivable in A; thus \( \{ p \land b \} S_1 \{ q \} \) and \( \{ p \land \neg b \} S_2 \{ q \} \) are also derivable in A (rule of consequence). Therefore \( \{ p \} \text{if } b \text{ then } S_1 \text{ else } S_2 \{ q \} \) is derivable in A
Total Correctness

So far we only had **partial correctness**

Want to generalize to handle
  Reading from empty input
  Division by zero and other run-time errors
  Idea: **add correctness check to pre-condition**

Also, want to handle non-termination
  Do this through a **termination function**
Hoare Triples – Total Correctness

\[ \langle p \mid S \mid q \rangle \]

S is a piece of code (program fragment)
p: pre-condition, q: post-condition

If we start executing S from any state \( \sigma \) that satisfies \( p \), then \( S \) terminates and the resulting state \( \sigma' \) satisfies \( q \)
Total Correctness Rules

New assignment axiom

\[ p \Rightarrow (D(e) \land q[e/x]) \]

\[ \langle \ p \mid x := e \mid q \rangle \]

where \( D(e) \) is a predicate to ensure that \( e \) is well-defined (e.g., no division by zero)

New read axiom: make sure \( \text{IN} \) is not empty

\[ p \Rightarrow (\text{IN} \neq \langle \rangle \land (q[\text{tail(IN)}/\text{IN}])[\text{head(IN)}/x]) \]

\[ \langle \ p \mid \text{read} \ x \mid q \rangle \]
Examples

To derive \( \langle a>0 \land b>0 \mid c := a/b \mid c>0 \rangle \):
need
\[ a>0 \land b>0 \Rightarrow b\neq 0 \land a/b>0 \]

To derive \( \langle \text{IN}=<3,4> \mid \text{read} x \mid \text{IN}=<4> \land x=3 \rangle \):
need
\[ \text{IN}=<3,4> \Rightarrow \text{IN}\neq<> \land \text{tail(IN)}=<4> \land \text{head(IN)}=3 \]
Total Correctness Rule for While

Idea: find termination function $f$ (some expression based on program variables)

- Decreases with every iteration
- Always positive at start of loop body: $(I \land b) \Rightarrow f > 0$
- Also called “progress function”

\[
(l \land b) \Rightarrow f > 0 \quad \langle I \land b \land f=k \mid S \mid I \land f<k \rangle \]

\[\langle I \mid \text{while } b \text{ do } S \text{ end} \mid I \land \neg b \rangle\]
Example: Division

\( \langle (x \geq 0) \land (y > 0) \mid \)  
\( q := 0; \)  
\( r := x; \)  
\( \text{while} \ (r - y) \geq 0 \ \text{do} \)  
\( q := q + 1; \)  
\( r := r - y \)  
\( \text{end} \)  
\( \mid (x=q*y+r) \land (0 \leq r) \rangle \)

Invariant: \( (x=q*y+r) \land (0 \leq r) \)

Termination function: \( r \)

Question 1: does \( (x=q*y+r) \land (0 \leq r) \land (r-y) \geq 0 \) imply \( r > 0 \) ? Not quite ...

Question 2: can we derive \( \langle \ldots \ r = k \mid \)  
\( q := q+1; \ r := r-y \)  
\( \mid \ldots \ r < k \rangle \) ? Not quite ...
Example: Division revised

\[ \langle (x \geq 0) \land (y > 0) \mid \]

q := 0;
r := x;
while \((r - y) \geq 0\) do
    q := q + 1;
r := r - y
end
\[ \langle (x = q \cdot y + r) \land (0 \leq r) \land (y > 0) \rangle \]

Invariant: \((x = q \cdot y + r) \land (0 \leq r) \land (y > 0)\)

Termination function: \(r\)

Question 1: does \( (x = q \cdot y + r) \land (0 \leq r) \land (y > 0) \land (r - y) \geq 0 \) imply \( r > 0 \)? Yes

Question 2: can we derive \( \langle \ldots r = k \mid \)

\( q := q + 1; r := r - y \)

\( \langle \ldots r < k \rangle \)? Yes, since \( y > 0 \) is part of the loop invariant

Note: what if \( y > 0 \) was not in the precondition? Is this triple derivable?
Example: Fibonacci Numbers

\[
\begin{align*}
\langle \ n > 0 \ | \\
i & := n; \\
f & := 1; \\
h & := 1; \\
\text{while } i > 1 \text{ do} & \\
& \quad h := h + f; \\
& \quad f := h - f; \\
& \quad i := i - 1 \\
\text{end} \\
\mid f = \text{fib}(n) \rangle
\end{align*}
\]

Invariant:
\[f = \text{fib}(n-i+1) \land h = \text{fib}(n-i+2) \land i > 0\]

Termination function: \(i\)

Question 1: does
\[f = \text{fib}(n-i+1) \land h = \text{fib}(n-i+2) \land i > 0 \land i > 1\] imply \(i > 0\) ? Yes

Question 2: can we derive
\[
\langle \ ... \ i = k \mid \\
h := h+f; f := h-f; i := i-1 \\
\mid ... \ i < k \rangle\] ? Yes
Example: I/O

\[ \langle \text{IN} = <1,2,\ldots,100> \land \text{OUT} = <> \mid \]
read x;
while x \neq 100 do
  write x;
  read x
end
| OUT = <1,2,\ldots,99> \rangle

Invariant:
OUT^x^\text{IN} = <1,2,\ldots,100>

Termination function:
\text{length}(\text{IN})

Question 1: does
OUT^x^\text{IN} = <1,2,\ldots,100> \\
\land x \neq 100 \text{ imply } \\
\text{length}(\text{IN}) > 0 \ ? \ Yes

Question 2: can we derive
\langle \ldots \text{length}(\text{IN}) = k \mid \\
\text{write } x; \text{ read } x \\
\mid \ldots \text{length}(\text{IN}) < k \rangle \ ? \ Yes
Another Progress Function

\[ \langle s = 0 \land x = 0 \mid \text{while } x \neq 10 \text{ do } x := x + 1; \ s := s + x \text{ end} \mid s = \sum_{k=0}^{10} k \rangle \]

Invariant: \( 0 \leq x \leq 10 \land s = \sum_{k=0}^{x} k \)

Termination function: \( 10 - x \)
Other Total Correctness Rules

Essentially identical: e.g.

$$\langle p \mid S_1 \mid q \rangle \quad \langle q \mid S_2 \mid r \rangle$$

$$\frac{\langle p \mid S_1 ; S_2 \mid r \rangle}{\langle p \mid S_1 \mid q \rangle \quad \langle q \mid S_2 \mid r \rangle}$$
Summary: Axiomatic Semantics

**Assertions** are first-order logic formulas used to express sets of possible states

**Hoare triples (results)** express partial (total) correctness conditions

**Proof rules** used to define axiomatic semantics

Must be **sound (consistent)** and **complete** relative to the operational model
Program Verification

Given an already defined axiomatic semantics, we can try to prove partial or total correctness

- \( S \) is a program fragment, \( p \) is something we can guarantee,
- \( q \) is something we want \( S \) to achieve

**Specification:** \( \{ p \} S \{ q \} \) and/or \( \langle p \mid S \mid q \rangle \)

- Widely used in software eng., even without formal proofs

**Verification:** if we find a proof, \( S \) is correct

- A counter-example uncovers a bug
- How to do this with software tools rather than manually?

**Machinery**

- Need to find loop invariants
- Backward substitution across multiple assignments
- Termination function for proving total correctness