Deductive Verification

Mooly Sagiv

Original Slides: Rustan Leino (MSR)

Swart Chaudhuri (Rice U)

Formal Semantics of Programming Languages : Glynn Winskel Chapter 6

Goal: Correctness Proof

- Prove that an algorithm written in an imperative language is correct

Proving Correctness

- We will use an automated proof assistant to do proofs of programs
  - You write the proof, the assistant checks it for you
  - The ultimate TA; doesn't allow you to cheat
  - The technical part of the proof is automated
  - The user is responsible for the loop invariants

Functions on Lists

```
let rec length l =
    match l with
   [] -> 0
  | hd :: tl  -> 1 + length tl
```

length [1; 2; 3] + length [“red”; “yellow”; “green”]

Induction for algorithmic correctness

- Induction for functional programs:
  - The program computes the right outputs on the empty list
  - Assuming the program computes the right outputs on a list L, it computes the right outputs on m :: L
- Induction for imperative programs:
  - All program executions of length 1 are correct
  - If all executions of length k lead to correct outputs, then so do all executions of length (k +1)

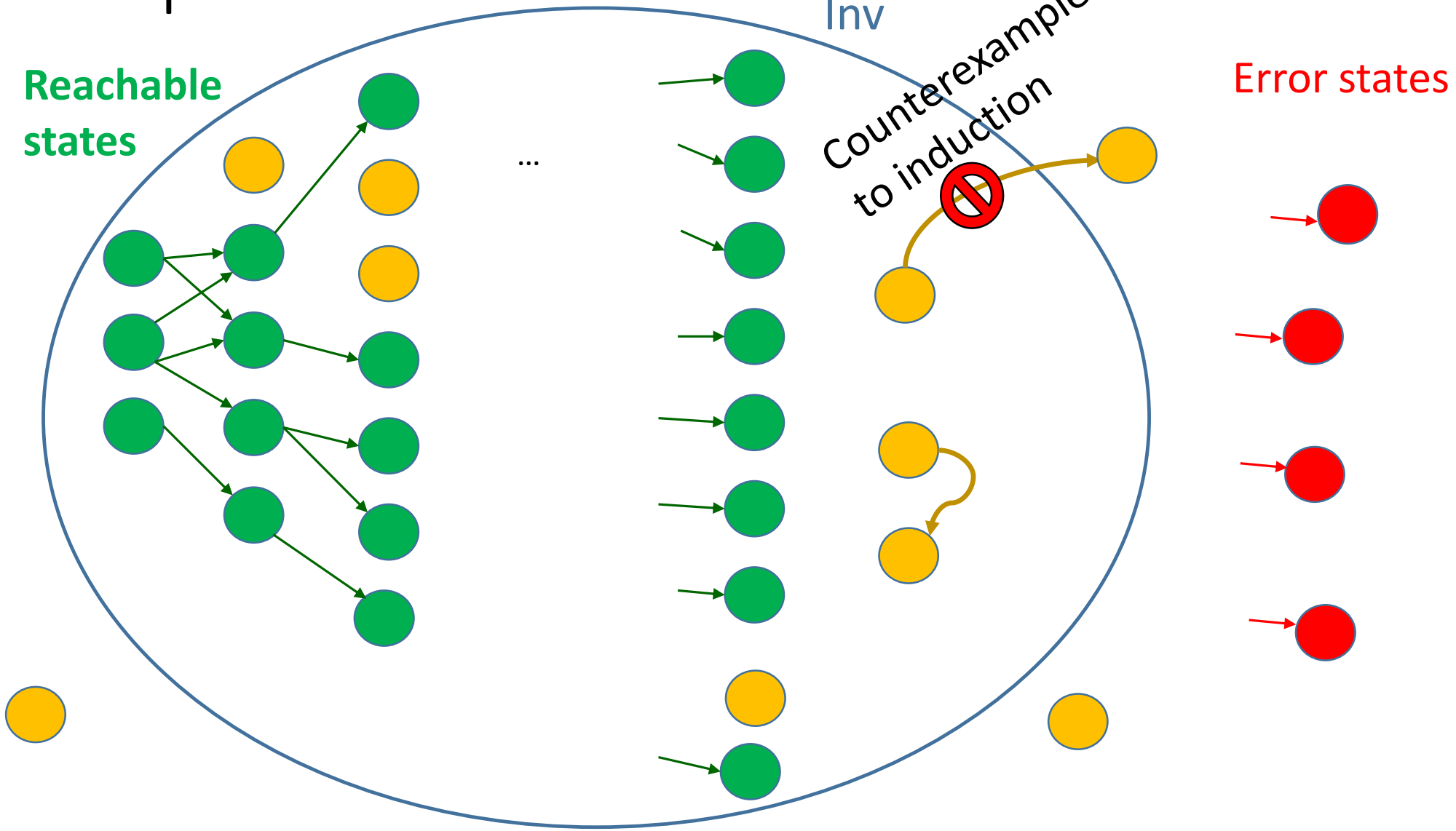
How do we know this program is correct?

```
method Find(a: array<int>, x: int) returns (j : int)
 requires a != null;
{
 var m := 0;  var n := a.Length;
 while (m < n)
 {
 j := (m + n) / 2;
 if (a[j] < x) {
     m := j + 1;
     } else if (x < a[j]) {
          n := j;
          } else {
          return;
          }
  }
 j := -1;
}
```

Correctness of programs

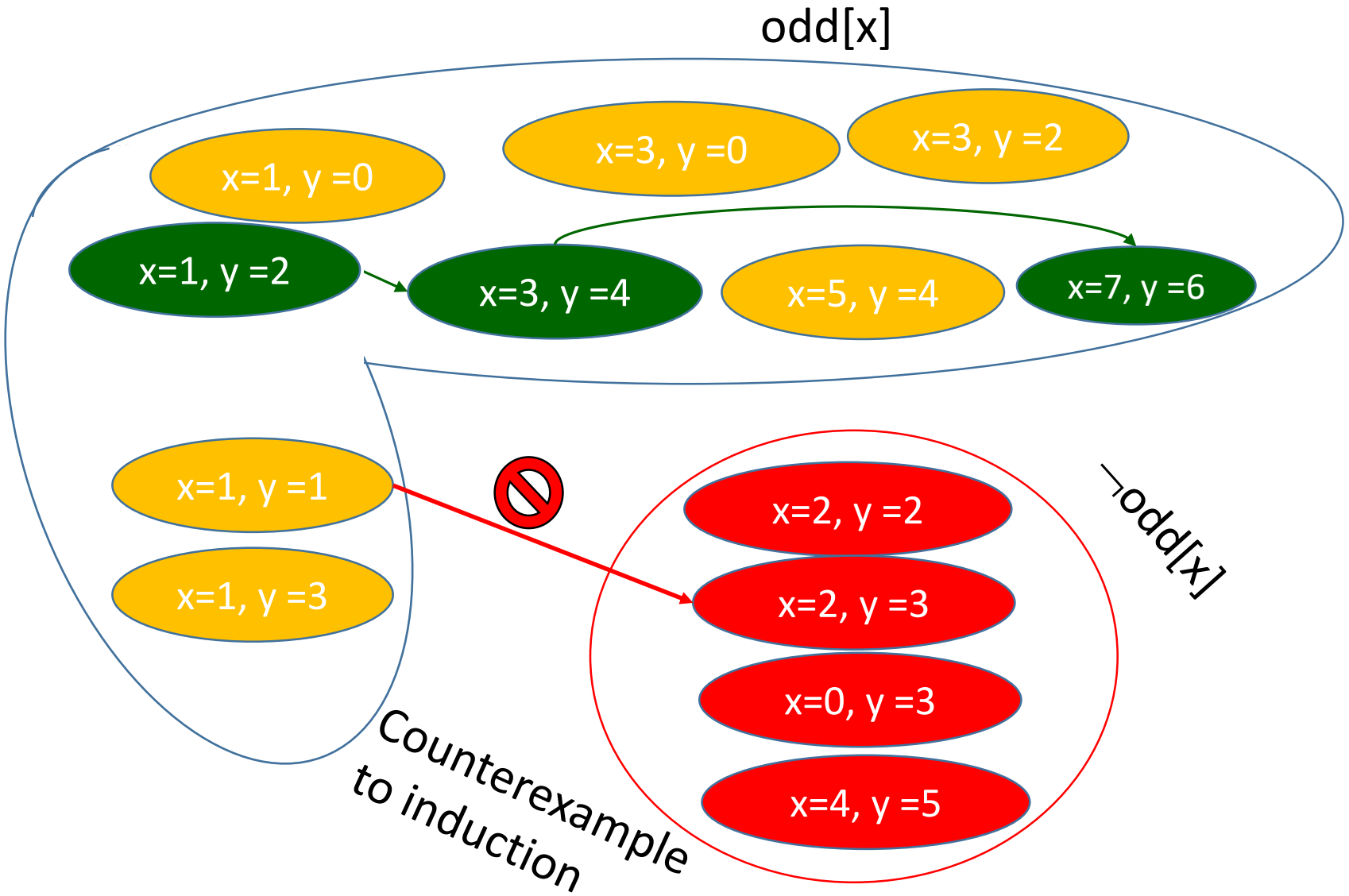
- Operational semantics defines how a program actually behaves
- Speciations state how a program should behave
- Goal: Guarantee that the program follows the

