Program Analysis and Verification

0368-4479

http://www.cs.tau.ac.il/~maon/teaching/2013-2014/paav/paav1314b.html

Noam Rinetzky

Lecture 5: Aximatic Semantics

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

Good manners

- Mobiles

Home Work Assignment #1

In the following, we refer to the "Semantics with Application" book as "the book". The book can be found here: http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html.

1. Solve Ex 2.8 and 2.18 in the book.
2. In the previous question, you were asked to extend the While language with a new construct (a for loop). Extend the proof of theorem 2.26 in the book (semantic equivalence) to handle for commands.
3. Solve Ex 2.34 in the book.
4. Read Section 2.5 in the book and solve Ex 2.45.
5. Prove or disprove: The denotational semantics of any statement in the While language shown in the lectures is a monotone and continuous function.
6. Define a denotational semantics for the the While language extended with the random command. (The extension is described in Question 3) .

Denotational Sematnics

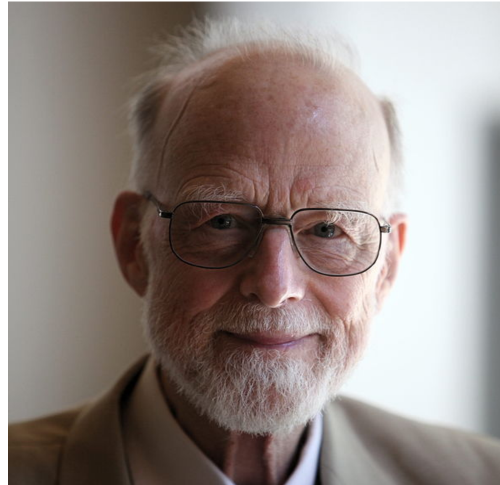
- Added examples
- Equivalences of operational and denotational semantics

- Complete lattices
- Tarski-Kantor Fixed-point theorem

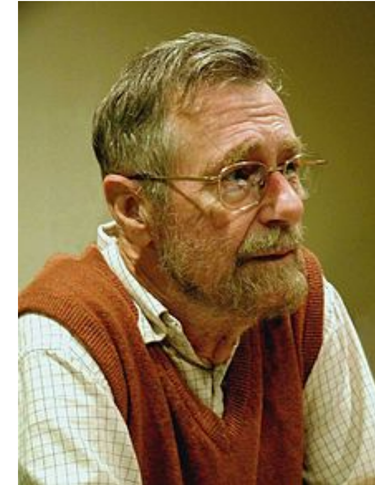
Axiomatic Semantics

Robert Floyd

C.A.R. Hoare



Edsger W. Dijkstra



Proving program correctness

- Why prove correctness?
- What is correctness?
- How?
  - Reasoning at the operational semantics level
    - Tedious
    - Error prone
  - Formal reasoning using "axiomatic" semantics
    - Syntactic technique ("game of tokens")
    - Mechanically checkable
      - Sometimes automatically derivable

A simple imperative language: **While**

Abstract syntax:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$$

$$b ::= \texttt{true} \mid \texttt{false}$$
$$\mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

$$S ::= x \texttt{ := } a \mid \texttt{skip} \mid S_1 \texttt{;}\ S_2$$
$$\mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$
$$\mid \texttt{while}\ b\ \texttt{do}\ S$$

Program correctness concepts

- Property = a certain relationship between initial state and final state

  Other notions of properties exist

- Partial correctness = properties that hold ***if*** program terminates

  Mostly focus in this course

- Termination = program always terminates
  - i.e., for every input state

partial correctness + termination = total correctness

Other correctness concepts exist: resource usage, linearizability, …

Factorial example

$S_{fac} \equiv$ `y := 1; while ¬(x=1) do (y := y*x; x := x-1)`

- Factorial partial correctness property =
  - *if* the statement terminates *then* the final value of **`y`** will be the factorial of the initial value of **`x`**
    - What if s x < 0?
- Formally, using natural semantics:
  $\langle S_{fac}, s\rangle \longrightarrow s'$ implies s' **`y`** = (s **`x`**)!

Natural semantics for **While**

$[\text{ass}_{\text{ns}}]$ $\langle \texttt{x} := a, s\rangle \rightarrow s[\texttt{x} \mapsto \mathcal{A}[\![a]\!]s]$

$[\text{skip}_{\text{ns}}]$ $\langle \texttt{skip}, s\rangle \rightarrow s$

$[\text{comp}_{\text{ns}}]$ $$\frac{\langle S_1, s\rangle \rightarrow s',\ \langle S_2, s'\rangle \rightarrow s''}{\langle S_1; S_2, s\rangle \rightarrow s''}$$

$[\text{if}^{\text{tt}}_{\text{ns}}]$ $$\frac{\langle S_1, s\rangle \rightarrow s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \rightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]\, s = \mathbf{tt}$$

$[\text{if}^{\text{ff}}_{\text{ns}}]$ $$\frac{\langle S_2, s\rangle \rightarrow s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \rightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]\, s = \mathbf{ff}$$

$[\text{while}^{\text{ff}}_{\text{ns}}]$ $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow s$ if $\mathcal{B}[\![b]\!]\, s = \mathbf{ff}$

$[\text{while}^{\text{tt}}_{\text{ns}}]$ $$\frac{\langle S, s\rangle \rightarrow s',\ \langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \rightarrow s''}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow s''} \quad \text{if } \mathcal{B}[\![b]\!]\, s = \mathbf{tt}$$

Staged proof

The proof proceeds in three stages:

Stage 1: We prove that the body of the `while` loop satisfies:

$$\begin{array}{l}\text{if } \langle \mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1,\ s\rangle \to s'' \text{ and } s''\ \mathtt{x} > \mathbf{0} \\ \text{then } (s\ \mathtt{y}) \star (s\ \mathtt{x})! = (s''\ \mathtt{y}) \star (s''\ \mathtt{x})! \text{ and } s\ \mathtt{x} > \mathbf{0}\end{array} \qquad (*)$$

Stage 2: We prove that the `while` loop satisfies:

$$\begin{array}{l}\text{if } \langle \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1),\ s\rangle \to s'' \\ \text{then } (s\ \mathtt{y}) \star (s\ \mathtt{x})! = s''\ \mathtt{y} \text{ and } s''\ \mathtt{x} = \mathbf{1} \text{ and } s\ \mathtt{x} > \mathbf{0}\end{array} \qquad (**)$$

Stage 3: We prove the partial correctness property for the complete program:

$$\begin{array}{l}\text{if } \langle \mathtt{y} := 1;\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1),\ s\rangle \to s' \\ \text{then } s'\ \mathtt{y} = (s\ \mathtt{x})! \text{ and } s\ \mathtt{x} > \mathbf{0}\end{array} \qquad (***)$$

In each of the three stages the derivation tree of the given transition is inspected in order to prove the property.

First stage

Stage 1: We prove that the body of the `while` loop satisfies:

$$\begin{array}{l} \text{if } \langle \texttt{y := y}\star\texttt{x; x := x}-1,\ s\rangle \to s'' \text{ and } s''\ \texttt{x} > \mathbf{0} \\ \text{then } (s\ \texttt{y}) \star (s\ \texttt{x})! = (s''\ \texttt{y}) \star (s''\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0} \end{array} \qquad (*)$$

In the *first stage* we consider the transition

$$\langle \texttt{y := y}\star\texttt{x; x := x}-1,\ s\rangle \to s''$$

According to [comp$_{\text{ns}}$] there will be transitions

$$\langle \texttt{y := y}\star\texttt{x},\ s\rangle \to s' \text{ and } \langle \texttt{x := x}-1,\ s'\rangle \to s''$$

for some s'. From the axiom [ass$_{\text{ns}}$] we then get that $s' = s[\texttt{y}\mapsto\mathcal{A}[\![\texttt{y}\star\texttt{x}]\!]s]$ and that $s'' = s'[\texttt{x}\mapsto\mathcal{A}[\![\texttt{x}-1]\!]s']$. Combining these results we have

$$s'' = s[\texttt{y}\mapsto(s\ \texttt{y})\star(s\ \texttt{x})][\texttt{x}\mapsto(s\ \texttt{x})-1]$$

Assuming that $s''\ \texttt{x} > \mathbf{0}$ we can then calculate

$$(s''\ \texttt{y}) \star (s''\ \texttt{x})! = ((s\ \texttt{y}) \star (s\ \texttt{x})) \star ((s\ \texttt{x})-1)! = (s\ \texttt{y}) \star (s\ \texttt{x})!$$

and since $s\ \texttt{x} = (s''\ \texttt{x}) + \mathbf{1}$ this shows that (*) does indeed hold.

Second stage

Stage 2: We prove that the `while` loop satisfies:

$$\text{if } \langle \texttt{while } \neg(\texttt{x=1}) \texttt{ do } (\texttt{y := y}\star\texttt{x; x := x}-1),\ s\rangle \to s'' \\ \text{then } (s\ \texttt{y}) \star (s\ \texttt{x})! = s''\ \texttt{y} \text{ and } s''\ \texttt{x} = \mathbf{1} \text{ and } s\ \texttt{x} > \mathbf{0} \qquad (**)$$

In the *second stage* we proceed by induction on the shape of the derivation tree for

$$\langle \texttt{while } \neg(\texttt{x=1}) \texttt{ do } (\texttt{y := y}\star\texttt{x; x := x}-1),\ s\rangle \to s'$$

$\langle$`while ¬(x=1) do (y := y*x; x := x-1)`$, s\rangle \rightarrow s'$

One of two axioms and rules could have been used to construct this derivation. If $[\text{while}^{\text{ff}}_{\text{ns}}]$ has been used then $s' = s$ and $\mathcal{B}[\![\neg(\texttt{x}{=}1)]\!]s = \mathbf{ff}$. This means that $s'\ \texttt{x} = \mathbf{1}$ and since $\mathbf{1}! = \mathbf{1}$ we get the required $(s\ \texttt{y}) \star (s\ \texttt{x})! = s\ \texttt{y}$ and $s\ \texttt{x} > \mathbf{0}$. This proves (**).

