

Program Analysis and Verification

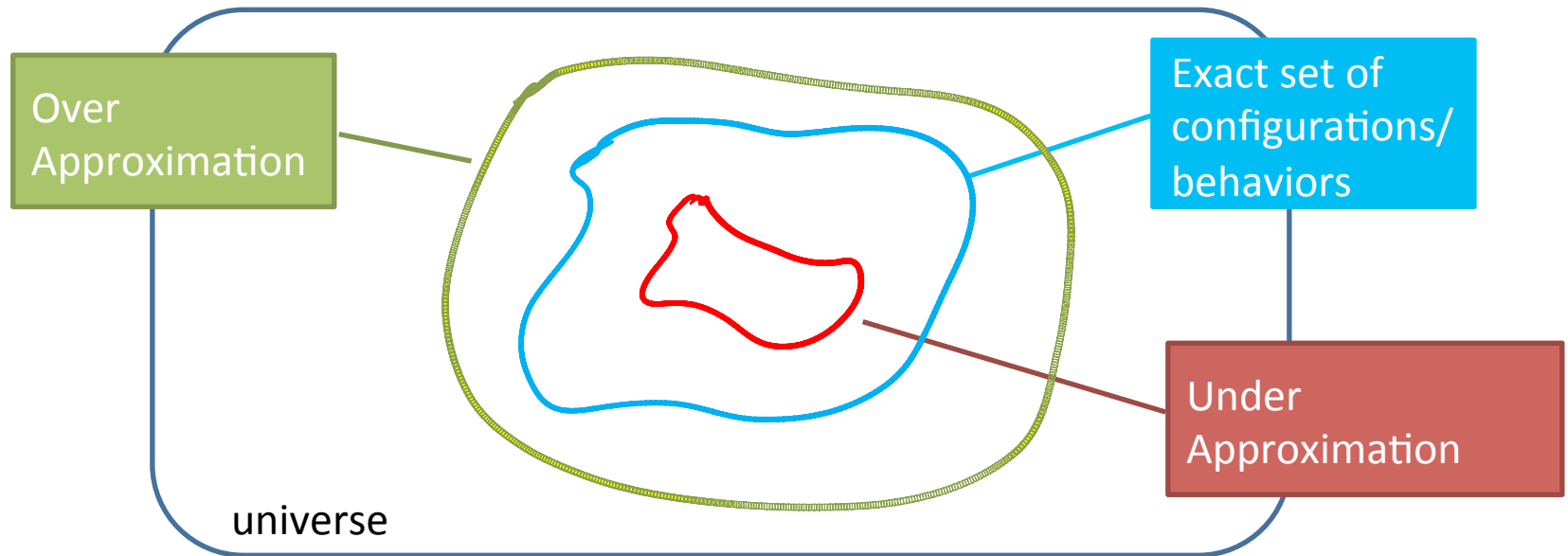
0368-4479

Noam Rinetzky

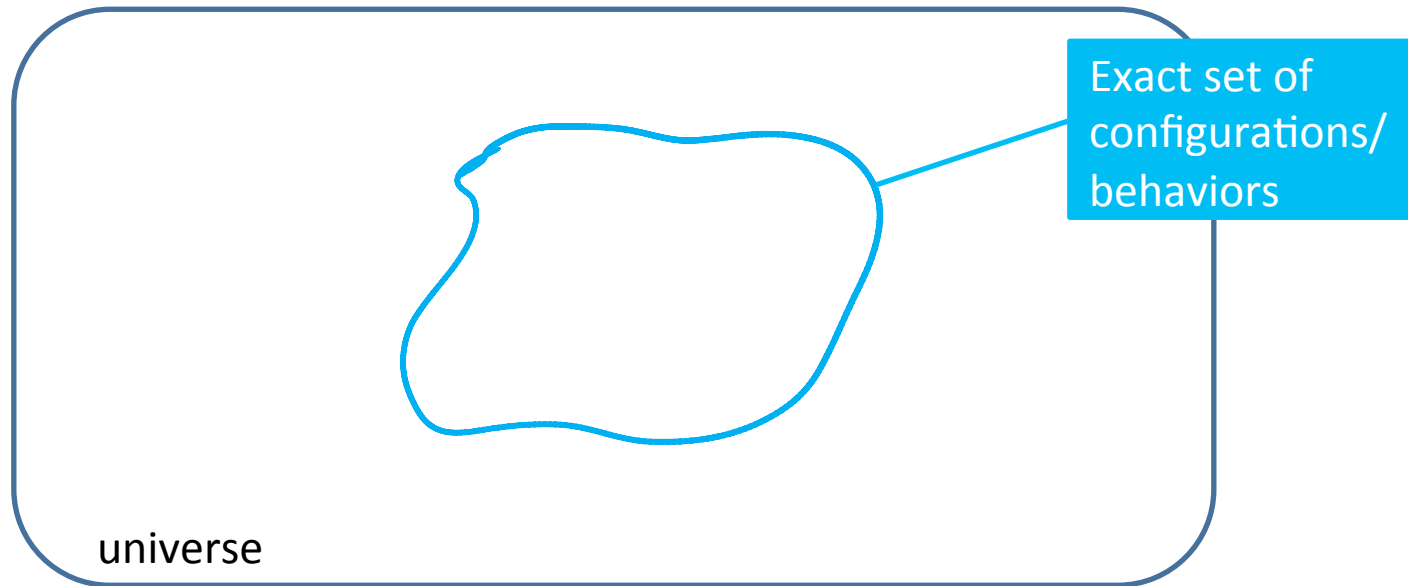
Lecture 2: Operational Semantics

Slides credit: Tom Ball, Dawson Engler, Roman Manevich, Erik Poll, Mooly Sagiv, Jean Souyris, Eran Tromer, Avishai Wool, Eran Yahav

Verification by over-approximation



Program semantics



Program analysis & verification

```
y = ?; x = ?;  
x = y * 2  
if (x % 2 == 0) {  
    y = 42;  
} else {  
    y = 73;  
    foo();  
}  
assert (y == 42);
```



What does P do?

```
y = ?; x = ?;  
x = y * 2  
if (x % 2 == 0) {  
    y = 42;  
} else {  
    y = 73;  
    foo();  
}  
assert (y == 42);
```



What does P mean?

```
y = ?; x = ?;  
x = y * 2  
if (x % 2 == 0) {  
  y = 42;  
} else {  
  y = 73;  
  foo();  
}  
assert (y == 42);
```

= ...