Loop invariants



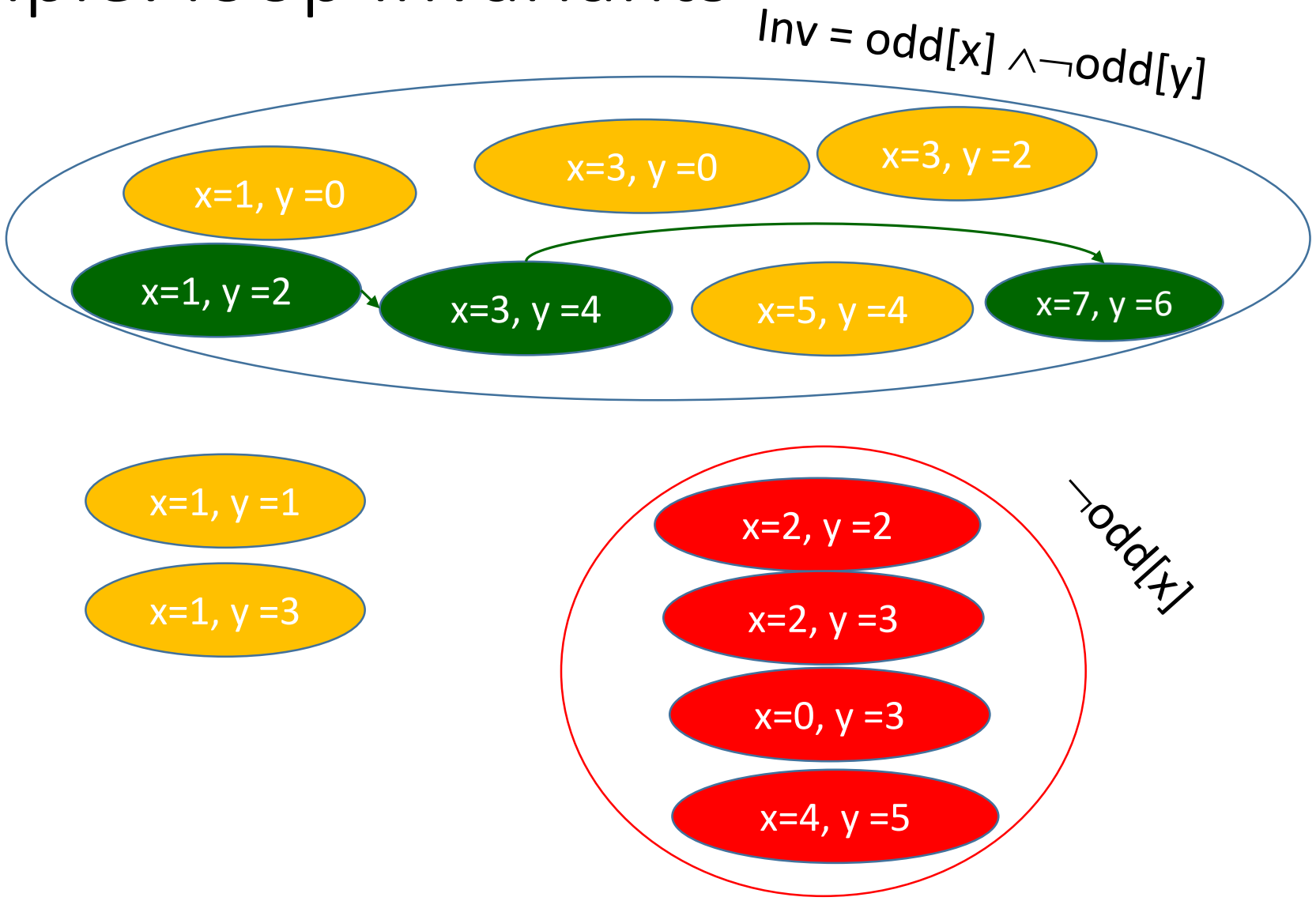
Simple Example: loop Invariants

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```

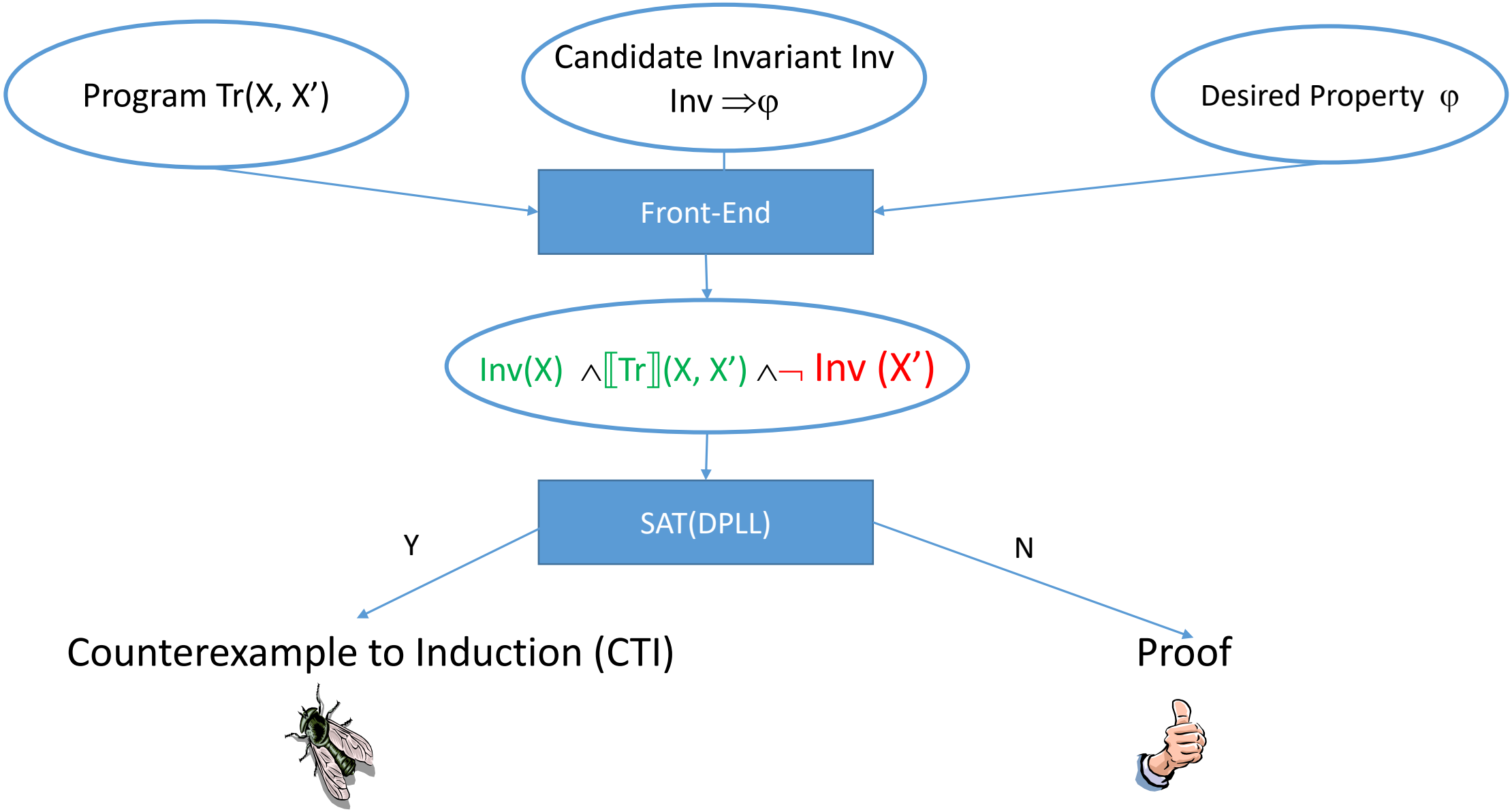


Simple Example: loop Invariants

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```



Deductive verification by reductions to SAT



Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```

Inv = at(3) ⇒ odd[x]

at(3) ⇒ odd[x]

Solver

Is there a behavior
of P that violates the inductiveness of I?

Y

Inv

¬Inv

3: odd[x]=1, odd[y]=1

odd[x]’ = (odd[x]∧¬odd[y])∨(¬odd[x]∧odd[y])
odd[y]’ = odd[y]

odd[x]’= 0, odd[y]’=1

Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4:  x:= x + y;
   5:  y := y + 2
}
6:
```

$Inv = at(3) \Rightarrow odd[x] \wedge \neg odd[y]$

$at(3) \Rightarrow odd[x]$

Solver

Is there a behavior of P that violates the inductiveness of I?

N

Proof

Inductive (loop) Invariants Mists and Reality

- Mist
  - An inductive invariant is a property that holds in every iteration
- Reality
  - An inductive invariant is a property that
    - Holds at every iteration
    - Suffices to prove that the invariant is preserved by any execution

Inductive Invariants Formally

- Inputs:
  - A set of states Σ
  - A set of initial states $\Sigma_0 \subseteq \Sigma$
  - A transition relation $\tau \subseteq \Sigma \times \Sigma$
- The set of reachable states is defined by:
  - reach[0] = Σ_0
  - reach[i+1] = $\{\sigma' \in \Sigma \mid \sigma \in \text{reach}[i] \wedge \langle\sigma, \sigma'\rangle \in \tau\}$
  - reach[∞] = reach[0] $\cup$ reach[1] $\cup$ …
- A set Inv$\subseteq\Sigma$ is inductive if
  - $\Sigma_0 \subseteq$ Inv (Initiation)
  - For all $\sigma \in$ Inv and $\langle\sigma, \sigma'\rangle \in \tau$, $\sigma' \in$ Inv (Consecution)
- A set Inv$\subseteq\Sigma$ is inductive for P $\subseteq\Sigma$ if it is inductive and Inv $\subseteq$ P
  - "The reason why P holds"

Simple Example