Next assume that $[\text{while}^{\text{tt}}_{\text{ns}}]$ is used to construct the derivation. Then it must be the case that $\mathcal{B}[\![\neg(\texttt{x}{=}1)]\!]s = \mathbf{tt}$ and

$$\langle \texttt{y := y}\star\texttt{x; x := x}-\texttt{1},\ s\rangle \rightarrow s''$$

and

$$\langle \texttt{while}\ \neg(\texttt{x}{=}1)\ \texttt{do}\ (\texttt{y := y}\star\texttt{x; x := x}-\texttt{1}),\ s''\rangle \rightarrow s'$$

for some state s''. The induction hypothesis applied to the latter derivation gives that

$$(s''\ \texttt{y}) \star (s''\ \texttt{x})! = s'\ \texttt{y} \text{ and } s'\ \texttt{x} = \mathbf{1} \text{ and } s''\ \texttt{x} > \mathbf{0}$$

From (*) we get that

$$(s\ \texttt{y}) \star (s\ \texttt{x})! = (s''\ \texttt{y}) \star (s''\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}$$

Putting these results together we get

$$(s\ \texttt{y}) \star (s\ \texttt{x})! = s'\ \texttt{y} \text{ and } s'\ \texttt{x} = \mathbf{1} \text{ and } s\ \texttt{x} > \mathbf{0}$$

This proves (**) and thereby the second stage of the proof is completed.

Third stage

Stage 3: We prove the partial correctness property for the complete program:

$$\text{if } \langle \texttt{y := 1; while } \neg(\texttt{x=1}) \texttt{ do (y := y}\star\texttt{x; x := x}-\texttt{1)}, s\rangle \to s' \qquad (***)$$
$$\text{then } s'\ \texttt{y} = (s\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}$$

Finally, consider the *third stage* of the proof and the derivation

$$\langle \texttt{y := 1; while } \neg(\texttt{x=1}) \texttt{ do (y := y}\star\texttt{x; x := x}-\texttt{1)}, s\rangle \to s'$$

According to $[\text{comp}_{\text{ns}}]$ there will be a state s'' such that

$$\langle \texttt{y := } \mathbf{1}, s\rangle \to s''$$

and

$$\langle \texttt{while } \neg(\texttt{x=1}) \texttt{ do (y := y}\star\texttt{x; x := x}-\texttt{1)}, s''\rangle \to s'$$

From axiom $[\text{ass}_{\text{ns}}]$ we see that $s'' = s[\texttt{y}\mapsto\mathbf{1}]$ and from (**) we get that $s''\ \texttt{x} > \mathbf{0}$ and therefore $s\ \texttt{x} > \mathbf{0}$. Hence $(s\ \texttt{x})! = (s''\ \texttt{y}) \star (s''\ \texttt{x})!$ holds and using (**) we get

$$(s\ \texttt{x})! = (s''\ \texttt{y}) \star (s''\ \texttt{x})! = s'\ \texttt{y}$$

as required. This proves the partial correctness of the factorial statement.

How easy was that?

- Proof is very laborious
  - Need to connect all transitions and argues about relationships between their states
  - Reason: too closely connected to semantics of programming language

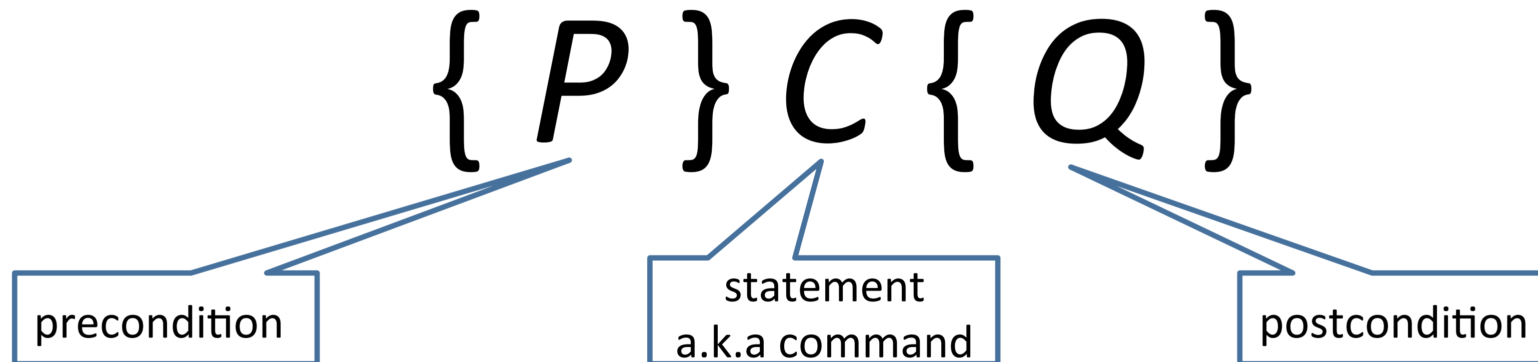
- Is the proof correct?

- How did we know to find this proof?
  - Is there a methodology?

Axiomatic verification approach

- What do we need in order to prove that the program does what it supposed to do?
- Specify the required behavior
- Compare the behavior with the one obtained by the operational semantics
- Develop a proof system for showing that the program satisfies a requirement
- Mechanically use the proof system to show correctness
- The meaning of a program is a set of verification rules

Assertions, a.k.a Hoare triples



- *P* and *Q* are state predicates
  - Example: **`x`**>0
- ***If*** *P* holds in the initial state, and
  if execution of *C* terminates on that state,
  then *Q* will hold in the state in which *C* halts
- *C* is not required to always terminate

```
{true} while true do skip {false}
```

Total correctness assertions

$$[\ P\ ]\ C\ [\ Q\ ]$$

- ***If*** *P* holds in the initial state,
  execution of *C* ***must terminate*** on that state,
  and *Q* will hold in the state in which *C* halts

Factorial example

{ ? }

```
y := 1; while ¬(x=1) do (y := y*x; x := x–1)
```

{ ? }

First attempt

We need a way to "remember" value of x before execution

$\{\, x>0 \,\}$

```
y := 1; while ¬(x=1) do (y := y*x; x := x-1)
```

$\{\, y=x! \,\}$

Holds only for value of x at state after execution finishes

Fixed assertion

A logical variable, must not appear in statement - immutable

$\{ \mathbf{x}=n \}$

```
y := 1; while ¬(x=1) do (y := y*x; x := x-1)
```

$\{ \mathbf{y}=n! \wedge n>0 \}$

The proof outline

```
{ x=n }
 y := 1;
{ x>0 ⇒ y*x!=n! ∧ n≥x }
while ¬(x=1) do
   { x-1>0 ⇒ (y*x)*(x-1)!=n! ∧ n≥(x-1) }
   y := y*x;
   { x-1>0 ⇒ y*(x-1)!=n! ∧ n≥(x-1) }
   x := x-1
{ y*x!=n! ∧ n>0 ∧ x=1 }
```

Factorial example

$S_{fac} \equiv$ `y := 1; while ¬(x=1) do (y := y*x; x := x-1)`

- Factorial partial correctness property = *if* the statement terminates *then* the final value of **`y`** will be the factorial of the initial value of **`x`**
  - What if s x < 0?
- Formally, using natural semantics: $\langle S_{fac}, s\rangle \longrightarrow s'$ implies s' **`y`** = (s **`x`**)!

Staged proof

The proof proceeds in three stages:

Stage 1: We prove that the body of the `while` loop satisfies:

$$\begin{array}{l}\text{if } \langle \mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1,\ s\rangle \to s'' \text{ and } s''\ \mathtt{x} > \mathbf{0} \\ \text{then } (s\ \mathtt{y}) \star (s\ \mathtt{x})! = (s''\ \mathtt{y}) \star (s''\ \mathtt{x})! \text{ and } s\ \mathtt{x} > \mathbf{0}\end{array} \qquad (*)$$

Stage 2: We prove that the `while` loop satisfies:

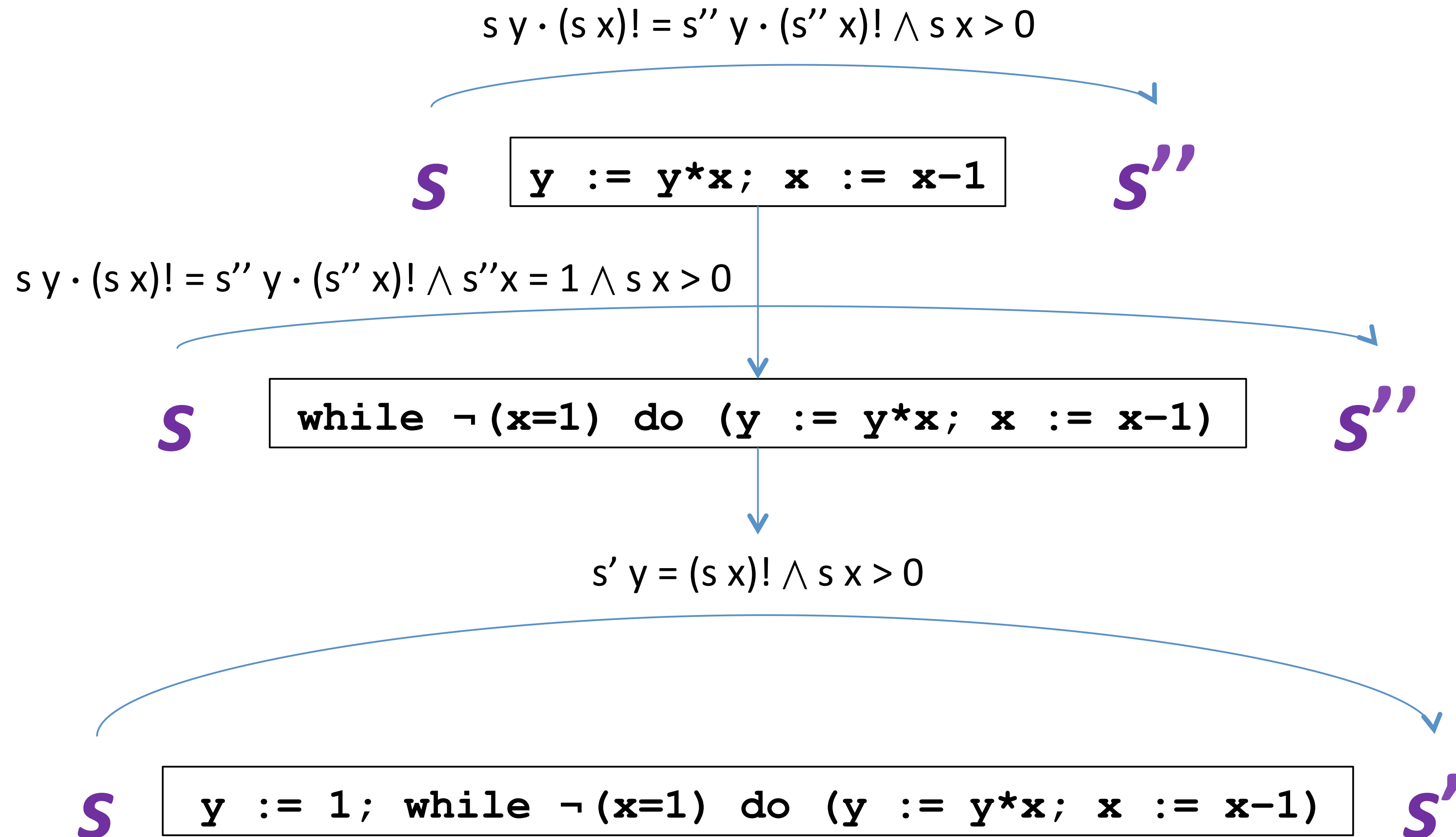
$$\begin{array}{l}\text{if } \langle \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1),\ s\rangle \to s'' \\ \text{then } (s\ \mathtt{y}) \star (s\ \mathtt{x})! = s''\ \mathtt{y} \text{ and } s''\ \mathtt{x} = \mathbf{1} \text{ and } s\ \mathtt{x} > \mathbf{0}\end{array} \qquad (**)$$

Stage 3: We prove the partial correctness property for the complete program:

$$\begin{array}{l}\text{if } \langle \mathtt{y} := 1;\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1),\ s\rangle \to s' \\ \text{then } s'\ \mathtt{y} = (s\ \mathtt{x})! \text{ and } s\ \mathtt{x} > \mathbf{0}\end{array} \qquad (***)$$

In each of the three stages the derivation tree of the given transition is inspected in order to prove the property.

Stages



Inductive proof over iterations

$s\ y \cdot (s\ x)! = s'\ y \cdot (s'\ x)! \wedge s\ x > 0$

s `(y := y*x; x := x-1)` *s'*

s' `while ¬(x=1) do (y := y*x; x := x-1)` *s''*

$s'\ y \cdot (s'\ x)! = s''\ y \cdot (s''\ x)! \wedge s''x = 1 \wedge s'\ x > 0$

---

s `while ¬(x=1) do (y := y*x; x := x-1)` *s''*

$s\ y \cdot (s\ x)! = s''\ y \cdot (s''\ x)! \wedge s''x = 1 \wedge s\ x > 0$

Assertions, a.k.a Hoare triples

$$\{\, P \,\}\; C \;\{\, Q \,\}$$

- *P* and *Q* are state predicates
  - Example: **`x`**>0
- ***If*** *P* holds in the initial state, and
  if execution of *C* terminates on that state,
  then *Q* will hold in the state in which *C* halts
- *C* is not required to always terminate

```
{true} while true do skip {false}
```

Total correctness assertions

$$[\ P\ ]\ C\ [\ Q\ ]$$

- ***If*** *P* holds in the initial state,
execution of *C* ***must terminate*** on that state,
and *Q* will hold in the state in which *C* halts

Factorial assertion

A logical variable, must not appear in statement - immutable

$$\{ \mathbf{x}=n \}$$

```
y := 1; while ¬(x=1) do (y := y*x; x := x-1)
```

$$\{ \mathbf{y}=n! \wedge n>0 \}$$

Factorial partial correctness proof

{ **x**=n }

```
y := 1;
```

{ **x**>0 ⇒ **y*****x**!=n! ∧ n≥x }

```
while ¬(x=1) do
```

  { **x-1**>0 ⇒ (**y*x**) *(**x-1**) !=n! ∧ n≥(x-1) }

```
  y := y*x;
```

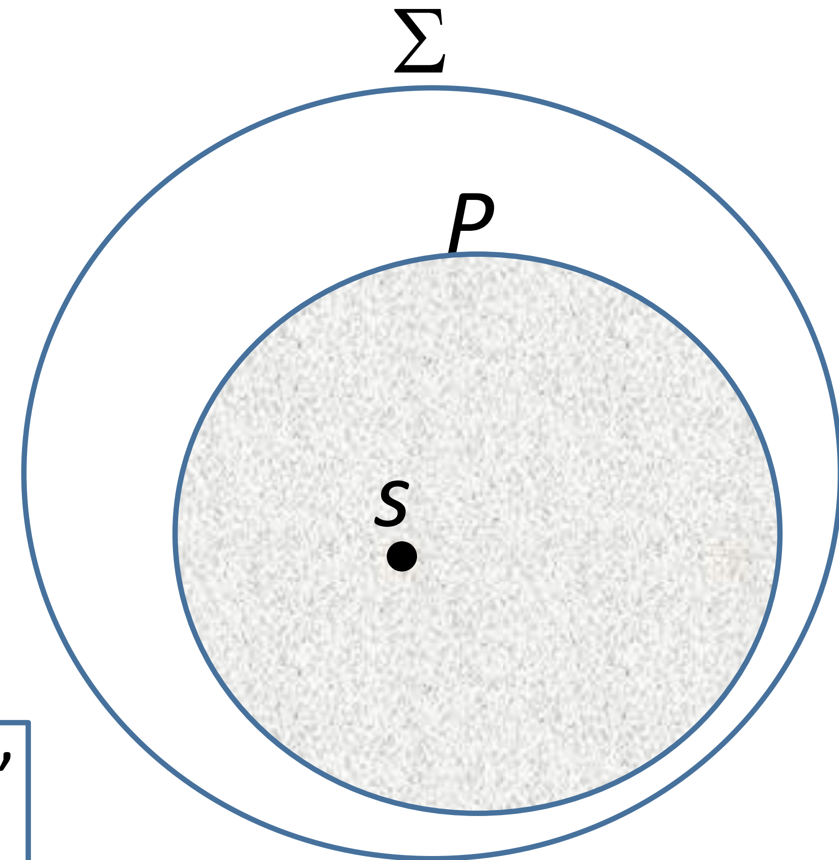
  { **x-1**>0 ⇒ **y***(**x-1**) !=n! ∧ n≥(x-1) }

```
  x := x-1
```

{ **y*****x**!=n! ∧ **n**>0 ∧ **x**=1 }

Formalizing partial correctness

- $s \models P$
  - P holds in state s
- Σ – program states
  $\bot$ – undefined

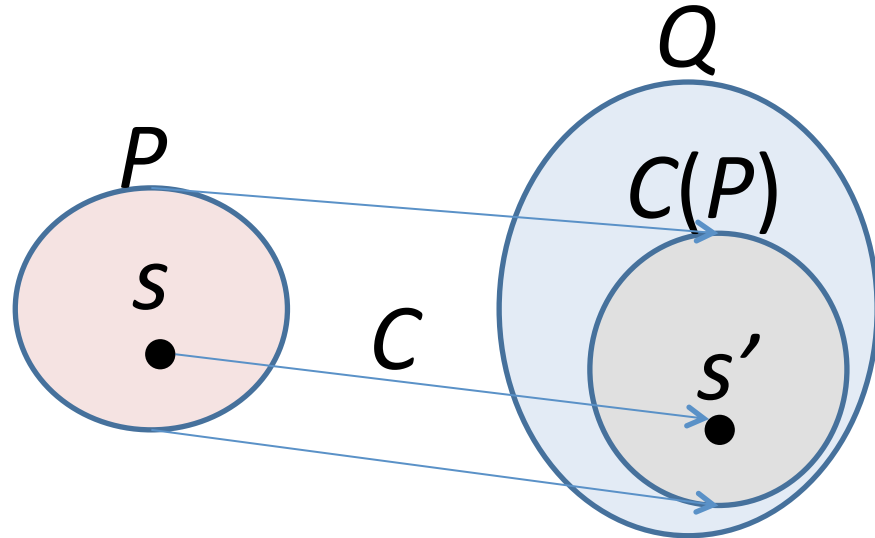


$$S_{ns} [\![C]\!] s = \begin{cases} s' & \text{if } \langle C, s \rangle \rightarrow s' \\ \bot & \text{else} \end{cases}$$

