Program Analysis and Verification

0368-4479

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

Noam Rinetzky

Lecture 6: Axiomatic Semantics II

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

Good manners

- Mobiles

Home Work Assignment #1

(due next lesson)

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) .

Axiomatic Semantics

Robert Floyd

C.A.R. Hoare

Edsger W. Dijkstra

A simple imperative language: **While**

Abstract syntax:

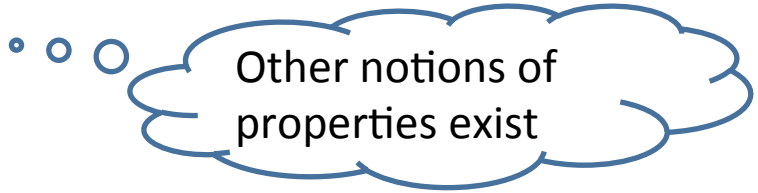
$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$$

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

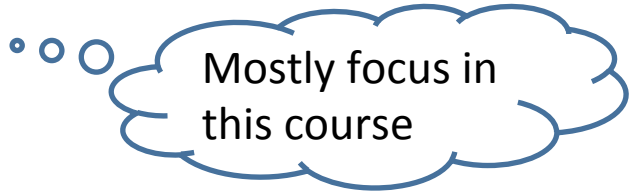
$$S ::= x := a \mid \texttt{skip} \mid S_1 ; S_2$$
$$\mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$$
$$\mid \texttt{while } b \texttt{ do } S$$

Program correctness concepts

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

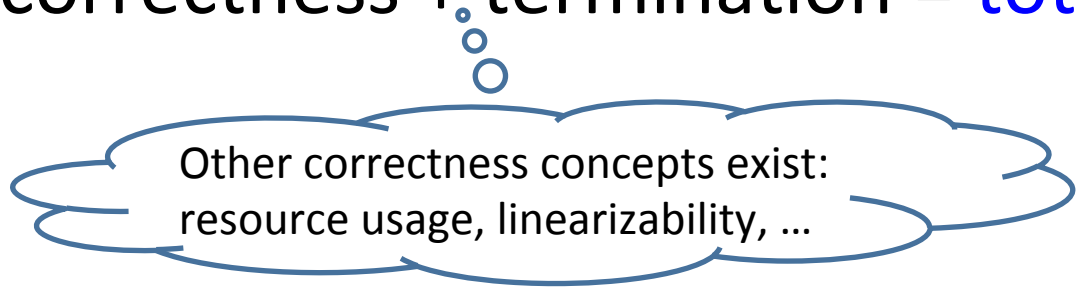


- Partial correctness = properties that hold ***if*** program terminates



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

partial correctness + termination = total correctness



Factorial example

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

- $\langle S_{fac}, s \rangle \longrightarrow s'$ implies s' **`y`** = (s **`x`**)!

Natural semantics for **While**

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

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

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

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

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

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

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

Semantics-based Proof

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

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

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

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

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

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

$[\mathrm{comp}_{\mathrm{ns}}]$ $\quad \dfrac{\langle S_1, s\rangle \rightarrow s',\ \langle S_2, s'\rangle \rightarrow s''}{\langle S_1; S_2, s\rangle \rightarrow s''}$

Semantics-based Proof

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

- $\langle S_{fac}, s \rangle \rightarrow s'$ implies s' **`y`** = (s **`x`**)!

- Tedious
- Correct?
  - How do we check correctness?
  - Rigorous vs. formal

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 (systematically) use the proof system to show correctness
- The meaning of a program language is a set of verification rules

Axiomatic Verification: Spec

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

- $\langle S_{fac}, s \rangle \longrightarrow s'$ implies s' **y** = (s **x**)!
- {x = N} S_{fac} {y = N!}
  - {Pre-condition (s)} Command (S_{fac}) {post-state(s')}
    - Not {true} S_{fac} {y = x!}

Partial vs. Total Correctness

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

- $\langle S_{fac}, s \rangle \longrightarrow s'$ implies s' **y** = (s **x**)!