```
require x >= 1
int it = x;
int res = 1;
int z = 0
while it != 1 do
    res := res * it
    it := it -1
    z := z + 1
```

Property	Inductive
true	yes
$res \geq 0$	no
$res \geq 0 \wedge it \geq 1$	yes
$res \geq 0 \wedge it \geq 1 \wedge x \geq 1$	yes
$res \geq 0 \wedge it \geq 1 \wedge res = x!/it!$	yes
$res \geq 0 \wedge it \geq 1 \wedge res = x!/it! \wedge z+it = x$	yes

The **While** Programming Language

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S
- Use parenthesizes for precedence
- Informal Semantics
  - **skip** behaves like no-operation
  - Import meaning of arithmetic and Boolean operations

Example While Program

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

General Notations

- Syntactic categories
  - Var the set of program variables
  - Aexp the set of arithmetic expressions
  - Bexp the set of Boolean expressions
  - Stm set of program statements
- Semantic categories
  - Natural values Z={…, -2, -1, 0, 1, 2, …}
  - Truth values T={ff, tt}
  - $\sum$= Var $\rightarrow$ Z
  - Lookup in a state σ: σ x
  - Update of a state σ : σ [x $\mapsto$ 5]

Example State Manipulations

- $[x \mapsto 1, y \mapsto 7, z \mapsto 16]\ y =$
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16]\ t =$
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5] =$
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5]\ x =$
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5]\ y =$

Semantics of arithmetic expressions

- Assume that arithmetic expressions are side-effect free
- $A[\![Aexp]\!] : \Sigma \rightarrow Z$
- Defined by structural induction on the syntax tree
  - $A[\![n]\!] \sigma = n$
  - $A[\![x]\!] \sigma = \sigma\, x$
  - $A[\![e_1 + e_2]\!] \sigma = A[\![e_1]\!] \sigma + A[\![e_2]\!] \sigma$
  - $A[\![e_1 * e_2]\!] \sigma = A[\![e_1]\!] \sigma * A[\![e_2]\!] \sigma$
  - $A[\![- e_1]\!] \sigma = -A[\![e_1]\!] \sigma$

Semantics of Boolean expressions

- Assume that Boolean expressions are side-effect free
- $B[\![Bexp]\!] : \Sigma \rightarrow T$
- Defined by induction on the syntax tree
  - $B[\![\text{true}]\!] \sigma = tt$
  - $B[\![\text{false}]\!] \sigma = ff$
  - $B[\![e_1 = e_2]\!] \sigma =$

$$\begin{cases} tt \text{ if } A[\![e_1]\!] \sigma = A[\![e_2]\!] \sigma \\ ff \text{ if } A[\![e_1]\!] \sigma \neq A[\![e_2]\!] \sigma \end{cases}$$

  - $B[\![e_1 \wedge e_2]\!] \sigma =$

$$\begin{cases} tt \text{ if } B[\![e_1]\!] \sigma = tt \text{ and } B[\![e_2]\!] \sigma = tt \\ ff \text{ if } B[\![e_1]\!] \sigma = ff \text{ or } B[\![e_2]\!] \sigma = ff \end{cases}$$

  - $B[\![e_1 \geq e_2]\!] s =$

$$tt \text{ if } A[\![e_1]\!] \sigma \geq A[\![e_2]\!] \sigma$$
$$ff \text{ if } A[\![e_1]\!] \sigma < A[\![e_2]\!] \sigma$$

Natural Semantics for While

axioms

$[ass_{ns}]$ $\langle x := a, \sigma\rangle \rightarrow \sigma[x \mapsto \mathbf{A}[\![a]\!]\sigma]$

$[skip_{ns}]$ $\langle \mathbf{skip}, \sigma\rangle \rightarrow \sigma$

rules

$$[comp_{ns}]\ \frac{\langle S_1, \sigma\rangle \rightarrow \sigma', \langle S_2, \sigma'\rangle \rightarrow \sigma''}{\langle S_1; S_2, \sigma\rangle \rightarrow \sigma''}$$