Formalizing partial correctness

- $s \models P$
  - P holds in state s
- Σ – program states
  $\bot$ – undefined
- $\{ P \}\ C\ \{ Q \}$
  - $\forall s, s' \in \Sigma\ .\ (s \models P \land \langle C, s\rangle \rightarrow s') \Rightarrow s' \models Q$

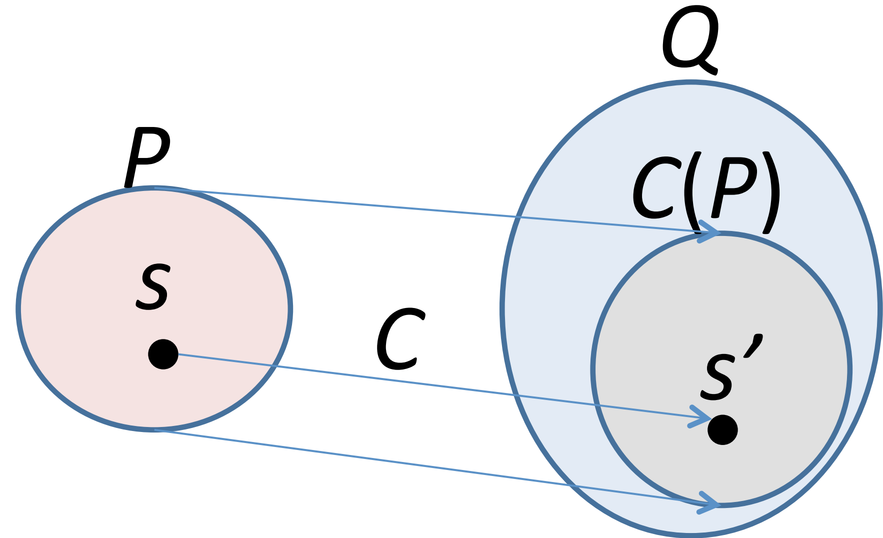
    alternatively
  - $\forall s \in \Sigma\ .\ (s \models P \land S_{ns}[\![C]\!]\ s \neq \bot) \Rightarrow S_{ns}[\![C]\!] \models Q$
  - Convention: $\bot \models P$ for all P
    $\forall s \in \Sigma\ .\ s \models P \Rightarrow S_{ns}[\![C]\!]\ s \models Q$



Why did we choose natural semantics?

Formalizing partial correctness

- $s \models P$
  - P holds in state s
- Σ – program states
  $\bot$ – undefined
- $\{ P \} C \{ Q \}$
  - $\forall s, s' \in \Sigma \,.\, (s \models P \land \langle C, s \rangle \Rightarrow^* s') \Rightarrow s' \models Q$
    alternatively
  - $\forall s \in \Sigma \,.\, (s \models P \land S_{sos}[\![C]\!]\, s \neq \bot) \Rightarrow S_{sos}[\![C]\!] \models Q$
  - Convention: $\bot \models P$ for all P
    $\forall s \in \Sigma \,.\, s \models P \Rightarrow S_{sos}[\![C]\!]\, s \models Q$



How do we express predicates?

- Extensional approach
  - Abstract mathematical functions
    P : **State** $\rightarrow$ **T**
- Intensional approach
  - Via language of formulae

An assertion language

- **Bexp** is not expressive enough to express predicates needed for many proofs
  - Extend **Bexp**
- Allow quantifications
  - $\forall z. \ldots$
  - $\exists z. \ldots$
    - $\exists z.\ z = k \times n$
- Import well known mathematical concepts
  - $n! \equiv n \times (n-1) \times \cdots 2 \times 1$

An assertion language

Either a program variables or a logical variable

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$$

$$A ::= \texttt{true} \mid \texttt{false}$$
$$\mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2$$
$$\mid A_1 \Rightarrow A_2 \mid \forall z.\ A \mid \exists z.\ A$$

First Order Logic Reminder

Free/bound variables

- A variable is said to be bound in a formula when it occurs in the scope of a quantifier. Otherwise it is said to be free
  - $\exists i.\ k=i\times m$
  - $(i+100\leq 77)\wedge\forall i.\ j+1=i+3)$
- FV(*A*) ≡ the free variables of *A*
- Defined inductively on the abstract syntax tree of *A*

Free variables

$\text{FV}(n) \equiv \{\}$

$\text{FV}(x) \equiv \{\text{x}\}$

$\text{FV}(a_1\text{+}a_2) \equiv \text{FV}(a_1\bigstar a_2) \equiv \text{FV}(a_1\text{-}a_2) \equiv \text{FV}(a_1) \cup \text{FV}(a_2)$

$\text{FV}(\mathbf{true}) \equiv \text{FV}(\mathbf{false}) \equiv \{\}$

$\text{FV}(a_1\text{=}a_2) \equiv \text{FV}(a_1 \leq a_2) \equiv \text{FV}(a_1) \cup \text{FV}(a_2)$

$\text{FV}(\neg A) \equiv \text{FV}(A)$

$\text{FV}(A_1 \land A_2) \equiv \text{FV}(A_1 \lor A_2) \equiv \text{FV}(A_1 \Rightarrow A_2)$

$\text{FV}(\forall z.\ A) \equiv \text{FV}(\exists z.\ A) \equiv \text{FV}(A) \setminus \{z\}$

Substitution

What if t is not pure?

- An expression t is pure (a term) if it does not contain quantifiers
- $A[t/z]$ denotes the assertion A' which is the same as A, except that all instances of the free variable z are replaced by t
- $A \equiv \exists i.\ k = i \times m$
  $A[5/k] =$
  $A[5/i] =$

Calculating substitutions

$n[t/z] = n$
$x[t/z] = x$
$x[t/x] = t$

$(a_1 + a_2)[t/z] \quad = a_1[t/z] + a_2[t/z]$
$(a_1 \star a_2)[t/z] \quad = a_1[t/z] \star a_2[t/z]$
$(a_1 - a_2)[t/z] \quad = a_1[t/z] - a_2[t/z]$

Calculating substitutions

$\mathbf{true}[t/x] = \mathbf{true}$

$\mathbf{false}[t/x] = \mathbf{false}$

$(a_1 = a_2)[t/z] = a_1[t/z] = a_2[t/z]$

$(a_1 \leq a_2)[t/z] = a_1[t/z] \leq a_2[t/z]$

$(\neg A)[t/z] = \neg(A[t/z])$

$(A_1 \wedge A_2)[t/z] = A_1[t/z] \wedge A_2[t/z]$

$(A_1 \vee A_2)[t/z] = A_1[t/z] \vee A_2[t/z]$

$(A_1 \Rightarrow A_2)[t/z] = A_1[t/z] \Rightarrow A_2[t/z]$

$(\forall z.\ A)[t/z] = \forall z.\ A$

$(\forall z.\ A)[t/y] = \forall z.\ A[t/y]$

$(\exists\ z.\ A)[t/z] = \exists\ z.\ A$

$(\exists\ z.\ A)[t/y] = \exists\ z.\ A[t/y]$

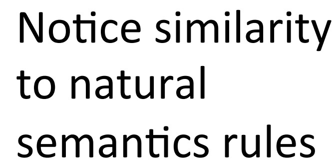
Proof Rules

Axiomatic semantics for **While**

[ass$_p$] $\{ P[a/\mathtt{x}] \}\ \mathtt{x} := a\ \{ P \}$

[skip$_p$] $\{ P \}\ \mathtt{skip}\ \{ P \}$

[comp$_p$] $$\frac{\{ P \}\ S_1\ \{ Q \}, \quad \{ Q \}\ S_2\ \{ R \}}{\{ P \}\ S_1 ;\ S_2\ \{ R \}}$$



[if$_p$] $$\frac{\{ b \wedge P \}\ S_1\ \{ Q \}, \quad \{ \neg b \wedge P \}\ S_2\ \{ Q \}}{\{ P \}\ \mathtt{if}\ \ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ \{ Q \}}$$

[while$_p$] $$\frac{\{ b \wedge P \}\ S\ \{ P \}}{\{ P \}\ \mathtt{while}\ \ b\ \mathtt{do}\ S\ \{ \neg b \wedge P \}}$$

[cons$_p$] $$\frac{\{ P' \}\ S\ \{ Q' \}}{\{ P \}\ S\ \{ Q \}}$$ if $P \Rightarrow P'$ and $Q' \Rightarrow Q$

Assignment rule

$$[\text{ass}_p] \quad \{\, P[a/\mathtt{x}] \,\}\ \mathtt{x} := a\ \{\, P \,\}$$

$s[\mathrm{x}\mapsto\mathcal{A}[\![a]\!]s] \models P$

- A "backwards" rule
- $\mathtt{x} := a$ always finishes
- Why is this true?
  - Recall operational semantics:

$$[\text{ass}_{ns}] \quad \langle \mathtt{x} := a, s\rangle \longrightarrow s[\mathrm{x}\mapsto\mathcal{A}[\![a]\!]s]$$

- Example: **{y*z<9} x:=y*z {x<9}**
  What about **{y*z<9∧w=5} x:=y*z {w=5}**?