- {x = N} S_{fac} {y = N!}
  - {Pre-condition (s)} Command (S_{fac}) {post-state(s')}
    - Not {true} S_{fac} {y = x!}
- [x = N] S_{fac} [y = N!]

Hoare Triples

Verification: Assertion-Based [Floyd, '67]

- **Assertion**: invariant at specific program point
  - E.g., assert(*e*)

- use assertions as foundation for **static correctness proofs**
- specify assertions at *every* program point
- correctness reduced to **reasoning about individual statements**

Annotated Flow Programs

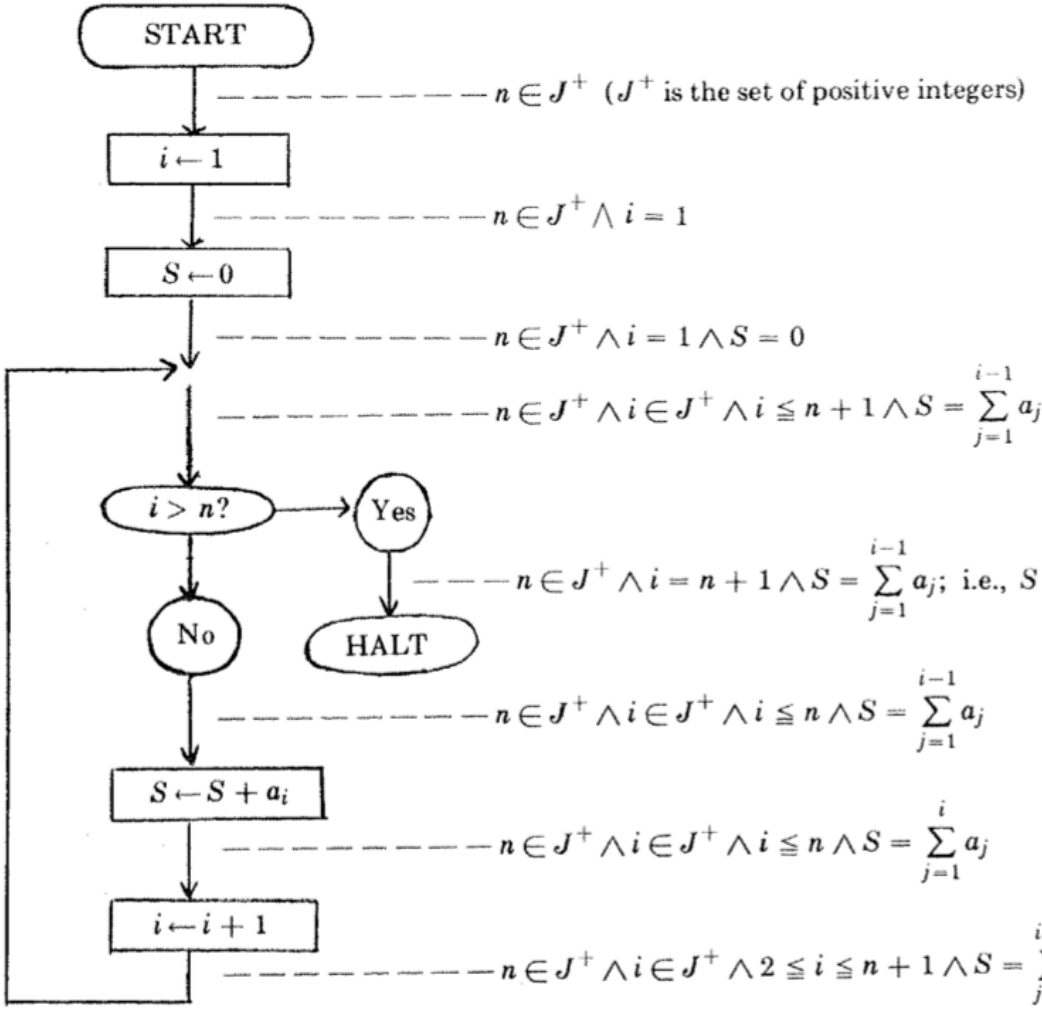


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geqq 0)$

Reduction: Program verification is reduced to claims about the subject of discourse

Straight line code: claims are determined "by construction"