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

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

Natural Semantics for While (More rules)

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

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

Simple Examples

- Let σ_0 be the state which assigns zero to all program variables
- Assignments
  $[ass_{ns}]$ $\langle x := x+1, \sigma_0\rangle \rightarrow [x \mapsto 1]$
- Skip statement
  $[skip_{ns}]$ $\langle skip, \sigma_0\rangle \rightarrow \sigma_0$
- Composition

$$\frac{[comp_{ns}]\ \langle skip, \sigma_0\rangle \rightarrow \sigma_0,\ \langle x := x+1, \sigma_0\rangle \rightarrow [x \mapsto 1]}{\langle skip; x := x+1, \sigma_0\rangle \rightarrow [x \mapsto 1]}$$

A Derivation Tree

- A "proof" that $<S, \sigma> \rightarrow \sigma'$
- The root of tree is $<S, \sigma> \rightarrow \sigma'$
- Leaves are instances of axioms
- Internal nodes rules
  - Immediate children match rule premises
- Simple Example

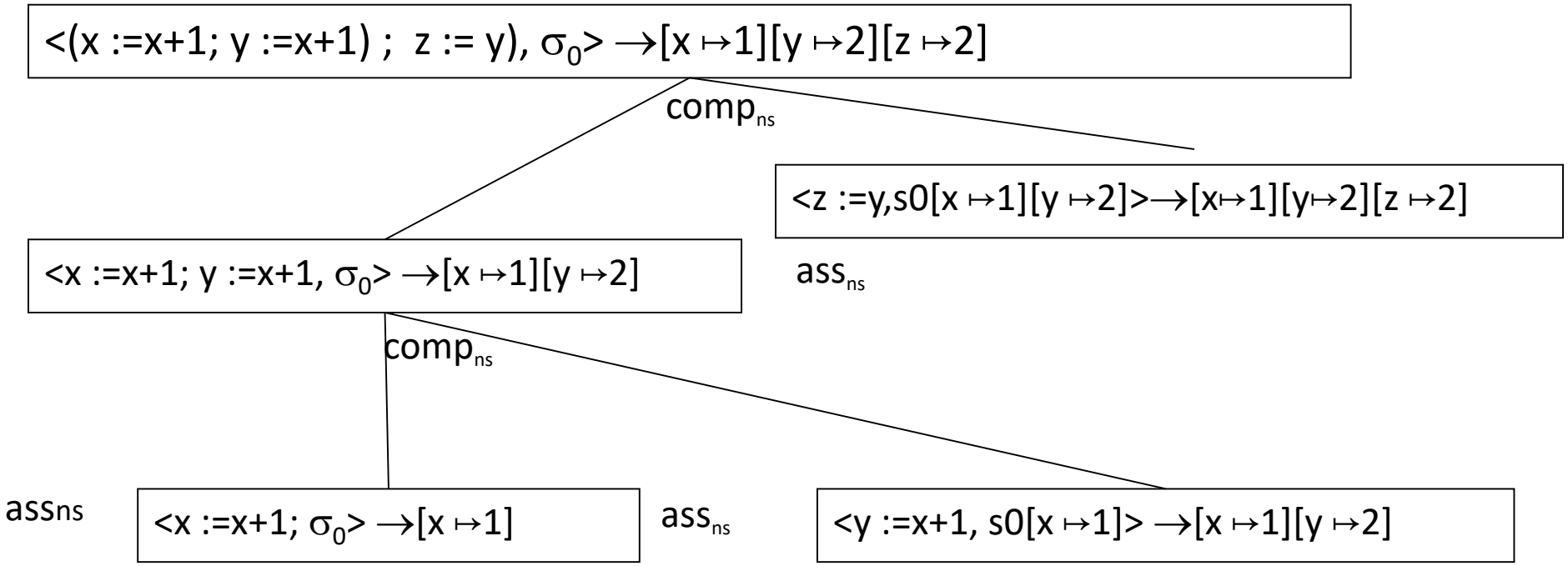
$<skip; x := x + 1, \sigma_0> \rightarrow [x \mapsto 1]$

$comp_{ns}$

$<skip, \sigma_0> \rightarrow \sigma_0$

$< x := x + 1, \sigma_0> \rightarrow [x \mapsto 1]$

An Example Derivation Tree

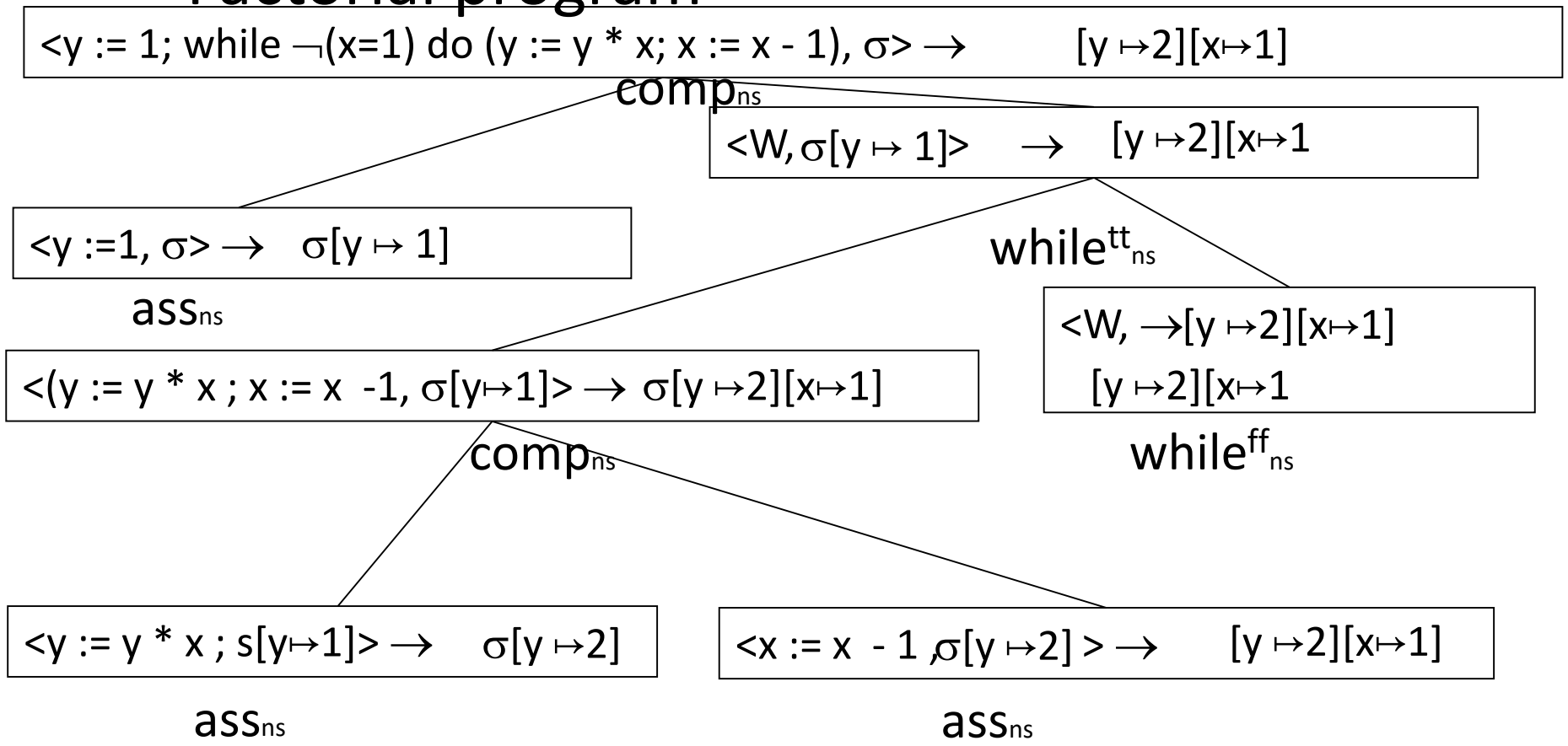


Top Down Evaluation of Derivation Trees

- Given a program S and an input state σ
- Find an output state σ' such that
  $<S, \sigma> \rightarrow \sigma'$
- Start with the root and repeatedly apply rules until the axioms are reached
- Inspect different alternatives in order
- In While σ' and the derivation tree is unique

Example of Top Down Tree Construction

- Input state s such that σ x = 2
- Factorial program



Program Termination

- Given a statement S and input σ
- S terminates on σ if there exists a state s' such that $<S, \sigma> \rightarrow \sigma'$
  - S loops on s if there is no state s' such that $<S, \sigma> \rightarrow \sigma'$
- Given a statement S
  - S always terminates if for every input state σ, S terminates on σ
  - S always loops if for every input state σ, S loops on σ

The Semantic Function

- The meaning of a statement S is defined as a partial function from Σ to Σ
- S_{ns}: **Stm** $\rightarrow \Sigma_{\perp} \rightarrow \Sigma_{\perp}$
- $S_{ns}[\![S]\!]\sigma = \sigma'$ if $\langle S, \sigma\rangle \rightarrow \sigma'$ and otherwise

  $S_{ns}[\![S]\!]\sigma = \perp$

- $S_{ns}[\![S]\!] \perp = \perp$

- Examples
  - $S_{ns}[\![skip]\!]\sigma = \sigma$
  - $S_{ns}[\![x := 1]\!]\sigma = \sigma[x \mapsto 1]$
  - $S_{ns}[\![\text{while true do skip}]\!]\sigma = \perp$

An Assertion Language

- Extend Bexp
- Allow quantifications
  - $\forall i$: …
  - $\exists i$: …
    - $\exists i.\ k=i\times l$
- Import well known mathematical concepts
  - $n! \triangleq n \times (n-1) \times \cdots 2 \times 1$

Assertion Language

Aexpv

$a := n \mid X \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$

Assn

$A := \text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid \neg A \mid$

$A_0 \Rightarrow A_1 \mid \forall i.\ A \mid \exists i.\ A$

Example