`skip` rule

[skip_p] $\{ P \}\ \texttt{skip}\ \{ P \}$

[skip_{ns}] $\langle \texttt{skip}, s \rangle \rightarrow s$

Composition rule

[comp$_p$]
$$\frac{\{ P \}\, S_1 \,\{ Q \}, \quad \{ Q \}\, S_2 \,\{ R \}}{\{ P \}\, S_1; S_2 \,\{ R \}}$$

[comp$_{ns}$]
$$\frac{\langle S_1, s\rangle \longrightarrow s', \langle S_2, s'\rangle \longrightarrow s''}{\langle S_1; S_2, s\rangle \longrightarrow s''}$$

- Holds when S_1 terminates in every state where P holds and then Q holds
  and S_2 terminates in every state where Q holds and then R holds

Condition rule

[if$_p$]

$$\frac{\{\, b \land P \,\}\, S_1 \,\{\, Q \,\},\ \ \{\, \neg b \land P \,\}\, S_2 \,\{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ \ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\, Q \,\}}$$

[if$^{tt}{}_{ns}$]

$$\frac{\langle S_1, s\rangle \longrightarrow s'}{\langle \texttt{if}\ \ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \longrightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]\, s = \mathbf{tt}$$

[if$^{ff}{}_{ns}$]

$$\frac{\langle S_2, s\rangle \longrightarrow s'}{\langle \texttt{if}\ \ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \longrightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]\, s = \mathbf{ff}$$

Loop rule

$$[\text{while}_p] \quad \frac{\{\, b \land P \,\}\, S \,\{\, P \,\}}{\{\, P \,\}\ \texttt{while}\ \ b\ \texttt{do}\ S\ \{\neg b \land P\,\}}$$

$$[\text{while}^{ff}_{ns}] \quad \langle \texttt{while}\ \ b\ \texttt{do}\ S, s\rangle \rightarrow s \qquad \text{if } \mathcal{B}[\![b]\!]\, s = \mathbf{ff}$$

$$[\text{while}^{tt}_{ns}] \quad \frac{\langle S, s\rangle \rightarrow s',\ \langle \texttt{while}\ \ b\ \texttt{do}\ S, s'\rangle \rightarrow s''}{\langle \texttt{while}\ \ b\ \texttt{do}\ S, s\rangle \rightarrow s''} \qquad \text{if } \mathcal{B}[\![b]\!]\, s = \mathbf{tt}$$

- Here *P* is called an invariant for the loop
  - Holds before and after each loop iteration
  - Finding loop invariants – most challenging part of proofs
- When loop finishes, *b* is false

Rule of consequence

$$[\text{cons}_p] \quad \frac{\{\, P' \,\}\ S\ \{\, Q' \,\}}{\{\, P \,\}\ S\ \{\, Q \,\}} \quad \text{if } P \Rightarrow P' \text{ and } Q' \Rightarrow Q$$

- Allows strengthening the precondition and weakening the postcondition
- The only rule that is not sensitive to the form of the statement

Rule of consequence

$$[\text{cons}_p] \quad \frac{\{\, P' \,\}\, S \,\{\, Q' \,\}}{\{\, P \,\}\, S \,\{\, Q \,\}} \quad \text{if } P \Rightarrow P' \text{ and } Q' \Rightarrow Q$$

- Why do we need it?
- Allows the following

$$\frac{\texttt{\{y*z<9\} x:=y*z \{x<9\}}}{\texttt{\{y*z<9}\wedge\texttt{w=5\} x:=y*z \{x<10\}}}$$

Inference trees

- Similar to derivation trees of natural semantics
- Leaves are …
- Internal nodes correspond to …
- Inference tree is called
  - Simple if tree is only an axiom
  - Composite otherwise

Provability

- We say that an assertion $\{ P \}\ C\ \{ Q \}$ is provable if there exists an inference tree
  - Written as $\vdash_p \{ P \}\ C\ \{ Q \}$
  - Are inference trees unique?
    {true} x:=1; x:=x+5 {x≥0}
- Proofs of properties of axiomatic semantics use *induction on the shape of the inference tree*
  - Example: prove $\vdash_p \{ P \}\ C\ \{$ **`true`** $\}$ for any P and C

Factorial proof

Goal: { x=n } `y:=1; while (x≠1) do (y:=y*x; x:=x-1)` { $y=n! \land n>0$ }

W = `while (x≠1) do (y:=y*x; x:=x-1)`

INV = $x > 0 \Rightarrow (y \cdot x! = n! \land n \geq x)$

[comp]
{ INV[x-1/x][y*x/y] } `y:=y*x` { INV[x-1/x] } { INV[x-1/x] } `x:=x-1` {INV}
—
{ INV[x-1/x][y*x/y] } `y:=y*x; x:=x-1` {INV}

[cons]
—
{`x≠1` ∧ INV } `y:=y*x; x:=x-1` { INV }

[while]
—
{ INV } W {x=1 ∧ INV }

[cons]
—
{ INV } W { $y=n! \land n>0$ }

[cons]
{ INV[1/y] } `y:=1` { INV }
—
{ x=n } `y:=1` { INV }

[comp]
—
{ x=n } `while (x≠1) do (y:=y*x; x:=x-1)` { $y=n! \land n>0$ }

Annotated programs

- A streamlined version of inference trees
  - Inline inference trees into programs
  - A kind of "proof carrying code"
  - Going from annotated program to proof tree is a linear time translation

Annotating composition

- We can inline inference trees into programs
- Using proof equivalence of S_1; (S_2; S_3) and (S_1; S_2); S_3 instead writing deep trees, e.g.,

$$\dfrac{\dfrac{\{P\}\ S_1\ \{P'\} \quad \{P'\}\ S_2\ \{P''\}}{\{P\}\ (S_1;\ S_2)\ \{P''\}} \qquad \dfrac{\{P''\}\ S_3\ \{P'''\} \quad \{P'''\}\ S_4\ \{P''\}}{\{P''\}\ (S_3\ ;\ S_4)\ \{Q\}}}{\{P\}\ (S_1;\ S_2);\ (S_3\ ;\ S_4)\ \{Q\}}$$

- We can annotate a composition S_1; S_2;…; S_n by

$$\{P_1\}\ S_1\ \{P_2\}\ S_2 \ldots \{P_{n-1}\}\ S_{n-1}\ \{P_n\}$$

Annotating conditions

$$[\text{if}_p]\ \frac{\{\, b \wedge P \,\}\, S_1 \,\{\, Q \,\}, \quad \{\, \neg b \wedge P \,\}\, S_2 \,\{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ \ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\, Q \,\}}$$

$\{\, P \,\}$
`if` b `then`
    $\{\, b \wedge P \,\}$
    S_1
`else`
    S_2
$\{\, Q \,\}$

Annotating conditions

$$[\text{if}_p]\quad \frac{\{\,b \wedge P\,\}\,S_1\,\{\,Q\,\},\quad \{\,\neg b \wedge P\,\}\,S_2\,\{\,Q\,\}}{\{\,P\,\}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\,Q\,\}}$$

$\{ P \}$
`if` b `then`
    $\{ b \wedge P \}$
    S_1
    $\{ Q_1 \}$
`else`
    S_2
    $\{ Q_2 \}$
$\{ Q \}$

Usually Q is the result of using the consequence rule, so a more explicit annotation is

Annotating loops

$$[\text{while}_p] \quad \frac{\{\, b \wedge P \,\}\, S \,\{\, P \,\}}{\{\, P \,\}\, \texttt{while}\ b\ \texttt{do}\ S \,\{\neg b \wedge P \,\}}$$