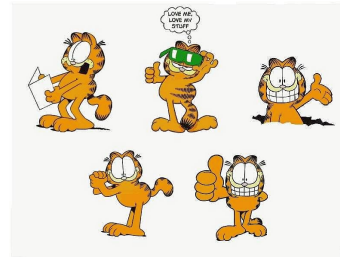
syntax

semantics

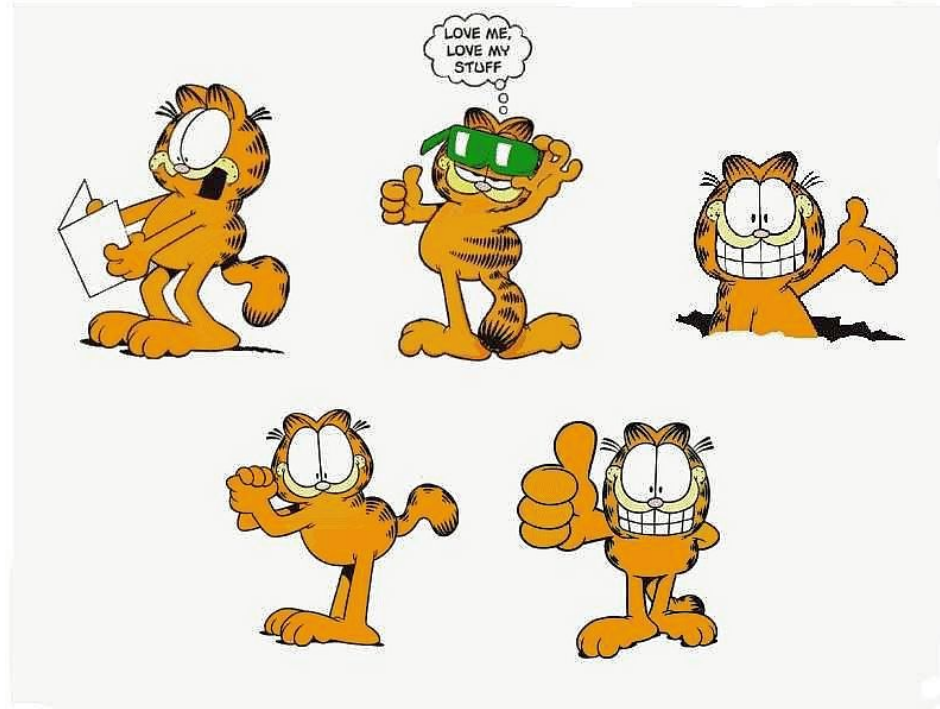
Program semantics

- State-transformer
 - Set-of-states transformer
 - Trace transformer
- Predicate-transformer
- Functions

- Cat-transformer



Program semantics & Verification



Agenda

- Operational semantics
 - Natural operational semantics
 - Structural operational semantics

What does P mean?

```
y = ?; x = ?;  
x = y * 2  
if (x % 2 == 0) {  
    y = 42;  
} else {  
    y = 73;  
    foo();  
}  
assert (y == 42);
```

= ...

syntax

semantics

“Standard” semantics

```
y = ?;  
x = y * 2  
if (x % 2 == 0) {  
    y = 42;  
} else {  
    y = 73;  
    foo();  
}  
assert (y == 42);
```

...-1,0,1,-1,0,1,...

y **x**

“Standard” semantics

(“state transformer”)

```
y = ?;  
x = y * 2  
if (x % 2 == 0) {  
    y = 42;  
} else {  
    y = 73;  
    foo();  
}  
assert (y == 42);
```

...-1,0,1,-1,0,1,...

y **x**

“Standard” semantics

(“state transformer”)