```
while ¬(M=N) do
        if M ≤ N
            then N := N – M
            else  M := M - N
```

Example

```
int power(int x, unsigned int y)
{
    int temp;
    if (y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
```

Free and Bound Variables

- An integer variable is bound when it occurs in the scope of a quantifier
- Otherwise it is free
- Examples $\exists i.\ k=i\times L \quad (i+100\leq 77)\wedge\forall i.j+1=i+3)$

$FV(n) = FV(X) = \varnothing$ $\qquad$ $FV(i) = \{i\}$

$FV(a_0 + a_1)=FV(a_0-a_1)=FV(a_0\times a_1) = FV(a_0) \cup FV(a_1)$

$FV(true)=FV(false)= \varnothing \quad FV(a_0 = a_1)=FV(a_0 \leq a_1)= FV(a_0) \cup FV(a_1)$

$FV(A_0\wedge A_1)=FV(A_0\vee A_1) =FV(A_0\Rightarrow A_1)= FV(A_0) \cup FV(A_1)$

$FV(\neg A)=FV(A)$

$FV(\forall i.\ A)=FV(\exists i.\ A)= FV(A)\setminus \{i\}$

Substitution Arithmetic Expression

- Visualization of arithmetic expression

  ---i---i----

- Consider a “pure” arithmetic expression

  A[a/i] ---a---a---

$n[a/i] = n$

$X[a/i]=X$

$i[a/i] = a$

$j[a/i] = j$

$(a_0 + a_1)[a/i] = a_0[a/i] + a_1/[a/i]$

$(a_0 - a_1)[a/i] = a_0[a/i] - a_1[a/i]$

$(a_0 \times a_1)[a/i] = a_0[a/i] \times a_1[a/i]$

Substitution Assertion

- Visualization of an assertion A

  ---i---i----

- Consider a “pure” arithmetic expression

  A[a/i] ---a---a---

$true[a/i] = true$ $\qquad$ $false[a/i]=false$

$(a_0 = a_1)[a/i] = (a_0/[a/i] = a_1[a/i])$ $\qquad$ $(a_0 \leq a_1)[a/i] = (a_0/[a/i] \leq a_1[a/i])$

$(A_0 \wedge A_1)[a/i] = (A_0[a/i] \wedge A_1[a/i])$ $\qquad$ $(A_0 \vee A_1)[a/i]= (A_0[a/i] \vee A_1[a/i])$

$(A_0 \Rightarrow A_1)[a/i] = (A_0[a/i] \Rightarrow A_1[a/i])[a/i]$

$(\neg A)[a/i] = \neg(A[a/i])$

$(\forall i.A)[a/i] = \forall i.\ A$ $\qquad$ $(\forall j.A)[a/i] = (\forall j.\ A[a/i])$

$(\exists i.A)[a/i] = \exists i.\ A$ $\qquad$ $(\exists j.A)[a/i] = (\exists j.\ A[a/i])$

Location Substitution

- Visualization of an assertion A

  ---X---X----

- Consider a “pure” arithmetic expression

  A[a/X] ---a---a---

Example Assertions

- i is a prime number
- i is the least common multiple of j and k

Semantics of Assertions

- An interpretation $Iz: zvar \rightarrow Z$
- The meaning of Aexpv
  - $Av[\![n]\!]Iz\ \sigma = n$
  - $Av[\![X]\!]Iz\ \sigma = \sigma(X)$
  - $Av[\![i]\!]Iz\ \sigma = Iz(i)$
  - $Av[\![a_0+a_1]\!]\ Iz\sigma = Av[\![a_0]\!]Iz\sigma + Av\ [\![a_1]\!]\ Iz\sigma$
  - …
- For all $a \in$ Aexp states σ and Interpretations Iz
  - $A[\![a]\!]\sigma = Av[\![a]\!]Iz\ \sigma$

Semantics of Assertions (II)

- Iz[n/i] change i in Iz to n
- For Iz and $\sigma \in \Sigma_\perp$, define $\sigma \models^{Iz} A$ by structural induction
  - $\sigma \models^{Iz}$ true
  - $\sigma \models^{Iz} (a_0 = a_1)$ if $Av[\![a_0]\!]\ Iz\ \sigma = Av[\![a_1]\!]\ Iz\ \sigma$
  - $\sigma \models^{Iz} (A \wedge B)$ if $\sigma \models^{Iz} A$ and $\sigma \models^{Iz} B$
  - $\sigma \models^{Iz} \neg A$ if not $\sigma \models^{Iz} A$
  - $\sigma \models^{Iz} A \Rightarrow B$ if (not $\sigma \models^{Iz} A$) or $\sigma \models^{Iz} B$)
  - $\sigma \models^{Iz} \forall i.A$ if $\sigma \models^{Iz[n/i]} A$ for all $n \in N$
  - $\perp \models A$

Partial Correctness Assertions

- $\{P\}\ S\ \{Q\}$
  - $P, Q \in$ Assn and $S \in$ Stmt
- For a state $\sigma \in \Sigma_\perp$ and interpretation Iz
  - $\sigma \models^{Iz} \{P\}\ S\ \{Q\}$ if $(\sigma \models^{Iz} P \Rightarrow S_{ns}[\![S]\!]\sigma \models^{Iz} Q)$
- Validity
  - When $\forall \sigma \in \Sigma_\perp$, $\sigma \models^{Iz} \{P\}\ S\ \{Q\}$ we write
    - $\models^{Iz} \{P\}S\{Q\}$
  - When $\forall \sigma \in \Sigma_\perp$, and I $\sigma \models^{Iz} \{P\}\ S\ \{Q\}$ we write
    - $\models \{P\}S\{Q\}$
    - $\{P\}S\{Q\}$ is valid

Example Hoare Triples

{true} X := 12 { X = 12 }

{true} X := 12 { X = 0 }

{true} X := 12 { X > 0 }

{true} X := 12 { true }

{false} X := 12 { X=0 }

{X=i} X := X+1 { X=i+1 }

{true} while true do skip { false }

The extension of an assertion

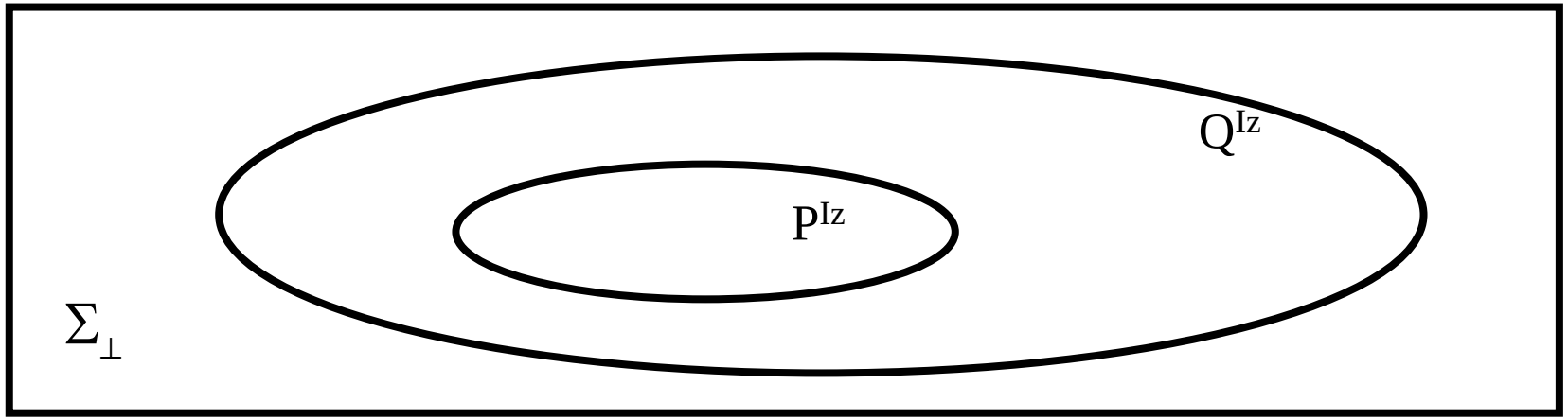
$$A^{Iz} \doteq \{\sigma \in \Sigma_{\perp} \mid \sigma \models^{Iz} A\}$$

The extension of assertions

Suppose that $\models (P \Rightarrow Q)$

Then for any interpretation Iz
$\forall \sigma \in \Sigma_{\perp}.\ \sigma \models^{Iz} P \Rightarrow \sigma \models^{Iz} Q$

$P^{Iz} \subseteq Q^{Iz}$

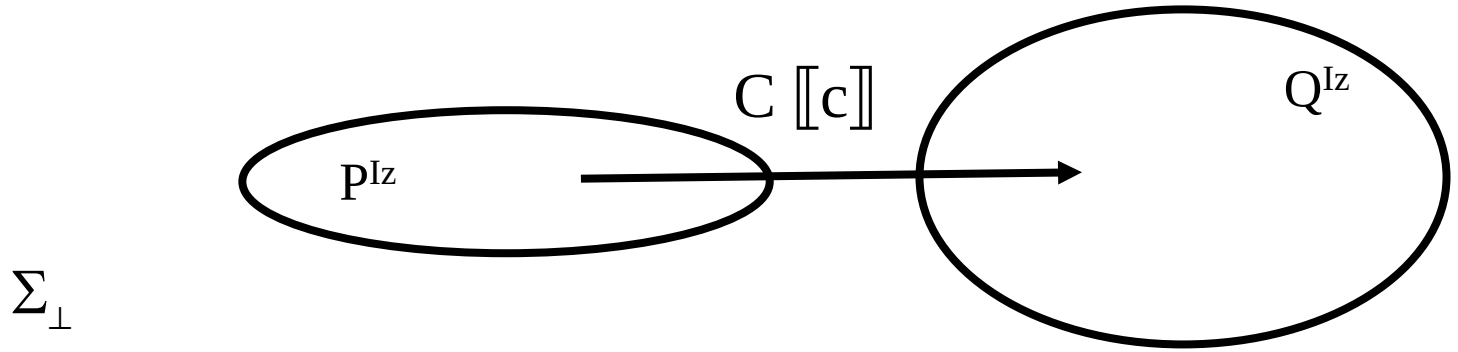


The extension of Hoare Triples

Suppose that $\models\{P\}S\{Q\}$

Then for any interpretation Iz
$\forall\sigma \in\Sigma_{\perp}.\ \sigma \models^{Iz} P \Rightarrow S_{ns}\ [\![S]\!]\sigma \models^{Iz} Q$

$S_{ns}\ [\![S]\!]P^{Iz}\subseteq Q^{Iz}$



Hoare Proof Rules for Partial Correctness (Take 1)

$\{A\}$ skip $\{A\}$

$\{B[a/X]\}$ X:=a $\{B\}$

$$\frac{\{P\}\ S_0\ \{C\}\ \{C\}\ S_1\ \{Q\}}{\{P\}\ S_0;S_1\{Q\}}$$

$$\frac{\{P\wedge b\}\ S_0\ \{Q\}\ \{P \wedge\neg b\}\ S_1\ \{Q\}}{\{P\}\ \text{if } b \text{ then } S_0 \text{ else } S_1\{Q\}}$$

$$\frac{\{Inv\wedge b\}\ S\ \{Inv\}}{\{Inv\}\ \text{while } b \text{ do } S\{Inv\wedge\neg b\}}$$

$$\frac{\models P \Rightarrow P'\quad \{P'\}\ S\ \{Q'\}\ \models Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Simple Example

$\{X = 5\}$

$Y := X$

$\{Y \geq 0\}$

Hoare Proof Rules for Partial Correctness

$\{A\}$ skip $\{A\}$

$\{B[a/X]\}$ X:=a $\{B\}$

$$\frac{\{P\}\ S_0\ \{C\}\ \{C\}\ S_1\ \{Q\}}{\{P\}\ S_0;S_1\{Q\}}$$

$$\frac{\{P\wedge b\}\ S_0\ \{Q\}\ \{P\wedge\neg b\}\ S_1\ \{Q\}}{\{P\}\ \text{if } b \text{ then } S_0 \text{ else } S_1\{Q\}}$$

$$\frac{\{Inv\wedge b\}\ S\ \{Inv\}}{\{Inv\}\ \text{while } b \text{ do } S\{Inv\wedge\neg b\}}$$

$$\frac{\models P \Rightarrow P'\quad \{P'\}\ S\ \{Q'\}\quad \models Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Simple Example

$\{X = 5\}$

$Y := X$

$\{Y \geq 0\}$

Example

$\{X = n \wedge n \geq 0\}$

Y := 1;