Annotated Flow Programs

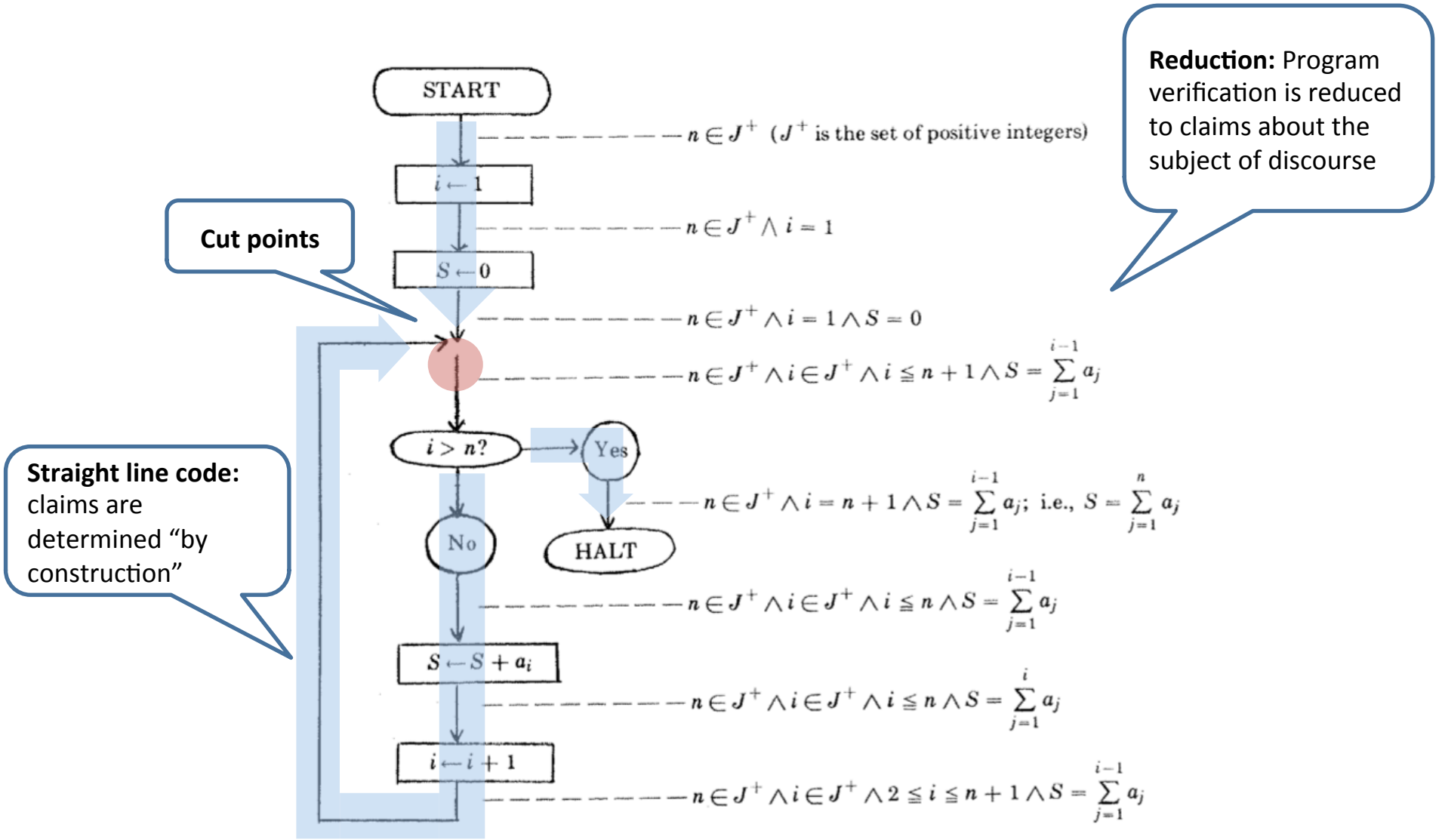


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geqq 0)$

Assertion-Based Verification [Floyd, '67]

- **Assertion**: invariant at specific program point
  - E.g., assert(*e*)

- Proof reduced to logical claims
  - Considering the effect of statements
  - But, not reusable