`y = ?;`

`y=3, x=9`

`x = y * 2`

`...-1,0,1,-1,0,1,...`

`if (x % 2 == 0) {`

y

x

`y = 42;`

`} else {`

`y = 73;`

`foo();`

`}`

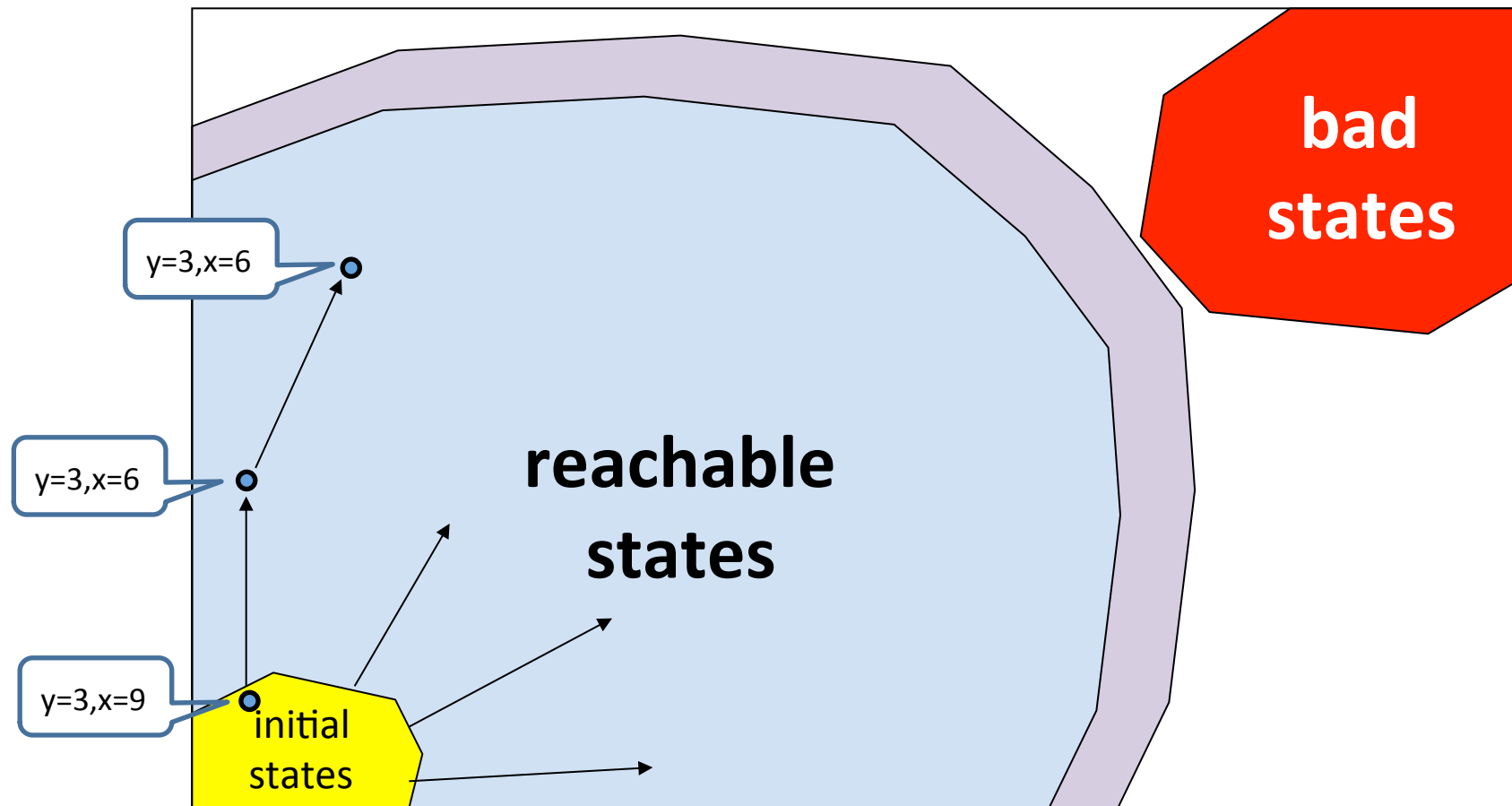
`assert (y == 42);`

“Standard” semantics

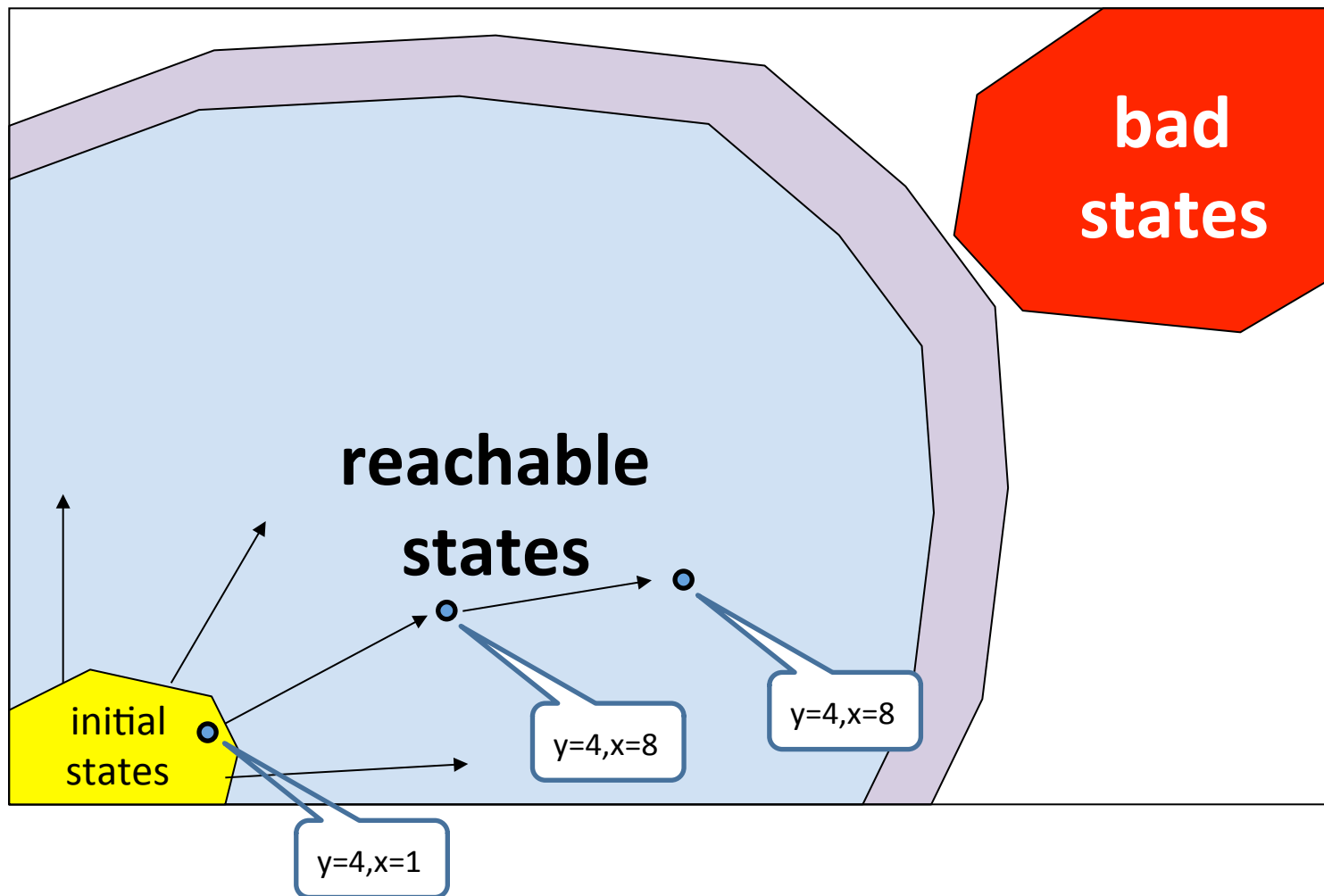
(“state transformer”)

<code>y = ?;</code>	<code>y=3, x=9</code>	
<code>x = y * 2</code>	<code>y=3, x=6</code>	<code>...-1,0,1, ...</code>
<code>if (x % 2 == 0) {</code>	<code>y=3, x=6</code>	<code>...-1,0,1,...</code>
<code>y = 42;</code>	<code>y=42, x=6</code>	y x
<code>} else {</code>		
<code>y = 73;</code>	<code>...</code>	
<code>foo();</code>	<code>...</code>	
<code>}</code>		
<code>assert (y == 42);</code>	<code>y=42, x=6</code>	