$\{\, P \,\}$

$\texttt{while}\ b\ \texttt{do}$

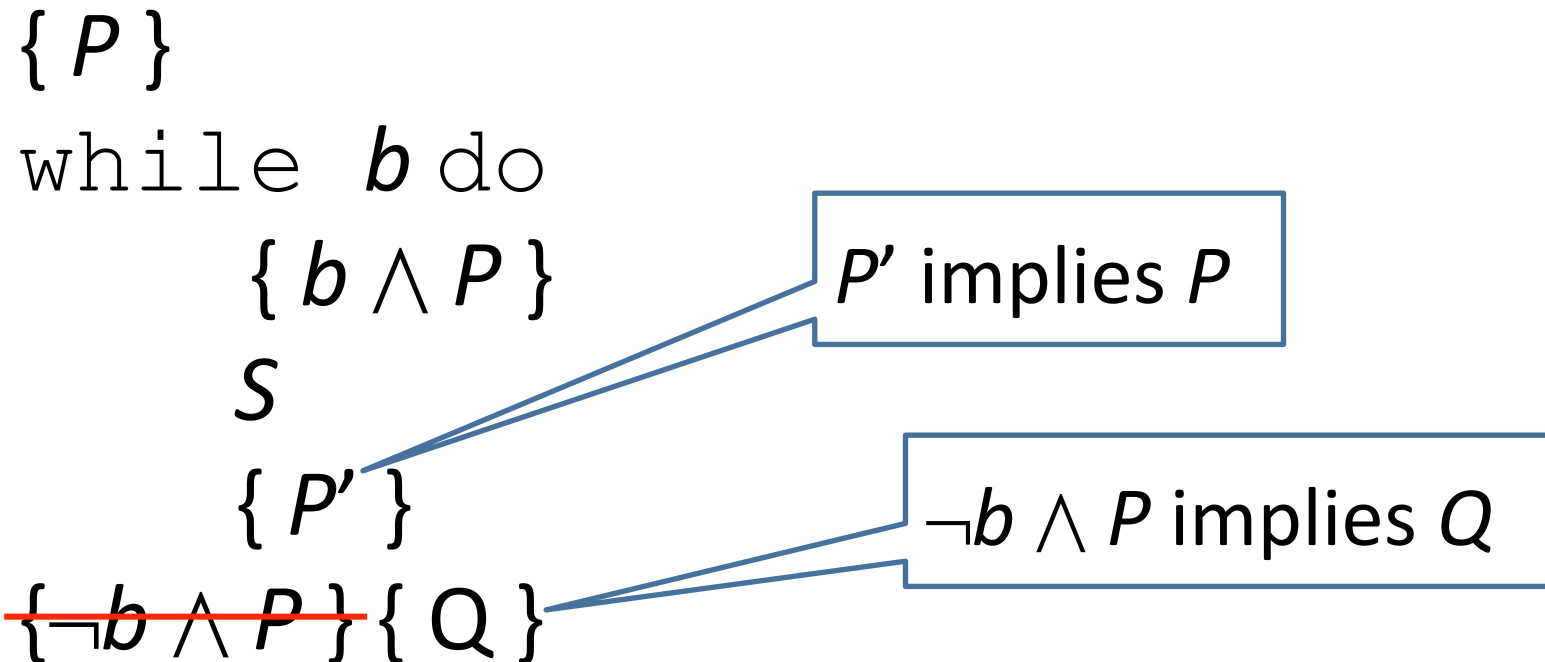
$\quad \{\, b \wedge P \,\}$

$\quad S$

$\{\neg b \wedge P \,\}$

Annotating loops

[while$_p$]
$$\frac{\{\, b \wedge P \,\}\ S\ \{\, P \,\}}{\{\, P \,\}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\neg b \wedge P\,\}}$$



Annotated factorial program

$\{ x=n \}$

```
y := 1;
```

$\{ x>0 \Rightarrow y*x!=n! \wedge n \geq x \}$

```
while ¬(x=1) do
```

$\{ x-1>0 \Rightarrow (y*x)*(x-1)!=n! \wedge n \geq (x-1) \}$

```
  y := y*x;
```

$\{ x-1>0 \Rightarrow y*(x-1)!=n! \wedge n \geq (x-1) \}$

```
  x := x-1
```

$\{ y*x!=n! \wedge n>0 \}$

- Contrast with proof via natural semantics
- Where did the inductive argument over loop iterations go?

Properties of the semantics

Equivalence

– What is the analog of program equivalence in axiomatic verification?

Soundness

– Can we prove incorrect properties?

Completeness

– Is there something we can't prove?

Provable equivalence

- We say that C_1 and C_2 are provably equivalent if for all P and Q
  $\vdash_p \{ P \}\ C_1\ \{ Q \}$ if and only if $\vdash_p \{ P \}\ C_2\ \{ Q \}$
- Examples:
  - S; **`skip`** and S
  - $S_1; (S_2; S_3)$ and $(S_1; S_2); S_3$

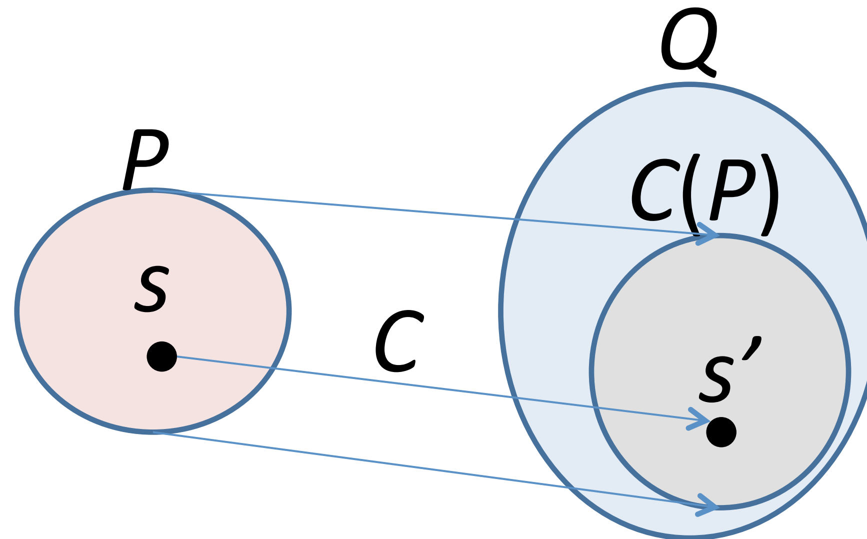
S_1; (S_2; S_3) is provably equivalent to (S_1; S_2); S_3

$$\dfrac{\{P\}\ S_1\ \{P'\} \qquad \dfrac{\{P'\}\ S_2\ \{P''\} \quad \{P''\}\ S_3\ \{Q\}}{\{P'\}\ (S_2;\ S_3)\ \{Q\}}}{\{P\}\ S_1;\ (S_2;\ S_3)\ \{Q\}}$$

$$\dfrac{\dfrac{\{P\}\ S_1\ \{P'\} \quad \{P'\}\ S_2\ \{P''\}}{\{P\}\ (S_1;\ S_2)\ \{P''\}} \qquad \{P''\}\ S_3\ \{Q\}}{\{P\}\ (S_1;\ S_2);\ S_3\ \{Q\}}$$

Valid assertions

- We say that $\{ P \}\ C\ \{ Q \}$ is valid if for all states s, if $s \models P$ and $\langle C, s\rangle \rightarrow s'$ then $s' \models Q$
- Denoted by $\models_p \{ P \}\ C\ \{ Q \}$



Logical implication and equivalence

- We write $A \Rightarrow B$ if for all states s if $s \models A$ then $s \models B$
  - $\{s \mid s \models A\} \subseteq \{s \mid s \models B\}$
  - For every predicate A: $false \Rightarrow A \Rightarrow true$
- We write $A \Leftrightarrow B$ if $A \Rightarrow B$ and $B \Rightarrow A$
  - $false \Leftrightarrow 5=7$
- In writing Hoare-style proofs, we will often replace a predicate A with A' such that $A \Leftrightarrow A'$ and A' is “simpler”

Soundness and completeness

- The inference system is sound:
  - $\vdash_p \{ P \}\ C\ \{ Q \}$ implies $\models_p \{ P \}\ C\ \{ Q \}$

- The inference system is complete:
  - $\models_p \{ P \}\ C\ \{ Q \}$ implies $\vdash_p \{ P \}\ C\ \{ Q \}$

Weakest liberal precondition

- A backward-going predicate transformer
- The weakest liberal precondition for Q is

$$s \models \mathrm{wlp}(C, \mathrm{Q})$$

  if and only if for all states s'
  if $\langle C, s\rangle \longrightarrow s'$ then $s' \models Q$

Propositions:

1. $\models_{\mathrm{p}} \{\ \mathrm{wlp}(C, Q)\ \}\ C\ \{\ Q\ \}$
2. If $\models_{\mathrm{p}} \{\ P\ \}\ C\ \{\ Q\ \}$ then $P \Rightarrow \mathrm{wlp}(C, Q)$

Strongest postcondition

- A forward-going predicate transformer
- The strongest postcondition for P is

  $$s' \models \text{sp}(P, \text{C})$$

  if and only if there exists s such that if $\langle C, s\rangle \longrightarrow s'$ and $s \models P$

1. $\models_{\text{p}} \{ P \}\ C\ \{ \text{sp}(P, C) \}$
2. If $\models_{\text{p}} \{ P \}\ C\ \{ Q \}$ then $\text{sp}(P, C) \Rightarrow Q$

Predicate transformer semantics

- wlp and sp can be seen functions that transform predicates to other predicates
  - wlp⟦C⟧ : **Predicate** → **Predicate**
    { P } C { Q } if and only if wlp⟦C⟧ Q = P
  - sp⟦C⟧ : **Predicate** → **Predicate**
    { P } C { Q } if and only if sp⟦C⟧ P = Q

Hoare logic is (relatively) complete

- Proving
  $\models_p \{ P \}\, C \,\{ Q \}$ implies $\vdash_p \{ P \}\, C \,\{ Q \}$
  is the same as proving
  $\vdash_p \{ \text{wlp}(C, Q) \}\, C \,\{ Q \}$
- Suppose that $\models_p \{ P \}\, C \,\{ Q \}$
  then (from proposition 2) $P \Rightarrow \{ \text{wlp}(C, Q) \}$

$$[\text{cons}_p]\ \frac{\{ P \}\, S \,\{ Q \}}{\{ \text{wlp}(C, Q) \}\, S \,\{ Q \}}$$

Calculating wlp

1. wlp(**`skip`**, Q) = Q
2. wlp(`x` := a, Q) = $Q[a/\texttt{x}]$
3. wlp(S_1; S_2, Q) = wlp(S_1, wlp(S_2, Q))
4. wlp(`if` b `then` S_1 `else` S_2, Q) =
   $(b \wedge \text{wlp}(S_1, Q)) \vee (\neg b \wedge \text{wlp}(S_2, Q))$
5. wlp(`while` b `do` S, Q) = … ?

   hard to capture

Calculating wlp of a loop

Idea: we know the following statements are semantically equivalent

```
while b do S
if b do (S; while b do S) else skip
```

Let's try to substitute and calculate on

wlp(while b do S, Q) =

wlp(if b do (S; while b do S) else skip, Q) =

$(b \land$ wlp(S; while b do S, Q)) $\lor$ $(\neg b \land$ wlp(skip, Q)) =

$(b \land$ wlp(S, **wlp(while b do S, Q)**)) $\lor$ $(\neg b \land Q)$

LoopInv = $(b \land$ wlp(S, **LoopInv**)) $\lor$ $(\neg b \land Q)$

We have a recurrence

The loop invariant

Prove the following triple

```
{ timer ≥ 0 }
while (timer > 0) do
      timer := timer - 1
{ timer = 0 }
```