- Challenge: Finding invariants at cut points in loops

Floyd-Hoare Logic 1969

- Use Floyd's ideas to define **axiomatic semantics**
  - Structured programming
    - No gotos
    - Modular (reusable claims)
  - Hoare triples
    - {P} C {Q}
    - [P] C [Q] (often <P> C <Q>)
  - Define the programming language semantics as a **proof system**

Axiomatic semantics for **While**

Axiom for every primitive statement

Inference rule for every composed statement

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

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

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

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

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

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

Proofs: Inference trees

- Leaves are axiom
- Internal nodes correspond to composed statements
- Inference tree is called
  - Simple if tree is only an axiom
  - Composite otherwise

- Similar to derivation trees of natural semantics
  - Reasoning about immediate constituents

Factorial proof

Goal: $\{ x=n \}\ \mathbf{y:=1;\ while\ (x\neq 1)\ do\ (y:=y*x;\ x:=x-1)}\ \{ \mathbf{y}=n! \wedge n>0 \}$

W = **while (x≠1) do (y:=y*x; x:=x−1)**

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

$$\dfrac{\dfrac{\dfrac{\{ x=n \}\ \mathbf{y:=1}\ \{ \text{INV} \}}{\{ x=n \}\ \mathbf{y:=1}\ \{ \text{INV} \}}\ [\text{cons}] \qquad \dfrac{\dfrac{\dfrac{\{ \text{INV}[x-1/x][y*x/y] \}\ \mathbf{y:=y*x}\ \{ \text{INV}[x-1/x] \} \quad \{ \text{INV}[x-1/x] \}\ \mathbf{x:=x-1}\ \{\text{INV}\}}{\{ \text{INV}[x-1/x][y*x/y] \}\ \mathbf{y:=y*x;\ x:=x-1}\ \{\text{INV}\}}\ [\text{comp}]}{\{ \text{INV}[1/y] \}\ \mathbf{y:=1}\ \{ \text{INV} \}}\ [\text{cons}]}{\{ \mathbf{x\neq 1} \wedge \text{INV} \}\ \mathbf{y:=y*x;\ x:=x-1}\ \{ \text{INV} \}}\ [\text{while}]}{\{ \text{INV} \}\ W\ \{ x=1 \wedge \text{INV} \}}\ [\text{cons}]}{\{ \text{INV} \}\ W\ \{ \mathbf{y}=n! \wedge n>0 \}}\ [\text{comp}]$$

$$\{ x=n \}\ \mathbf{while\ (x\neq 1)\ do\ (y:=y*x;\ x:=x-1)}\ \{ \mathbf{y}=n! \wedge n>0 \}$$

Factorial proof

Goal: $\{ x=n \}\ \mathbf{y:=1;\ while\ (x\neq 1)\ do\ (y:=y*x;\ x:=x-1)}\ \{ \mathbf{y}=n! \wedge n>0 \}$

W = **while (x≠1) do (y:=y*x; x:=x−1)**

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

$$\dfrac{\dfrac{\dfrac{\{ b \wedge P \}\, S\, \{ P \}}{\{ P \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{ \neg b \wedge P \}}\ [\text{while}]}{\{ \text{INV} \}\ W\ \{ x=1 \wedge \text{INV} \}}\ [\text{cons}] \qquad \dfrac{\{ \text{INV}[1/y] \}\ \mathbf{y:=1}\ \{ \text{INV} \}}{\{ x=n \}\ \mathbf{y:=1}\ \{ \text{INV} \}}\ [\text{cons}]}{\{ x=n \}\ \mathbf{while\ (x\neq 1)\ do\ (y:=y*x;\ x:=x-1)}\ \{ \mathbf{y}=n! \wedge n>0 \}}\ [\text{comp}]$$

$$\dfrac{\{ \mathbf{x\neq 1} \wedge \text{INV} \}\ \mathbf{y:=y*x;\ x:=x-1}\ \{ \text{INV} \}}{\{ \text{INV} \}\ W\ \{ x=1 \wedge \text{INV} \}}\ [\text{while}]$$

$$\dfrac{\{ \text{INV}[x-1/x][y*x/y] \}\ \mathbf{y:=y*x}\ \{ \text{INV}[x-1/x] \} \qquad \{ \text{INV}[x-1/x] \}\ \mathbf{x:=x-1}\ \{ \text{INV} \}}{\{ \text{INV}[x-1/x][y*x/y] \}\ \mathbf{y:=y*x;\ x:=x-1}\ \{ \text{INV} \}}\ [\text{comp}]$$