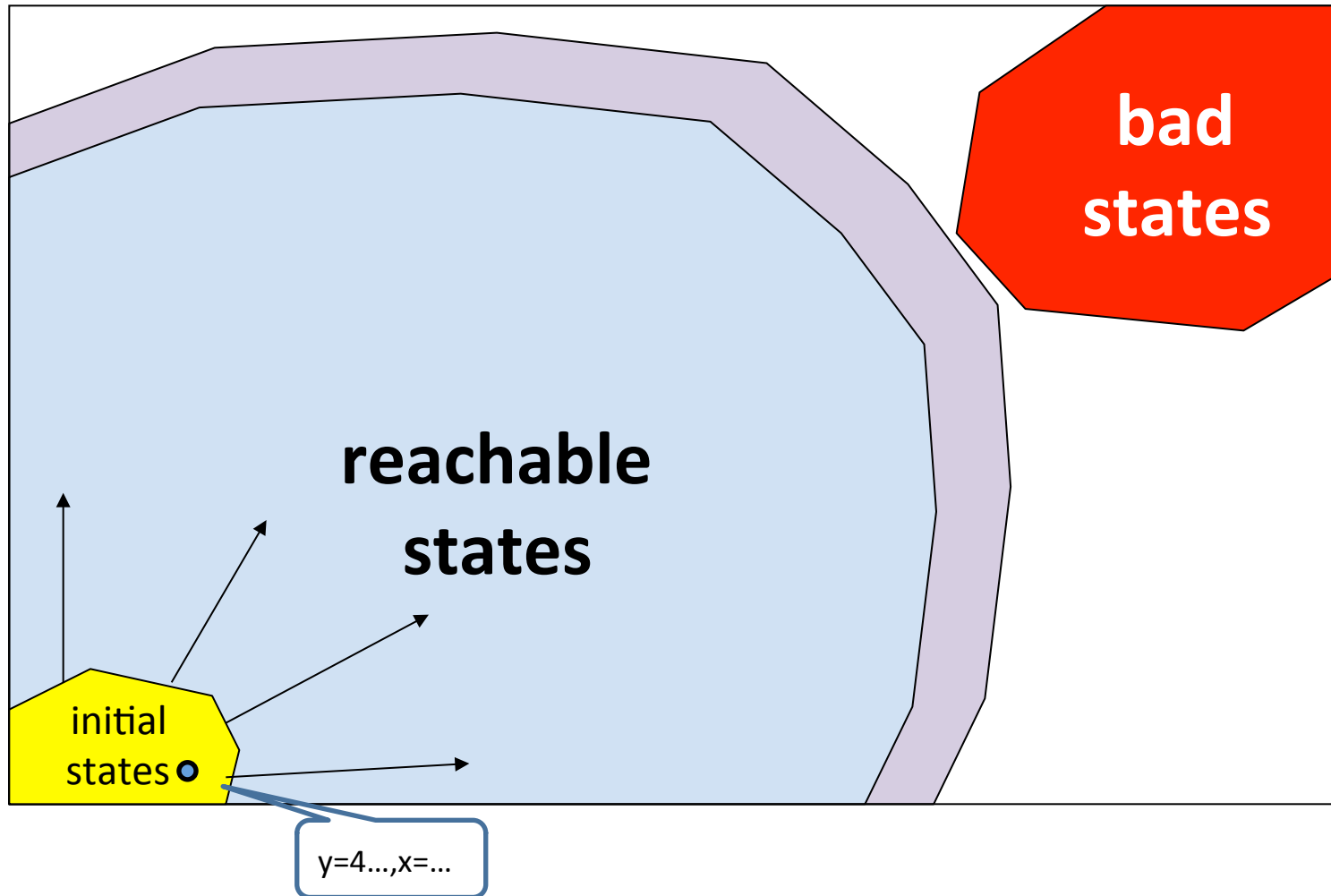
“State transformer” semantics



“State transformer” semantics

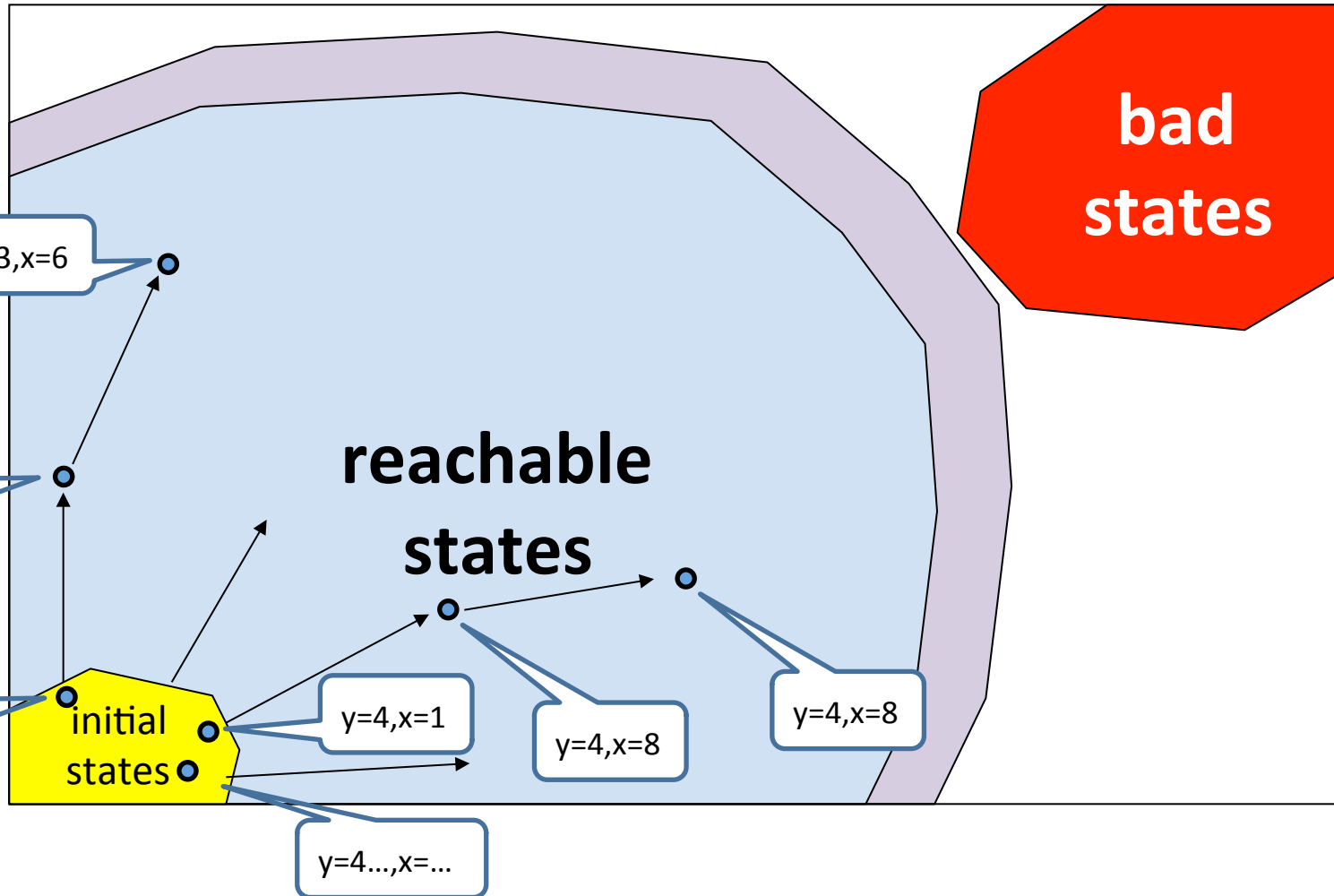


“State transformer” semantics



“State transformer” semantics

Main idea: find (properties of) all reachable states*



“Standard” (collecting) semantics

(“sets-of states-transformer”)

```
y = ?; x = ?;      {(y,x) | y,x ∈ Nat}
x = y * 2
if (x % 2 == 0) {
  y = 42;
} else {
  y = 73;
  foo();
}
assert (y == 42);
```