- LoopInv = (b $\wedge$ wlp(S, LoopInv)) $\vee$ ($\neg$b $\wedge$ Q)
- Let's substitute LoopInv with timer$\geq$0
- Show that timer$\geq$0 is equal to
  (timer>0 $\wedge$ wlp(timer:=timer-1, timer$\geq$0)) $\vee$ (timer$\leq$0 $\wedge$ timer=0)
  = (timer>0 $\wedge$ (timer$\geq$0)[timer-1/timer]) $\vee$ (timer$\leq$0 $\wedge$ timer=0)
  = (timer>0 $\wedge$ timer-1$\geq$0) $\vee$ (timer$\leq$0 $\wedge$ timer=0)
  = timer>0 $\vee$ timer=0
  = timer$\geq$0

Issues with wlp-based proofs

- Requires backwards reasoning – not very intuitive
- Backward reasoning is non-deterministic – causes problems when While is extended with dynamically allocated heaps (aliasing)
- Also, a few more rules will be helpful

Conjunction rule

$$[\text{conj}_p]\quad \frac{\{\,P\,\}\;S\;\{\,Q\,\} \qquad \{\,P'\,\}\;S\;\{\,Q'\,\}}{\{\,P \wedge P'\,\}\;S\;\{Q \wedge Q'\,\}}$$

- Not necessary (for completeness) but practically useful
- Starting point of extending Hoare logic to handle parallelism
- Related to Cartesian abstraction
  - Will point this out when we learn it

Structural Rules

$$[\text{disj}_p]\ \frac{\{\,P\,\}\ C\ \{\,Q\,\} \qquad \{\,P'\,\}\ C\ \{\,Q'\,\}}{\{\,P \lor P'\,\}\ C\ \{Q \lor Q'\,\}}$$

$$[\text{exist}_p]\ \frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,\exists \text{v}.\ P\,\}\ C\ \{\,\exists \text{v}.\ Q\,\}}\ \text{v} \notin \text{FV}(\text{C})$$

$$[\text{univ}_p]\ \frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\forall \text{v}.\ P\,\}\ C\ \{\forall \text{v}.\ Q\,\}}\ \text{v} \notin \text{FV}(\text{C})$$

$$[\text{Inv}_p]\ \{\,F\,\}\ C\ \{\,F\,\} \quad \text{Mod}(C) \cap \text{FV}(F)=\{\}$$

- Mod(C) = set of variables assigned to in sub-statements of C
- FV(F) = free variables of F

Invariance + Conjunction = Constancy

$$[\text{constancy}_p] \; \frac{\{\, P \,\}\; C \;\{\, Q \,\}}{\{\, F \wedge P \,\}\; C \;\{\, F \wedge Q \,\}} \quad \mathrm{Mod}(C) \cap \mathrm{FV}(F) = \{\}$$

- Mod(C) = set of variables assigned to in sub-statements of C
- FV(F) = free variables of F

Floyd's strongest postcondition rule

[$\text{ass}_{\text{Floyd}}$] $\{ P \}\ x := a\ \{ \exists v.\ x=a[v/x] \land P[v/x] \}$

where v is a fresh variable

- Example
  { z=x } x:=x+1 { ?$\exists v.\ x=v+1 \land z=v$ }
- This rule is often considered problematic because it introduces a quantifier – needs to be eliminated further on
- We will now see a variant of this rule

"Small" assignment axiom

Create an explicit Skolem variable in precondition

Then assign the resulting value to x

First evaluate a in the precondition state (as a may access x)

$$[ass_{floyd}]\ \{\ x=v\ \}\ x:=a\ \{\ x=a[v/x]\ \}$$

where $v \notin FV(a)$

- Examples:
  {x=n} x:=5*y {x=5*y}
  {x=n} x:=x+1 {x=n+1}

  {x=y} x:=y+1 {x=y+1}
  {x=n} x:=y+1 {x=y+1}
  $[exist_p]$ {∃n. x=n} x:=y+1 {∃n. x=y+1} therefore {true} x:=y+1 {x=y+1}
  $[constancy_p]$ {z=9} x:=y+1 {z=9 ∧ x=y+1}

Buggy sum program

```
{ y≥0 }
x := 0
{ y≥0 ∧ x=0 }
res := 0
{ y≥0 ∧ x=0 ∧ res=0 }
Inv = { y≥0 ∧ res=Sum(0, x) }
    = { y≥0 ∧ res=m ∧ x=n ∧ m=Sum(0, n) }
while (x≤y) do
        { y≥0 ∧ res=m ∧ x=n ∧ m=Sum(0, n) ∧ x≤y ∧ n≤y }
         x := x+1
        { y≥0 ∧ res=m ∧ x=n+1 ∧ m=Sum(0, n) ∧ n≤y}
        res := res+x
        { y≥0 ∧ res=m+x ∧ x=n+1 ∧ m=Sum(0, n) ∧ n≤y}
        { y≥0 ∧ res-x=Sum(0, x-1) ∧ n≤y}
        { y≥0 ∧ res=Sum(0, x) }

{ y≥0 ∧ res=Sum(0, x) ∧ x>y } ⇏
{res = Sum(0, y) }
```

Sum program

- Define Sum(0, n) = 0+1+…+n

Background axiom

$$\frac{\{\, x=\mathrm{Sum}(0, n) \,\}\ \{\, y=n+1 \,\}}{\{\, x+y=\mathrm{Sum}(0, n+1) \,\}}$$

```
{ y≥0 }
x := 0
{ y≥0 ∧ x=0 }
res := 0
{ y≥0 ∧ x=0 ∧ res=0 }
Inv = { y≥0 ∧ res=Sum(0, x) ∧ x≤y                        }
      { y≥0 ∧ res=m ∧ x=n ∧ n≤y ∧ m=Sum(0, n) }
while (x<y) do
       { y≥0 ∧ res=m ∧ x=n ∧ m=Sum(0, n) ∧ x<y ∧ n<y }
       res := res+x
       { y≥0 ∧ res=m+x ∧ x=n ∧ m=Sum(0, n) ∧ n<y      }
       x := x+1
       { y≥0 ∧ res=m+x ∧ x=n+1 ∧ m=Sum(0, n) ∧ n<y    }
       { y≥0 ∧ res-x=Sum(0, x-1) ∧ x-1<y }
       { y≥0 ∧ res=Sum(0, x) }
{ y≥0 ∧ res=Sum(0, x) ∧ x≤y ∧ x≥y }
{ y≥0 ∧ res=Sum(0, y) ∧ x=y }
{ res = Sum(0, y) }
```

Floyd's strongest postcondition rule

$[\text{ass}_{\text{Floyd}}]$ $\{ P \}\ x := a\ \{ \exists v.\ x=a[v/x] \wedge P[v/x] \}$
where v is a fresh variable

- Example
  $\{ z=x \}\ x:=x+1\ \{ ?\exists v.\ x=v+1 \wedge z=v \}$
- This rule is often considered problematic because it introduces a quantifier – needs to be eliminated further on
- We will now see a variant of this rule

Floyd's strongest postcondition rule

$[\text{ass}_{\text{Floyd}}]$ $\{ P \}\ x := a\ \{ \exists v.\ x=a[v/x] \land P[v/x] \}$
where v is a fresh variable

- Example
  $\{ z=x \}\ x:=x+1\ \{ \exists v.\ x=v+1 \land z=v \}$
- This rule is often considered problematic because it introduces a quantifier – needs to be eliminated further on
- We will now see a variant of this rule

“Small” assignment axiom

Create an explicit Skolem variable in precondition

Then assign the resulting value to *x*

First evaluate *a* in the precondition state (as *a* may access *x*)

$$[ass_{floyd}]\ \{\ x=v\ \}\ x:=a\ \{\ x=a[v/x]\ \}$$

where $v \notin FV(a)$

- Examples:
  {x=n} x:=5*y {x=5*y}
  {x=n} x:=x+1 {x=n+1}

  {x=n} x:=y+1 {x=y+1}
  $[exist_p]$ {∃n. x=n} x:=y+1 {∃n. x=y+1} therefore {true} x:=y+1 {x=y+1}
  $[constancy_p]$ {z=9} x:=y+1 {z=9 ∧ x=y+1}

“Small” assignment axiom

$[\mathrm{ass}_{floyd}]$ { x=v } x:=a { x=a[v/x] }

where $v \notin FV(a)$

- Examples:
  {x=n} x:=5*y {x=5*y}
  {x=n} x:=x+1 {x=n+1}

  {x=n} x:=y+1 {x=y+1}
  $[\mathrm{exist}_p]$ {∃n. x=n} x:=y+1 {∃n. x=y+1} therefore {true} x:=y+1 {x=y+1}
  $[\mathrm{constancy}_p]$ {z=9} x:=y+1 {z=9 ∧ x=y+1}

“Small” assignment axiom

$[ass_{floyd}]$ $\{ x=v \}\ x:=a\ \{ x=a[v/x] \}$

where $v \notin FV(a)$

- Examples:
  {x=n} x:=5*y {x=5*y}
  {x=n} x:=x+1 {x=n+1}

  {x=n} x:=y+1 {x=y+1}
  $[exist_p]$ {∃n. x=n} x:=y+1 {∃n. x=y+1} therefore {true} x:=y+1 {x=y+1}
  $[constancy_p]$ {z=9} x:=y+1 {z=9 ∧ x=y+1}