Factorial proof

Goal: { x=n } **y:=1; while (x≠1) do (y:=y*x; x:=x−1)** { **y**=*n*! ∧ *n*>0 }

W = **while (x≠1) do (y:=y*x; x:=x−1)**

INV = x > 0 ⇒ (y · x! = n! ∧ n ≥ x)

$$\frac{\{P'\}\,S\,\{Q'\}}{\{P\}\,S\,\{Q\}}\ \text{if } P \Rightarrow P' \text{ and } Q' \Rightarrow Q$$

$$\dfrac{\dfrac{\dfrac{\{\text{INV}[1/y]\}\ \mathbf{y:=1}\ \{\text{INV}\}}{\{x=n\}\ \mathbf{y:=1}\ \{\text{INV}\}}\ [\text{cons}] \qquad \dfrac{\dfrac{\{\mathbf{x\neq 1} \wedge \text{INV}\}\ \mathbf{y:=y*x;\ x:=x-1}\ \{\text{INV}\}}{\{\text{INV}\}\ W\ \{x=1 \wedge \text{INV}\}}\ [\text{while}]}{\{\text{INV}\}\ W\ \{\mathbf{y}=n! \wedge n>0\}}\ [\text{cons}]}{\{x=n\}\ \mathbf{while\ (x\neq 1)\ do\ (y:=y*x;\ x:=x-1)}\ \{\mathbf{y}=n! \wedge n>0\}}\ [\text{comp}]$$

$$\dfrac{\dfrac{\{\text{INV}[x-1/x][y*x/y]\}\ \mathbf{y:=y*x}\ \{\text{INV}[x-1/x]\} \qquad \{\text{INV}[x-1/x]\}\ \mathbf{x:=x-1}\ \{\text{INV}\}}{\{\text{INV}[x-1/x][y*x/y]\}\ \mathbf{y:=y*x;\ x:=x-1}\ \{\text{INV}\}}\ [\text{comp}]}{\{\text{INV}[x-1/x][y*x/y]\}\ \mathbf{y:=y*x;\ x:=x-1}\ \{\text{INV}\}}\ [\text{cons}]$$

Axiomatic semantics for **While**

Axiom for every primitive statement

Inference rule for every composed statement

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

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

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

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

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

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

Axiomatic semantics for **While**

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

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

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

Notice similarity to natural semantics rules

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

$[\text{while}_p]$ $\dfrac{\{ b \wedge P \}\, S\, \{ P \}}{\{ P \}\ \mathtt{while}\ b\ \mathtt{do}\ S\ \{ \neg b \wedge P \}}$