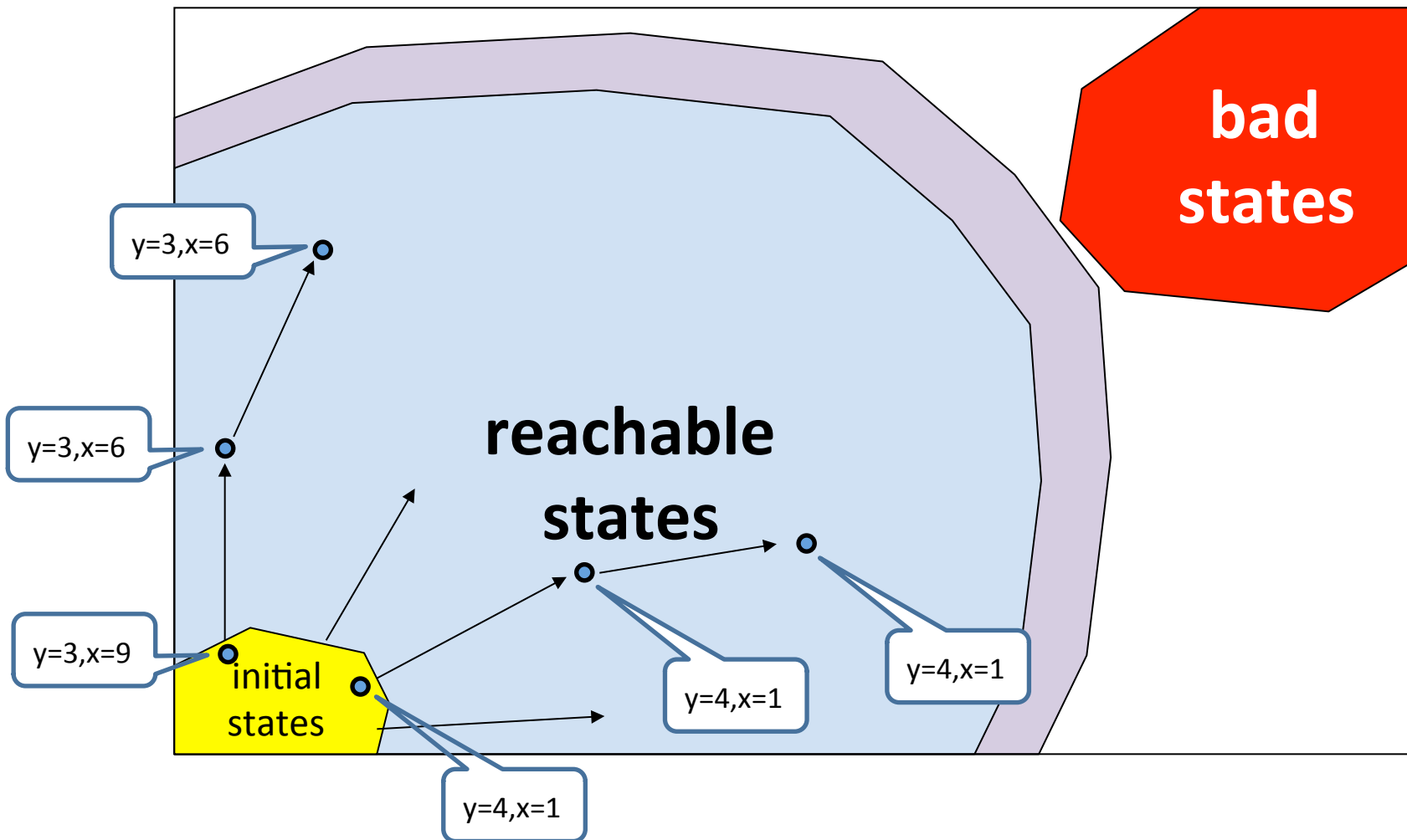
“Standard” (collecting) semantics

(“sets-of states-transformer”)

<code>y = ?;</code>	<code>{(y=3, x=9),(y=4,x=1),(y=..., x=...)}</code>
<code>x = y * 2</code>	<code>{(y=3, x=6),(y=4,x=8),(y=..., x=...)}</code>
<code>if (x % 2 == 0) {</code>	<code>{(y=3, x=6),(y=4,x=8),(y=..., x=...)}</code>
<code>y = 42;</code>	<code>{(y=42, x=6),(y=42,x=8),(y=42, x=...)}</code>
<code>}</code>	
<code>else {</code>	
<code>y = 73;</code>	<code>{}</code>
<code>foo();</code>	<code>{}</code>
<code>}</code>	
<code>assert (y == 42);</code>	<code>{(y=42, x=6),(y=42,x=8),(y=42, x=...)}</code>

Yes

“Set-of-states transformer” semantics



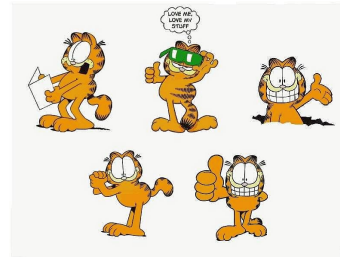
Program semantics

- State-transformer
 - Set-of-states transformer
 - Trace transformer
- Predicate-transformer
- Functions

Program semantics

- State-transformer
 - Set-of-states transformer
 - Trace transformer
- Predicate-transformer
- Functions

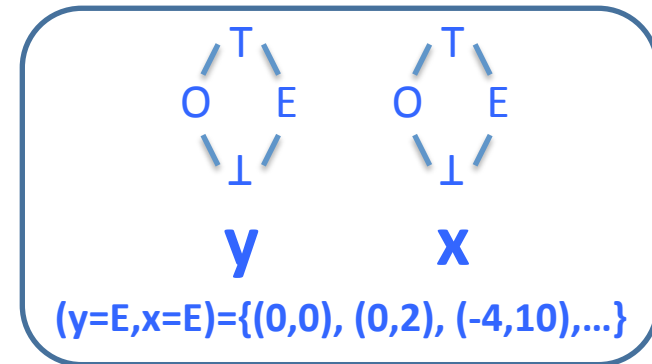
- Cat-transformer



“Abstract-state transformer” semantics

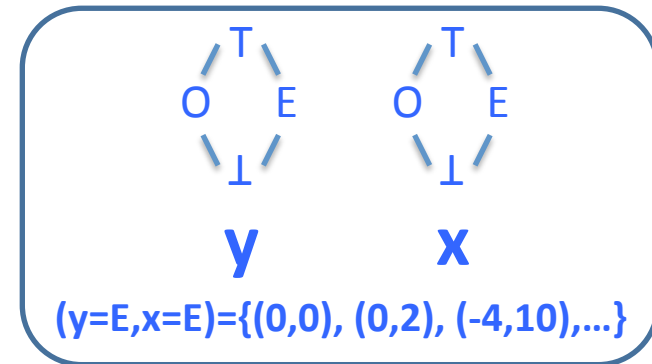
```
y = ?;  
x = y * 2  
if (x % 2 == 0) {  
    y = 42;  
} else {  
    y = 73;  
    foo();  
}  
assert (y == 42);
```