$\{X = n \wedge Y=1 \wedge n \geq 0\}$

while X > 0 do $\{X \geq 0 \wedge n \geq 0 \wedge Y=n!/X!\}$

$\{X > 0 \wedge n \geq 0 \wedge Y=n!/X!\}$

Y := X $\times$Y;

$\{X > 0 \wedge n \geq 0 \wedge Y=n!/(X-1)!\}$

X := X – 1

$\{X > 0 \wedge n \geq 0 \wedge Y=n!/X!\}$

$\{Y = n!\}$

Example Formal

$\{X = n \wedge n \geq 0\}$ Y :=1 $\{X = n \wedge Y=1 \wedge n \geq 0\}$

$\{X = n \wedge n \geq 0\}$ Y :=1 $\{X \geq 0 \wedge n \geq 0 \wedge Y=n!/X!\}$

$\{X > 0 \wedge n \geq 0 \wedge Y=n!/X!\}$ Y := X $\times$Y; $\{X > 0 \wedge n \geq 0 \wedge Y=n!/(X-1)!\}$

$\{X > 0 \wedge n \geq 0 \wedge Y=n!/(X-1)!\}$ X := X-1; $\{X \geq 0 \wedge n \geq 0 \wedge Y=n!/X!\}$

$\{X > 0 \wedge n \geq 0 \wedge Y=n!/X!\}$ Y := X $\times$Y; X := X-1 $\{X \geq 0 \wedge n \geq 0 \wedge Y=n!/X!\}$

$\{ X \geq 0 \wedge n \geq 0 \wedge Y=n!/X! \wedge X>0\}$ Y := X $\times$Y; X := X-1 $\{X \geq 0 \wedge n \geq 0 \wedge Y=n!/X!\}$

$\{ X \geq 0 \wedge n \geq 0 \wedge Y=n!/X!\}$ while X > 0 do Y := X $\times$Y; X := X-1
$\{X \geq 0 \wedge n \geq 0 \wedge Y=n!/X! \wedge \neg X > 0 \}$

$\{ X \geq 0 \wedge n \geq 0 \wedge Y=n!/X!\}$ while X > 0 do Y := X $\times$Y; X := X-1 $\{Y=n! \}$

$\{ X=n \wedge n \geq 0\}$ Y :=1; while X > 0 do Y := X $\times$Y; X := X-1 $\{Y=n! \}$

Unsound Proof Rules for Partial Correctness

$\{A\}$ skip $\{B\}$

$\{A\}$ X:=a $\{A[a/X]\}$

$$\frac{\{P\}\ S_1\ \{C\}\ \{C\}\ S_0\ \{Q\}}{\{P\}\ S_0;S_1\{Q\}}$$

$$\frac{\{P\wedge b\}\ S_0\ \{Q\}\ \{P \wedge\neg b\}\ S_1\ \{Q\}}{\{P\}\ \text{if } b \text{ then } S_0 \text{ else } S_1\{Q\}}$$

$$\frac{\{Inv\wedge b\}\ S\ \{Q\}}{\{Inv\}\ \text{while } b \text{ do } S\{Q\wedge\neg b\}}$$

$$\frac{\models P' \Rightarrow P\ \ \{P'\}\ S\ \{Q'\} \models Q \Rightarrow Q'}{\{P\}\ S\ \{Q\}}$$

Incomplete Proof Rules for Partial Correctness

$\{A\}$ skip $\{A\}$

$\{B[a/X]\}$ X:=a $\{B\}$

$$\frac{\{P\}\ S_0\ \{P\}\quad \{P\}\ S_1\ \{P\}}{\{P\}\ S_0;S_1\{P\}}$$

$$\frac{\{P\}\ S_0\ \{Q\}\quad \{P\ \}\ S_1\ \{Q\}}{\{P\}\ \text{if b then } S_0 \text{ else } S_1\{Q\}}$$

$$\frac{\{Inv\}\ S\ \{Inv\}}{\{Inv\}\ \text{while b do } S\{Inv\wedge\neg b\}}$$

$$\frac{\models P \Rightarrow P'\quad \{P'\}\ S\ \{Q'\}\quad \models Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Soundness

- Every theorem obtained by the rule system is valid
  - $\vdash\{P\}\ c\ \{Q\} \Rightarrow \models\{P\}\ c\ \{Q\}$
- The system can be implemented (HOL, LCF, Coq)
  - Requires user assistance
- Proof of soundness
  - Every rule preserves validity (Theorem 6.1)

Soundness of skip axiom

$$\models \{A\}\ \text{skip}\ \{A\}$$

Soundness of the assignment axiom

$$\models \{B[a/X]\}\ X{:=}a\ \ \{B\}$$

Soundness of the sequential composition rule

- Assume that
  $\models \{P\}\ S_0\ \{C\}$
  and
  $\models \{C\}\ S_1\ \{Q\}$
- Show that
  $\models \{P\}\ S_0;S_1\{Q\}$

Soundness of the conditional rule

- Assume that

  $\models \{P \wedge b\}\ S_0\ \{Q\}$

  and

  $\models \{P \wedge \neg b\}\ S_1\ \{Q\}$
- Show that

  $\models \{P\}$ if b then S_0 else $S_1 \{Q\}$

Soundness of the while rule

- Assume that

  $\models \{Inv \wedge b\}\ S\ \{Inv\}$

- Show that

  $\models \{Inv\}\ \text{while } b \text{ do } S\ \{Inv \wedge \neg b\}$

Soundness of the consequence rule

- Assume that
  $\models \{P'\}\ S\ \{Q'\}$
  and
  $\models P \Rightarrow P'$
  and
  $\models Q' \Rightarrow Q$
- Show that
  $\models \{P\}\ S\ \{Q\}$

(Ideal) Completeness

- Every valid theorem can be proved by the rule system
- For every P and Q such that $\models \{P\}\ S\ \{Q\}$
  there exists a proof such $\vdash \{P\}\ S\ \{Q\}$
- But what about Gödel's incompleteness?
  $\models \{true\}$ skip $\{Q\}$
- What does $\models \{true\}\ S\ \{false\}$ mean?

Relative Completeness (Chapter 7)

- Assume that every math theorem can be proved
  $\models \{P\}\ S\ \{Q\}$ implies $\vdash \{P\}\ S\ \{Q\}$

Relative completeness of composition rule

- Prove that $\{P\}\ S_0;S_1\{Q\}$
- Does there exist an assertion I such that

  $\models\{P\}\ S_0\ \{C\}$

  and

  $\models\{I\}\ S_1\ \{Q\}$

Precise Hoare Triples

- If {P} S {Q} and {P} S {R} then what can we tell?
- The most precise Q such that {P}S{Q} is called the strongest (liberal) postcondition of S with respect to P
- Denoted by sp(S, P)
- An imperative program is a predicate transformer

Weakest (Liberal) Precondition

- If {P} S {Q} and {R} S {Q} then what can we tell?
- The most general P such that {P}S{Q} is called the weakest (liberal) precondition of S with respect to Q
- Denoted by wp(S, Q)
- An imperative program is a predicate transformer

Hoare Triples vs. sp vs. wp

- {P} S {Q} holds when $P \Rightarrow wp(S, Q)$ and $sp(S, p) \Rightarrow Q$

Weakest (Liberal) Precondition

- wp(S, Q) – the weakest condition such that every terminating computation of S results in a state satisfying Q
- $[\![wp^l(S, Q)]\!] = \{\sigma \in \Sigma^{\perp} \mid S[\![S]\!]\sigma \models^l Q\}$
- [Can employ predicate transformer semantics to formally define the meaning (Chapter 7.5)]
- Prove that $\{P\}\ S_0;S_1\{Q\}$ by proving
  $\models\{P\}\ S_0\ \{I\}$
  and
  $\models\{I\}\ S_1\ \{Q\}$ where $I=wp(S_1, Q)$
- $\models\{P\}\ S\ \{Q\}$ iff for all $I\ [\![\ P\ ]\!] \subseteq [\![wp^l(S, Q)]\!]$
- $\models\{P\}\ S\ \{Q\}$ iff for $P \Rightarrow wp(S, Q)$

The rule of excluded miracle

- wp(S, false) =

Proving that programs correct: skip

- wp(skip, Q) = Q
- Example
  - wp(skip, $X^n+Y^n=Z^n$)=

To prove {P} skip {Q}, show that P $\Rightarrow$ Q

Proving that programs correct: assignments

- wp(X :=a, Q) = Q[a/X]
- Examples
  - wp(X := X+1, X= 7)=
  - wp(X := X+1, X $\leq$7) =
  - wp(X := 15, X $\leq$10) =
  - wp(Y := X+3*Y, X $\leq$10) =

To prove {P} X :=a {Q}, show that P $\Rightarrow$ Q[a/X]

Proving that programs correct: pointers

- wp(*X :=a, Q) =

Dafny Example