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

Assignment rule

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

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

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

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

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

`skip` rule

$$[\text{skip}_p] \quad \{ P \}\, \texttt{skip}\, \{ P \}$$

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

Composition rule

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

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

- 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

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

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

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

Loop rule

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

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

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

- 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

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

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

Rule of consequence

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

- Why do we need it?
- Allows the following

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

Inference trees

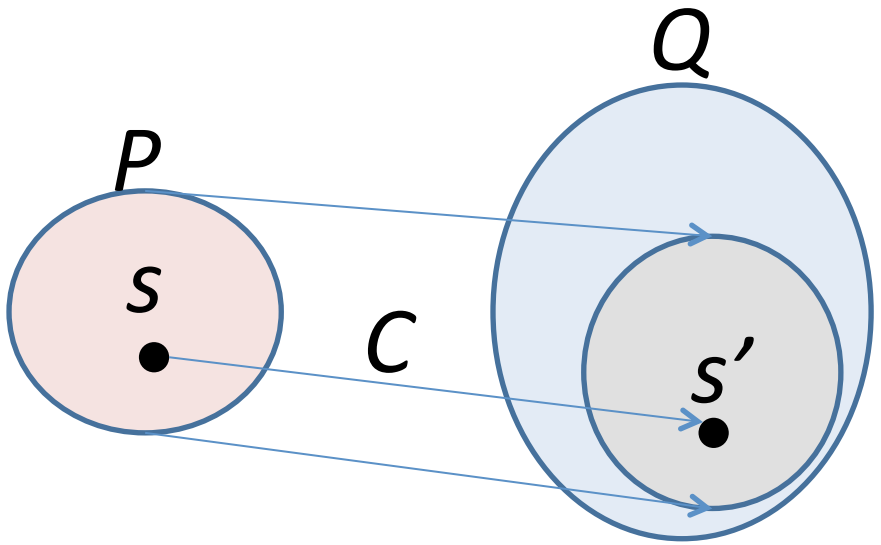
- Similar to derivation trees of natural semantics
- Leaves are …
- Internal nodes correspond to …
- Inference tree is called
  - Simple if tree is only an axiom
  - Composite otherwise

Provability

- We say that an assertion $\{ P \}\ C\ \{ Q \}$ is provable if there exists an inference tree
  - Written as $\vdash_p \{ P \}\ C\ \{ Q \}$

Validity

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



- $\{ P \}\ C\ \{ Q \}$ is valid if

$$\forall s, s' \in \Sigma\ .\ (s \models P \wedge \langle C, s \rangle \rightarrow s') \Rightarrow s' \models Q$$

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 $\vdash_p \{ P \}\ C\ \{ Q \}$

Hoare logic is sound and (relatively) complete

- **Soundness**:

  $\vdash_p \{ P \} C \{ Q \}$ implies $\models_p \{ P \} C \{ Q \}$

- **(Relative) completeness**:

  $\models_p \{ P \} C \{ Q \}$ implies $\vdash_p \{ P \} C \{ Q \}$

  – Provided we can prove any implication $R \Longrightarrow R'$

Hoare logic is sound and (relatively) complete

- **Soundness**:

  $\vdash_p \{ P \} C \{ Q \}$ implies $\models_p \{ P \} C \{ Q \}$

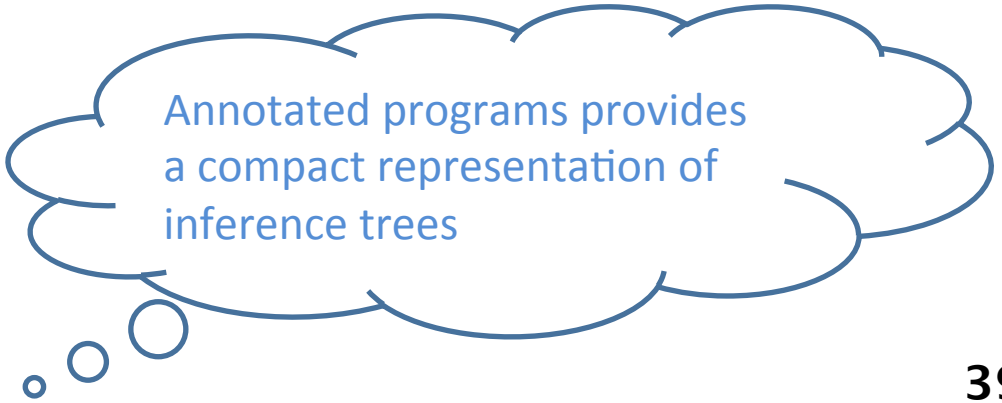
- **(Relative) completeness**:

  $\models_p \{ P \} C \{ Q \}$ implies $\vdash_p \{ P \} C \{ Q \}$

  – Provided we can prove any implication $R \Longrightarrow R'$
    - FYI, nobody tells us how to find a proof …

Is there an Algorithm?

{ **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 }



?