$y=T, x=T$



“Abstract-state transformer” semantics

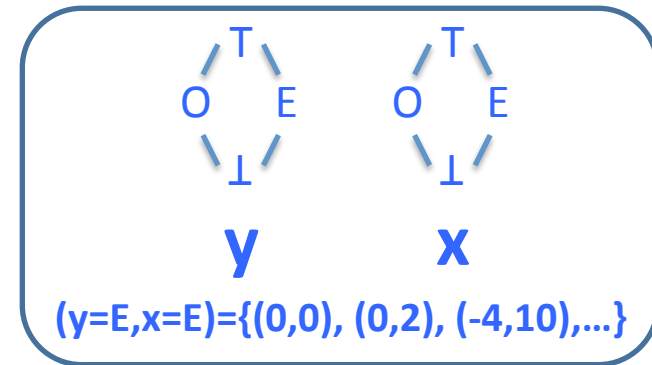
<code>y = ?;</code>	<code>y=T, x=T</code>
<code>x = y * 2</code>	<code>y=T, x=E</code>
<code>if (x % 2 == 0) {</code>	<code>y=T, x=E</code>
<code>y = 42;</code>	<code>y=T, x=E</code>
<code>} else {</code>	
<code>y = 73;</code>	<code>...</code>
<code>foo();</code>	<code>...</code>
<code>}</code>	
<code>assert (y == 42);</code>	<code>y=E, x=E</code>



Yes/?/No

“Abstract-state transformer” semantics

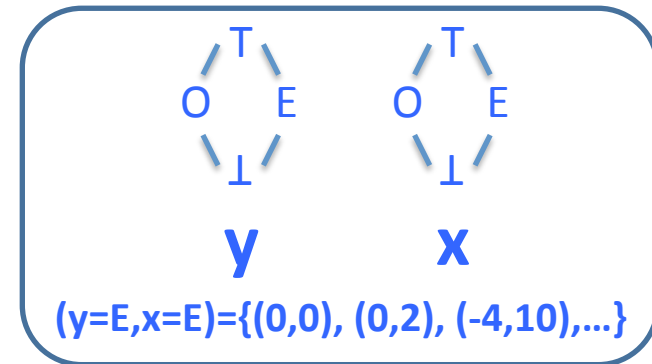
<code>y = ?;</code>	<code>y=T, x=T</code>
<code>x = y * 2</code>	<code>y=T, x=E</code>
<code>if (x % 2 == 0) {</code>	<code>y=T, x=E</code>
<code>y = 42;</code>	<code>y=T, x=E</code>
<code>} else {</code>	
<code>y = 73;</code>	<code>...</code>
<code>foo();</code>	<code>...</code>
<code>}</code>	
<code>assert (y == 42);</code>	<code>y=E, x=E</code>



Yes/?/No

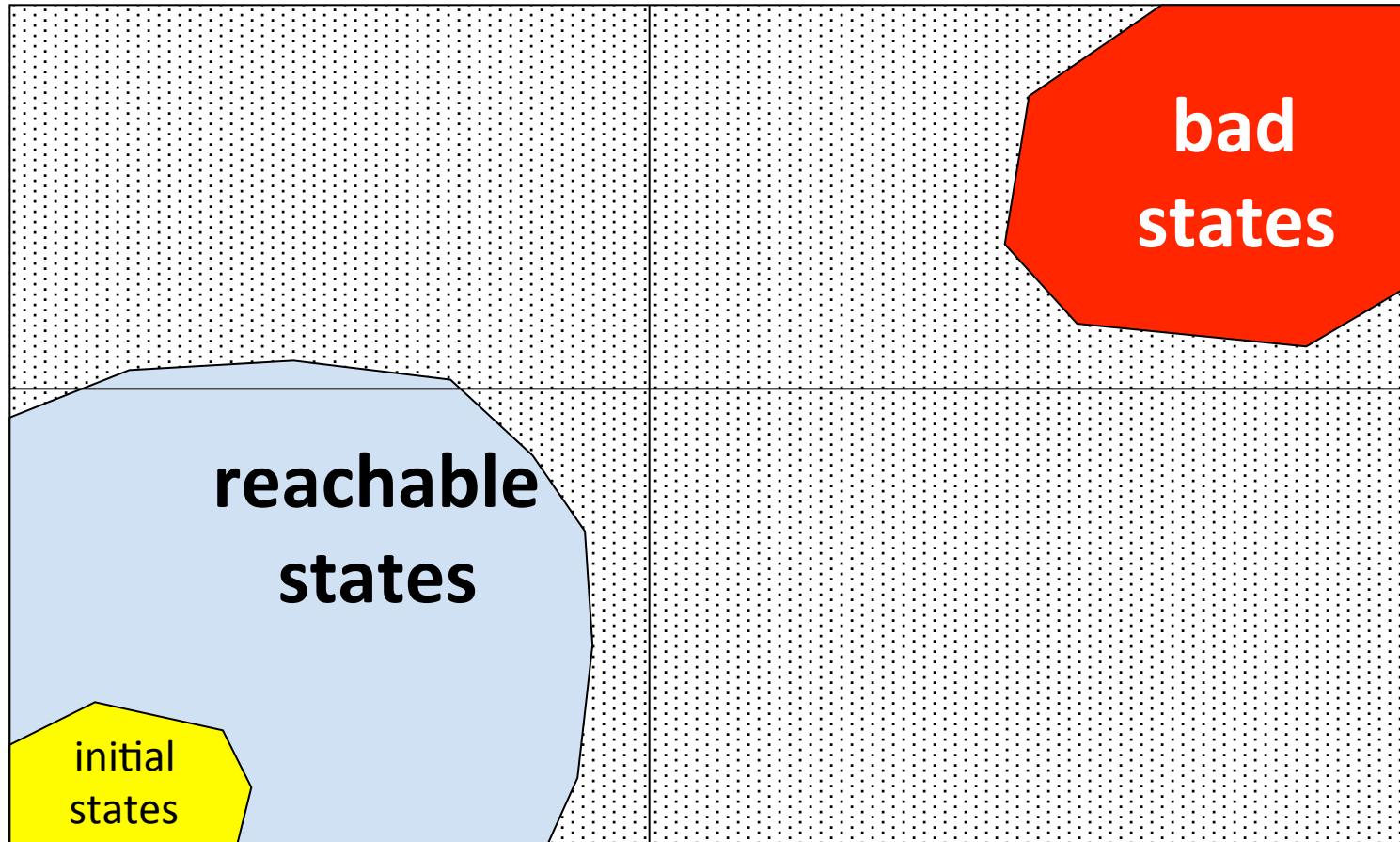
“Abstract-state transformer” semantics

<code>y = ?;</code>	<code>y=T, x=T</code>
<code>x = y * 2</code>	<code>y=T, x=E</code>
<code>if (x % 2 == 0) {</code>	<code>y=T, x=E</code>
<code>y = 42;</code>	<code>y=E, x=E</code>
<code>}</code>	
<code>else {</code>	
<code>y = 73;</code>	<code>...</code>
<code>foo();</code>	<code>...</code>
<code>}</code>	
<code>assert (y%2 == 0)</code>	<code>y=E, x=E</code>