```
method foo(x: int) returns (y: int)
requires x > 0;
ensures y > 1;
 {
 y := x + 1;
 }
```

http://rise4fun.com/Dafny
http://research.microsoft.com/en-us/um/people/leino/papers/krml220.pdf

Proving that programs correct: sequential composition

- wp(S1; S2, Q) = wp(S1, wp(S2, Q))
- Examples
- wp(y := y + 1; x := x + 3, $y \leq 10 \wedge 3 \leq x$)=

To prove {P} S1; S2 {Q}, show that P $\Rightarrow$ wp(S1;S2, Q)

Proving that programs correct: Conditional Statements

- wp(if b then S_0 else S_1, Q) =
  $b \wedge wp(S_0, Q) \vee \neg b \wedge wp(S_1, Q)$
- $b \wedge wp(S_0, Q)$ is the set of states which path the if condition
- $\neg b \wedge wp(S_1, Q)$ is the set of states which path the else condition
- wp(if $x < y$ then $z := y$ else $z := x$, $0 \leq z$)=
- wp(if $x \neq 10$ then $x := x + 1$ else $x := x + 2$, $x \leq 10$)

Proving that the programs are correct: Loops

- To prove that {P} S {Q} where S = while C do S’ do:

Example: Array Sum

$\{N \geq 0\}$