Predicate Transformers

Weakest liberal precondition

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

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

if and only if

$$\boldsymbol{\forall s'.\ \mathbf{if}\ \langle C, s\rangle \rightarrow s'\ \mathbf{then}\ s' \models Q}$$

Propositions:

1. $\models_p \{\ \mathrm{wlp}(C, Q)\ \}\ C\ \{\ Q\ \}$
2. If $\models_p \{\ P\ \}\ C\ \{\ Q\ \}$ then $P \Longrightarrow \mathrm{wlp}(C, Q)$

Strongest postcondition

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

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

if and only if

$$\exists s'.\ \langle C, s\rangle \rightarrow s' \text{ and } s \models P$$

Propositions:

a. $\models_p \{ P \}\ C\ \{ \mathrm{sp}(P, C) \}$

b. If $\models_p \{ P \}\ C\ \{ Q \}$ then $\mathrm{sp}(P, C) \Longrightarrow Q$

Predicate transformer semantics

- wlp and sp can be seen functions that transform predicates to other predicates
  - wlp$[\![C]\!]$: **Predicate** $\rightarrow$ **Predicate**
    $\{ P \} C \{ Q \}$ if and only if wlp$[\![C]\!]$ $Q = P$
  - sp$[\![C]\!]$: **Predicate** $\rightarrow$ **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) $P \Rightarrow \{ \text{wlp}(C, Q) \}$

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

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*) =

$(b \wedge \text{wlp}(S;\ \texttt{while}\ b\ \texttt{do}\ S, Q)) \vee (\neg b \wedge \text{wlp}(\texttt{skip}, Q)) =$

$(b \wedge \text{wlp}(S,\ \textbf{wlp(while } b \textbf{ do } S, Q))) \vee (\neg b \wedge Q)$

LoopInv $= (b \wedge \text{wlp}(S,\ \textbf{LoopInv})) \vee (\neg b \wedge Q)$

We have a recurrence

The loop invariant

Prove the following triple

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

- LoopInv = (b ∧ wlp(S, LoopInv)) ∨ (¬b ∧ Q)
- Let's substitute LoopInv with timer≥0
- Show that timer≥0 is equal to
  (timer>0 ∧ wlp(timer:=timer-1, timer≥0)) ∨ (timer≤0 ∧ timer=0)
  = (timer>0 ∧ (timer≥0)[timer-1/timer]) ∨ (timer≤0 ∧ timer=0)
  = (timer>0 ∧ timer-1≥0) ∨ (timer≤0 ∧ timer=0)
  = timer>0 ∨ timer=0
  = timer≥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

Verification Conditions

- Generate assertions that describe the partial correctness of the program
- Use automatic theorem provers to show partial correctness
- Existing tools ESC/Java, Spec#

Verification condition for annotated commands

S ::= skip | X := a | S; (X:=a) |
S_0 ; {D} S_1 | if b then S_0 else S_1 |
while b {D} do S

vc({P} skip {Q}) = {P⇒Q}

vc({P} X:= a {Q}) = {P ⇒Q[a/X]}

vc({P} S ; X:=a {Q}) = vc({P} S {Q[a/X]})

vc({P} S_0; {D} S_1 {Q}) = vc({P} S_0 {D}) ∪ vc({D} S_1 {Q})

vc({P} if b then S_0 else S_1 {Q}) = vc({P∧b} S_0 {Q}) ∪
vc({P ∧ ¬b} S_1 {Q})