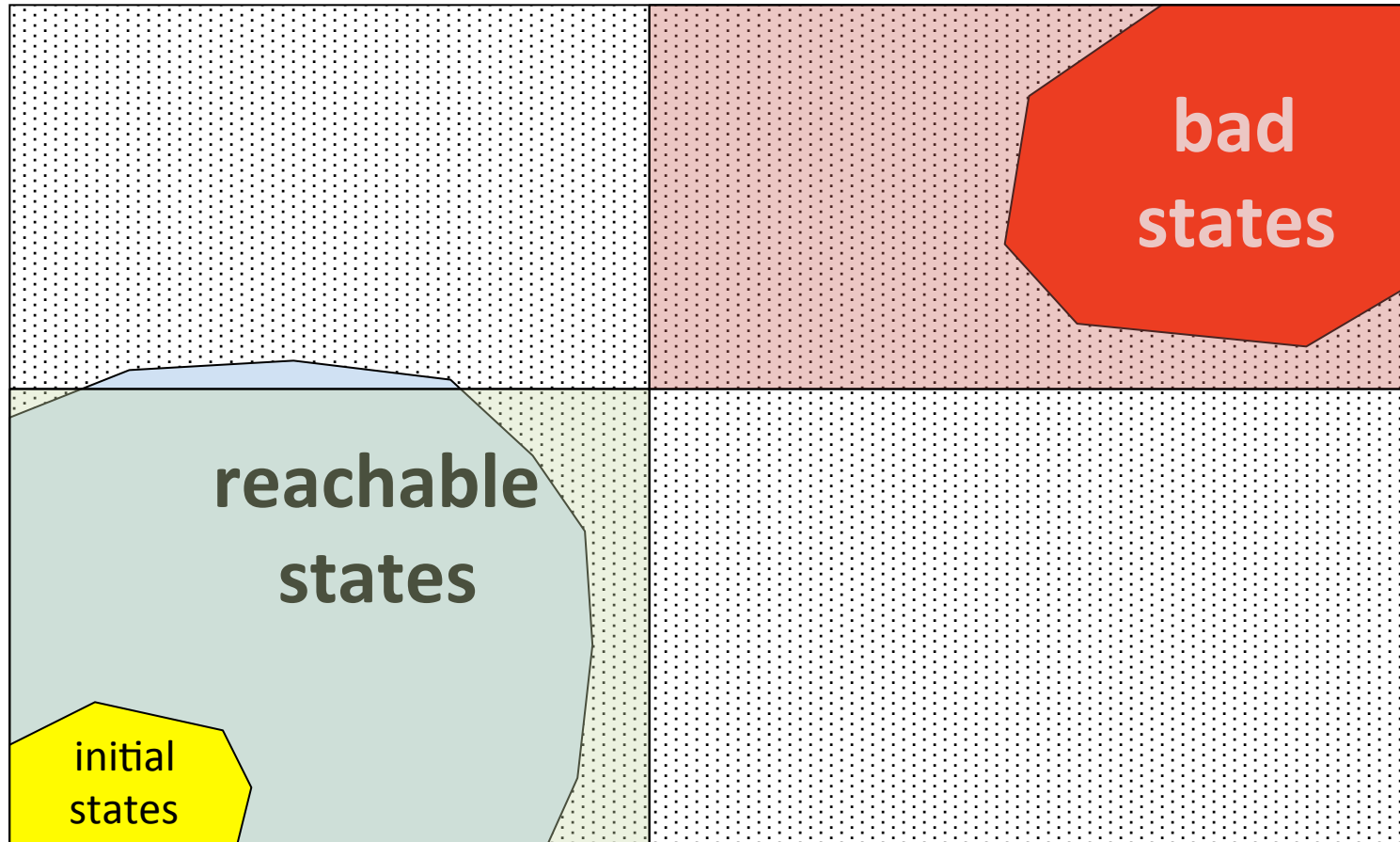


?