```
k := 0;
s := 0;
while (k != N) {
s := s + a[k];
k := k + 1
 }
```

$\{s = \sum_{1 \leq i \leq N} a[i]\}$

Exercise Computing Cubes

```
method Cube(N: int) returns (c: int)
requires 0 <= N;
ensures c == N*N*N;
 {
  c := 0;
  var n := 0;
  var k := 1;
  var m := 6;
  while (n < N) {
   c := c + k;
   k := k + m;
   m := m + 6;
   n := n + 1;
       }
  }
```

```
invariant n <= N;
invariant c == n*n*n;
invariant k == 3*n*n + 3*n + 1;
invariant m == 6*n + 6;
decreases (N - n);
```

Proving programs correct: Procedures

- Suppose you have a program

```
method M( ){

P( );
...
}
```

- Proof strategy:
  - Check that precondition of P is valid at the call
  - Assume that all the variables potentially changed by P are “havoked”
  - Assume that the postcondition of P holds after the call
  - Prove that the postcondition M holds at end of M
  - Technically implemented with wp and “projection”

Proving programs correct: Procedures

```
method Ackermann(m: int, n: int) returns (r: int)
decreases m, n;
requires m >= 0 && n >= 0;
ensures r > 0;
 {
  if (m <= 0) {
     r := n + 1;
    }
   else if (n <= 0) {
      r := Ackermann(m - 1, 1);
      }
   else {
      var z;
      z := Ackermann(m, n - 1);
      r := Ackermann(m - 1, z);
     }
 }
```

Procedures

Dafny also permits a “functional" notation:

```
function Ackermann(m: int, n: int): int
decreases m, n;
requires m >= 0 && n >= 0;
ensures Ackermann(m,n) > 0;
 {
  if m <= 0 then
    n + 1
   else if n <= 0 then
   Ackermann(m - 1, 1)
   else
   Ackermann(m - 1, Ackermann(m, n - 1))
 }
```

Proving Fibonacci

```
function Fib(n: nat): nat
{
if n < 2 then n else Fib(n - 1) + Fib(n-2)
}

method Compute_Fib(n: nat) returns (x: nat)
ensures x == Fib(n);
{
var i := 0;
x := 0;
var y := 1;
while (i < n) {
  x, y := y, x + y;
  i := i + 1;
   }
}
```

```
invariant n >= i && i >= 0;
invariant y == Fib(i + 1);
invariant x == Fib(i);
```

Mutual recursion example

```
method Odd(n: nat) returns (b: bool)
 ensures (b <==> (n % 2 == 1));
 {
  if (n == 0) {
    b := false; }
  else {
   b := Even(n - 1); }
 }
method Even(n: nat) returns (b: bool)
  ensures (b <==> (n % 2 == 0));
  {
   if (n == 0) {
     b := true; }
    else {
       b := Odd(n - 1); }
 }
```

Binary Search Example

```
method BinarySearch(a: array<int>, value: int)
 returns (index: int)
 {
 var low, high := 0, a.Length;
   while (low < high) {
   var mid := (low + high) / 2;
   if (a[mid] < value) {
     low := mid + 1;
    }
   else if (value < a[mid]) {
     high := mid;
     }
    else {
     return mid; }
    }
 return -1;
}
```

Binary Search (2)

```
method BinarySearch(a: array<int>, value: int)   returns (index: int)
 requires a != null && 0 <= a.Length;
 requires forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k];
 ensures  0 <= index ==> index < a.Length && a[index] == value;
 ensures index < 0 ==>  forall k :: 0 <= k < a.Length ==> a[k] != value;
 {
 var low, high := 0, a.Length;
 while (low < high) {
 ...
 }
return -1;
 }
```

Binary Search(3)

```
var low, high := 0, a.Length;
while (low < high)
invariant 0 <= low <= high <= a.Length;
invariant forall i :: 0 <= i < a.Length &&
                !(low <= i < high) ==> a[i] != value;
 {
 var mid := (low + high) / 2;
 if (a[mid] < value) {
    low := mid + 1; }
 else if (value < a[mid]) {
    high := mid; }
   else {
   return mid; }
 }
return -1;
}
```

Relative Completeness

- For every command S and assertion B
  - there exists an assertion A, such that A=wp(S, B) (Theorem 7.5)
  - $\vdash$\{wp(S, B)\} S \{B\}(Lemma 7.6)
- Theorem 7.7: The proof system is relatively complete
  - $\models$\{P\} S \{Q\} implies $\vdash$ \{P\} S \{Q\}

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#, Dafny

WP Compound statements

- wp (skip,Q) = Q
- wp(X := e,Q) = Q[e / x]
- wp(S_1; S_2,Q) = wp$[\![S_1]\!]$(wp$[\![S_2]\!]$(Q))
- wp(if B then S_1 else S_2, Q =
  (B $\wedge$wp(S_1 ,Q) $\vee$($\neg$B$\wedge$ wp(S_2 ,Q))
- wp(while B do {I} S,Q) = I

VC rules

- $VC_{gen}(\{P\}\ S\ \{Q\}) = P \rightarrow wp(S,Q) \wedge \bigwedge VC_{aux}(S, Q)$
- $VC_{aux}(S, Q) = \{\}$ (for any atomic statement)
- $VC_{aux}(S_1; S_2, Q) =$
  $VC_{aux}(S_1, wp(S_2, Q)) \cup VC_{aux}(S_2, Q)$
- $VC_{aux}(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) =$
  $VC_{aux}(S_1, Q) \cup VC_{aux}(S_2, Q)$
- $VC_{aux}(\text{while } B \text{ do } \{I\}\ S, Q) = VC_{aux}(S, I) \cup$
  $\{I \wedge B \rightarrow wp(S,I)\} \cup$
  $\{I \wedge \neg B \rightarrow Q\}$

Soundness and completeness

- The set of rules of Hoare logic is sound: any derivable triple is valid
- Completeness: The set of rules of Hoare logic is relatively complete
  - if the language of state predicates is expressive enough, then any valid Hoare triple {P} S {Q} can be derived using the rules

Verification Success Stories

- Comcert C compiler verification
- Sel4 Kernel verification
- The IronClad/IronFleet Projects Microsoft Research
  - End-to-end verification of a distributed system with Dafny

Projects with Dafny

- Verify a real algorithm/data structure
- Examples
  - Simplified DPLL without learned clause
    - Two watched literal invariant
  - A dynamic programming algorithm
  - Your own algorithm with complex loops/data structures

Projects with axiomatic verification

- Develop a verifier for a small Python subset
  - Enough to cover few assignments in Mavo

Summary

- Axiomatic semantics provides an abstract semantics
- Can be used to explain programming
- Extensions
  - Procedures
  - Concurrency
  - Events
  - Rely/Guarantee
  - Heaps
- Can be automated
- But requires considerable effort