vc({P} while b {D} do c {Q}) = vc({D∧b} c {D}) ∪ {P⇒D} ∪
{D∧ ¬b ⇒Q}

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

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

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

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

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

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

Invariance + Conjunction = Constancy

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

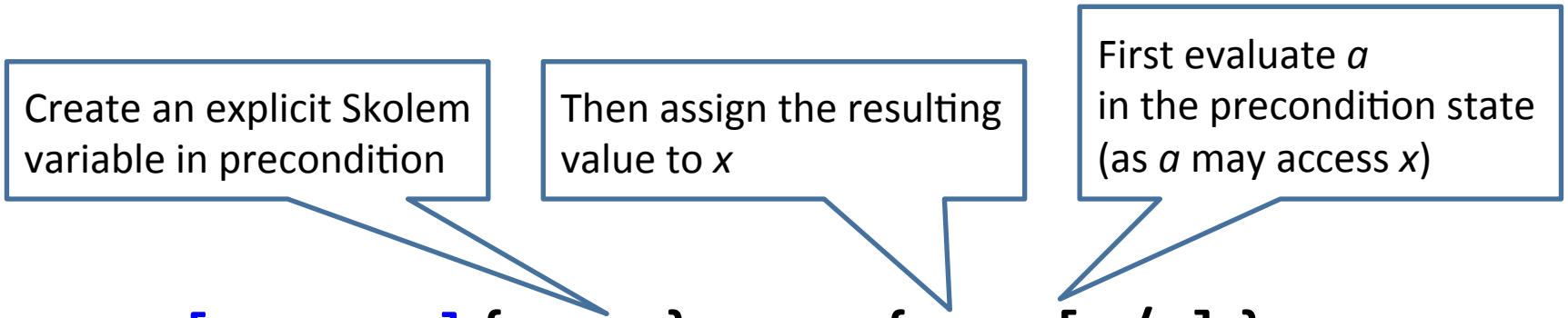
- Mod(*C*) = set of variables assigned to in sub-statements of C
- FV(*F*) = free variables of *F*

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

“Small” assignment axiom



- 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

Background axiom

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

- Define Sum(0, n) = 0+1+…+n

```
{ 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

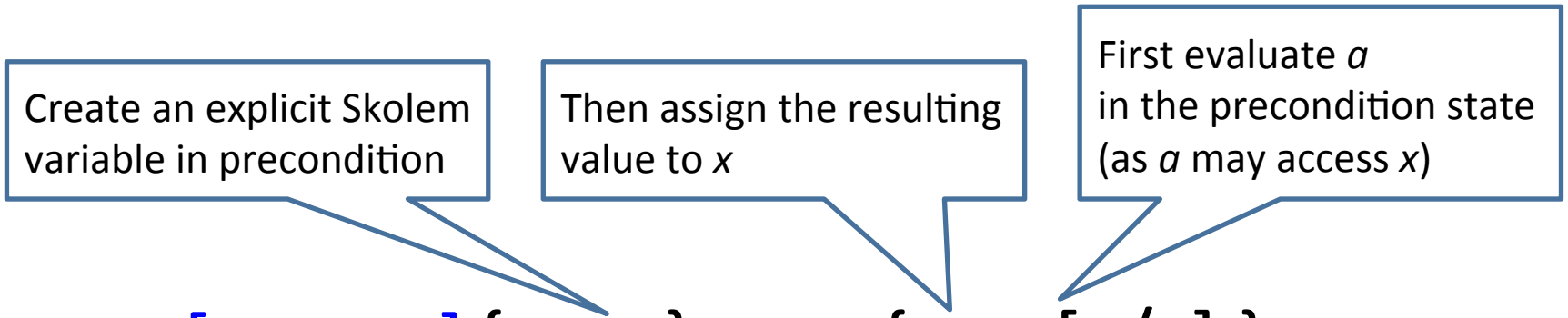
- Example
  $\{ z{=}x \}\ x{:=}x{+}1\ \{ ?\exists v.\ x{=}v{+}1 \wedge z{=}v \}$
- This rule is often considered problematic because it introduces a quantifier – needs to be eliminated further on
- We will now see a variant of this rule

Floyd's strongest postcondition rule

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

“Small” assignment axiom



- 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

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

- 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

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

- 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

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

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

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=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) }
```

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

[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

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

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

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

$$z \neq x \Longrightarrow y[x \mapsto a](z) = y(z)$$

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'=λv. v=a ? x : y(v)

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

{x=y[i↦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

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

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

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
```

$Post_1$: { x=N }

$Post_2$: { nums=orig_nums }

$Post_3$: { $\forall m.\ 0 \le m < N \Rightarrow nums(m) \le res$ }

$Post_4$: { $\exists m.\ 0 \le m < N \wedge nums(m) = res$ }

Summary

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

Extensions to axiomatic semantics

- Assertions for parallelism
  - Owicki-Gries
  - Concurrent Separation Logic
  - Rely-guarantee
- Assertions for dynamic memory
  - Separation Logic

- Procedures
- Total correctness assertions
- Assertions for execution time
  - Exact time
  - Order of magnitude time