“Small” assignment axiom

$[ass_{floyd}]$ $\{ x=v \}\ x:=a\ \{ x=a[v/x] \}$
where $v \notin FV(a)$

- Examples:
  {x=n} x:=5*y {x=5*y}
  {x=n} x:=x+1 {x=n+1}

  {x=n} x:=y+1 {x=y+1}
  $[exist_p]$ {∃n. x=n} x:=y+1 {∃n. x=y+1} therefore {true} x:=y+1 {x=y+1}
  $[constancy_p]$ {z=9} x:=y+1 {z=9 ∧ x=y+1}

Buggy sum program

```
{ y≥0 }
x := 0
{ y≥0 ∧ x=0 }
res := 0
{ y≥0 ∧ x=0 ∧ res=0 }
Inv = { y≥0 ∧ res=Sum(0, x) }
    = { y≥0 ∧ res=m ∧ x=n ∧ m=Sum(0, n) }
while (x≤y) do
        { y≥0 ∧ res=m ∧ x=n ∧ m=Sum(0, n) ∧ x≤y ∧ n≤y }
         x := x+1
        { y≥0 ∧ res=m ∧ x=n+1 ∧ m=Sum(0, n) ∧ n≤y}
        res := res+x
        { y≥0 ∧ res=m+x ∧ x=n+1 ∧ m=Sum(0, n) ∧ n≤y}
        { y≥0 ∧ res-x=Sum(0, x-1) ∧ n≤y}
        { y≥0 ∧ res=Sum(0, x) }

{ y≥0 ∧ res=Sum(0, x) ∧ x>y } ⇏
{res = Sum(0, y) }
```

Sum program

- Define Sum(0, n) = 0+1+…+n

Background axiom

$$\text{[axm-Sum]}\ \frac{\{\, x=\mathrm{Sum}(0, n) \,\}\ \{\, y=n+1 \,\}}{\{\, x+y=\mathrm{Sum}(0, n+1) \,\}}$$

```
{ y≥0 }
x := 1
{ y≥0 ∧ x=1 }
res := 0
{ y≥0 ∧ x=1 ∧ res=0 }
Inv = { y≥0 ∧ res=Sum(0, x-1) ∧ x≤y+1                    }
      { y≥0 ∧ res=m ∧ x=n ∧ n≤y+1 ∧ m=Sum(0, n-1)        }
while (x≤y) do
      { y≥0 ∧ res=m ∧ x=n ∧ m=Sum(0, n-1) ∧ x<y ∧ n≤y+1 }
      res := res+x
      { y≥0 ∧ res=m+x ∧ x=n ∧ m=Sum(0, n-1) ∧ n≤y+1      }
      x := x+1
      { y≥0 ∧ res=m+x ∧ x=n+1 ∧ m=Sum(0, n-1) ∧ n≤y+1    }
      { y≥0 ∧ res-x=Sum(0, x-1) ∧ x-1<y+1 }
      { y≥0 ∧ res=Sum(0, x-1) ∧ x≤y+1 } // axm-Sum
{ y≥0 ∧ res=Sum(0, x-1) ∧ x≤y+1 ∧ x>y }
{ y≥0 ∧ res=Sum(0, x-1) ∧ x=y+1 }
{ y≥0 ∧ res=Sum(0, y) }
{ res = Sum(0, y) }
```

Sum program

- Define Sum(0, n) = 0+1+…+n

Background axiom

[axm-Sum] $\dfrac{\{\, x=\mathrm{Sum}(0, n) \,\}\ \{\, y=n+1 \,\}}{\{\, x+y=\mathrm{Sum}(0, n+1) \,\}}$

```
{ y≥0 }
x := 1
{ y≥0 ∧ x=0 }
res := 0
{ y≥0 ∧ x=0 ∧ res=0 }
Inv = { y≥0 ∧ res=Sum(0, x-1) ∧ x≤y+1                  }
while (x≤y) do
        { y≥0 ∧ res=m ∧ x=n ∧ m=Sum(0, n-1) ∧ n≤y+1 ∧ x<y }
        res := res+x
        { y≥0 ∧ res=m+x ∧ x=n ∧ m=Sum(0, n-1) ∧ n≤y+1 }
        { y≥0 ∧ res=Sum(0, n) ∧ x=n ∧ n≤y+1 } // axm-Sum
        x := x+1
        { y≥0 ∧ res=Sum(0, n) ∧ x=n+1 ∧ n≤y+1    }
        { y≥0 ∧ res=Sum(0, x-1) ∧ x≤y+1 }
{ y≥0 ∧ res=Sum(0, x-1) ∧ x≤y+1 ∧ x>y }
{ y≥0 ∧ res=Sum(0, x-1) ∧ x=y+1 }
{ y≥0 ∧ res=Sum(0, y) }
{ res = Sum(0, y) }
```

Example 1: Absolute value program

```
{ }
if x<0 then
    x := -x
else
    skip
{ }
```

Absolute value program

```
{ x=v }
if x<0 then
    { x=v ∧ x<0 }
      x := -x
      { x=-v ∧ x>0 }
else
      { x=v ∧ x≥0 }
      skip
      { x=v ∧ x≥0 }
{ v<0 ∧ x=-v ∨ v≥0 ∧ x=v}
{ x=|v| }
```

Example 2: Variable swap program

```
{ }
t := x
x := y
y := t
{ }
```

Variable swap program

```
{ x=a ∧ y=b }
t := x
{ x=a ∧ y=b ∧ t=a }
x := y
{ x=b ∧ y=b ∧ t=a }
y := t
{ x=b ∧ y=a ∧ t=a }
{ x=b ∧ y=a } // cons
```

Example 3: Axiomatizing data types

$$S ::= x := a \mid x := y[a] \mid y[a] := x$$
$$\mid \texttt{skip} \mid S_1; S_2$$
$$\mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$
$$\mid \texttt{while}\ b\ \texttt{do}\ S$$

- We added a new type of variables – array variables
  - Model array variable as a function $y : \mathbf{Z} \rightarrow \mathbf{Z}$
- We need the two following axioms:

$$\{\ y[x \mapsto a](x) = a\ \}$$

$$\{\ z \neq x \Rightarrow y[x \mapsto a](z) = y(z)\ \}$$

Array update rules (wp)

$$S ::= x := a \mid x := y[a] \mid y[a] := x \mid \textbf{skip} \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } b \textbf{ do } S$$

A very general approach – allows handling many data types

- Treat an array assignment $y[a] := x$ as an update to the array function y
  - $y := y[a \mapsto x]$ meaning $y' = \lambda v.\ v=a\ ?\ X : y(v)$

[array-update] $\{ P[y[a \mapsto x]/y] \}\ y[a] := x\ \{ P \}$

[array-load] $\{ P[y(a)/x] \}\ x := y[a]\ \{ P \}$

Array update rules (wp) example

- Treat an array assignment $y[a]$ **:=** x as an update to the array function y
  - $y := y[a \mapsto x]$ meaning $y' = \lambda v.\ v=a\ ?\ x : y(v)$

[array-update] $\{ P[y[a \mapsto x]/y] \}\ y[a]$ **:=** $x\ \{ P \}$

$\{x=y[i \mapsto 7](i)\}\ y[i]:=7\ \{x=y(i)\}$

$\{x=7\}\ y[i]:=7\ \{x=y(i)\}$

[array-load] $\{ P[y(a)/x] \}\ x$ **:=** $y[a]\ \{ P \}$

$\{y(a)=7\}\ x:=y[a]\ \{x=7\}$

Array update rules (sp)

In both rules
v, g, and b are fresh

[array-update$_F$] $\{ x=v \wedge y=g \wedge a=b \}\ y[a] := x\ \{ y=g[b \mapsto v] \}$

[array-load$_F$] $\{ y=g \wedge a=b \}\ x := y[a]\ \{ x=g(b) \}$

Array-max program

```
nums : array
N : int // N stands for num's length
{ N≥0 ∧ nums=orig_nums }
x := 0
res := nums[0]
while x < N
    if nums[x] > res then
        res := nums[x]
    x := x + 1
```

```
1. { x=N }
2. { ∀m. (m≥0 ∧ m<N) ⇒ nums(m)≤res }
3. { ∃m. m≥0 ∧ m<N ∧ nums(m)=res }
4. { nums=orig_nums }
```

Array-max program

```
nums : array
N : int // N stands for num's length
{ N≥0 ∧ nums=orig_nums }
x := 0
res := nums[0]
while x < N
    if nums[x] > res then
        res := nums[x]
    x := x + 1
Post1: { x=N }
Post2: { nums=orig_nums }
Post3: { ∀m. 0≤m<N ⇒ nums(m)≤res }
Post4: { ∃m. 0≤m<N ∧ nums(m)=res }
```

Summary

- `C` programming language
- P assertions
- $\{P\}$ `C` $\{Q\}$ judgments
- $\{ P[a/\texttt{x}] \}$ `x` $:= a \{ P \}$ proof Rules
  - Soundness
  - Completeness
- $\{x = N\}$ `y:=factorial(x)` $\{ y = N! \}$ proofs

Extensions to axiomatic semantics

- Procedures
- Total correctness assertions
- Assertions for execution time
  - Exact time
  - Order of magnitude time
- Assertions for dynamic memory
  - Separation Logic
- Assertions for parallelism
  - Owicki-Gries
  - Concurrent Separation Logic
  - Rely-guarantee