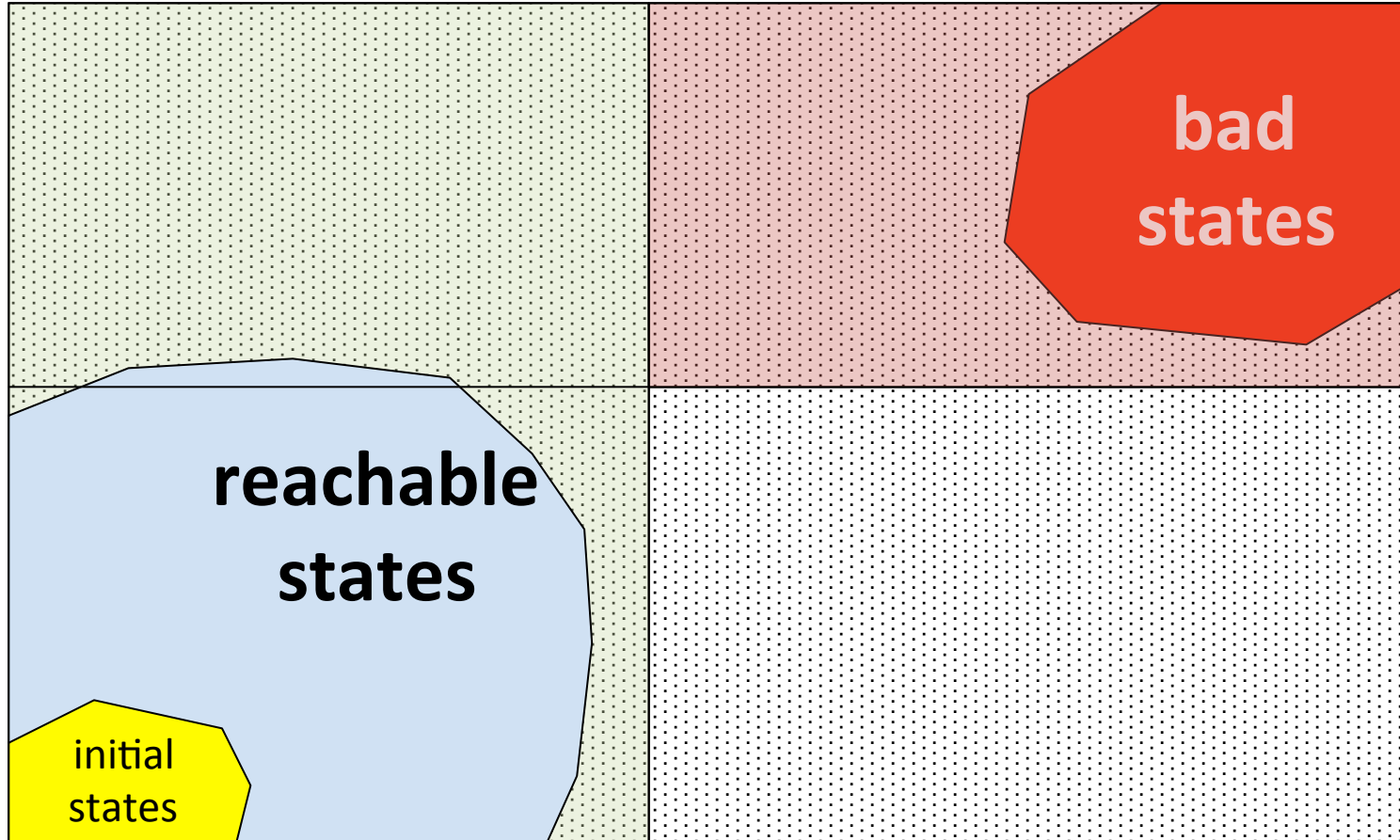
“Abstract-state transformer” semantics



“Abstract-state transformer” semantics



“Abstract-state transformer” semantics



How do we *say* what P *mean*?

```
y = ?; x = ?;  
x = y * 2  
if (x % 2 == 0) {  
    y = 42;  
} else {  
    y = 73;  
    foo();  
}  
assert (y == 42);
```

= ...

syntax

semantics

Agenda

- Operational semantics
 - Natural operational semantics
 - Structural operational semantics

Programming Languages

- Syntax
 - “how do I write a program?”
 - BNF
 - “Parsing”
- Semantics
 - “What does my program mean?”
 - ...

Program semantics

- State-transformer
 - Set-of-states transformer
 - Trace transformer
- Predicate-transformer
- Functions

Program semantics

- State-transformer
 - Set-of-states transformer
 - Trace transformer
- Predicate-transformer
- Functions

What semantics do we want?

- Captures the aspects of computations we care about
 - “adequate”
- Hides irrelevant details
 - “fully abstract”
- Compositional

What semantics do we want?

- Captures the aspects of computations we care about
 - “adequate”
- Hides irrelevant details
 - “fully abstract”
- Compositional

Formal semantics

“Formal semantics is concerned with rigorously specifying the meaning, or behavior, of programs, pieces of hardware, etc.”

[Semantics with Applications – a Formal Introduction](#) (Page 1)

Nielsen & Nielsen

Formal semantics

“This theory allows a program to be manipulated like a formula – that is to say, its properties can be calculated.”

G rard Huet & Philippe Flajolet homage to Gilles Kahn

Why formal semantics?

- Implementation-independent definition of a programming language
- Automatically generating interpreters
 - and some day maybe full fledged compilers
- Verification and debugging
 - if you don't know what it does, how do you know its incorrect?

Why formal semantics?

- Implementation-independent definition of a programming language
- Automatically generating interpreters
 - and some day maybe full fledged compilers
- Verification and debugging
 - if you don't know what it does, how do you know its incorrect?

Levels of abstractions and applications

Static Analysis
(abstract semantics)

\sqsubseteq

Program Semantics

\sqsubseteq

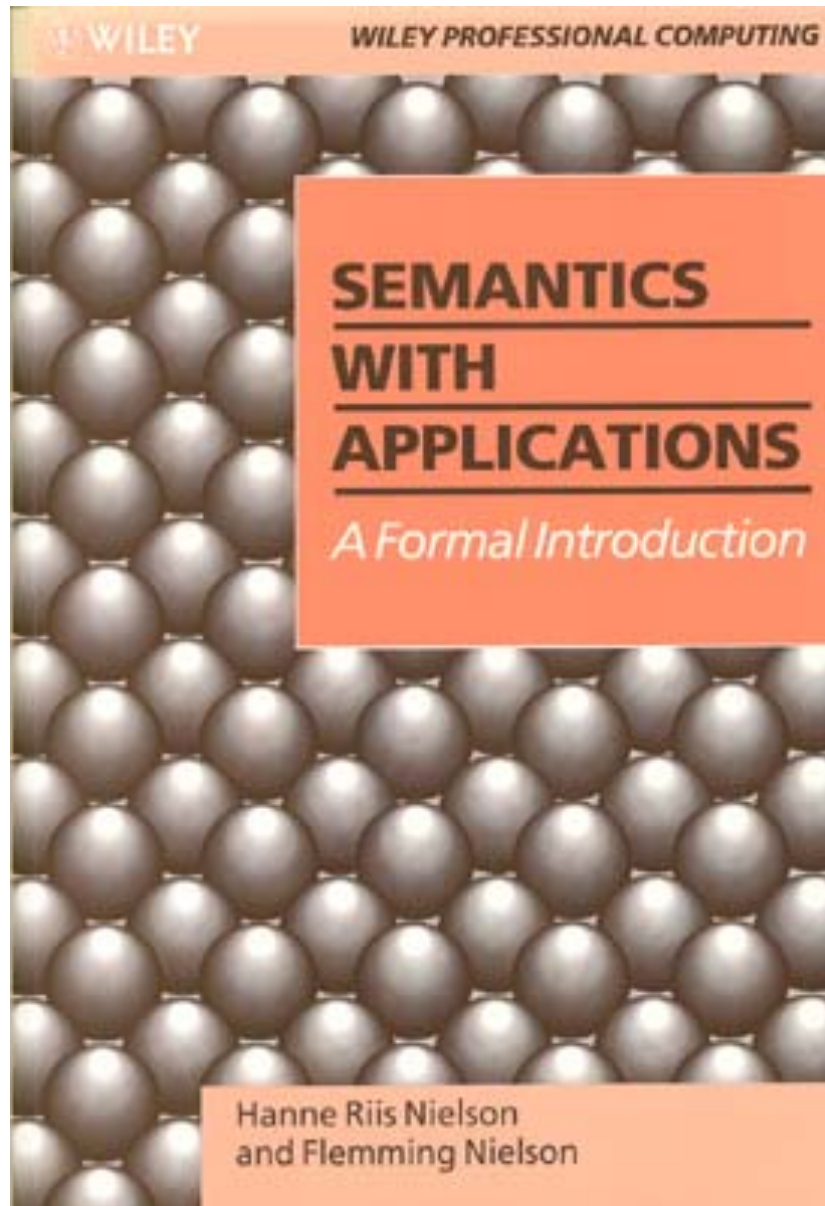
Assembly-level Semantics
(Small-step)

Semantic description methods

- Operational semantics
 - Natural semantics (big step) [G. Kahn]
 - Structural semantics (small step) [G. Plotkin]
 - *Trace semantics*
 - *Collecting semantics*
 - *[Instrumented semantics]*
- Denotational semantics [D. Scott, C. Strachy]
- Axiomatic semantics [C. A. R. Hoare, R. Floyd]

Operational Semantics

http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html



A simple imperative language: **While**

Abstract syntax:

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

$b ::= \mathbf{true} \mid \mathbf{false}$

$\mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$

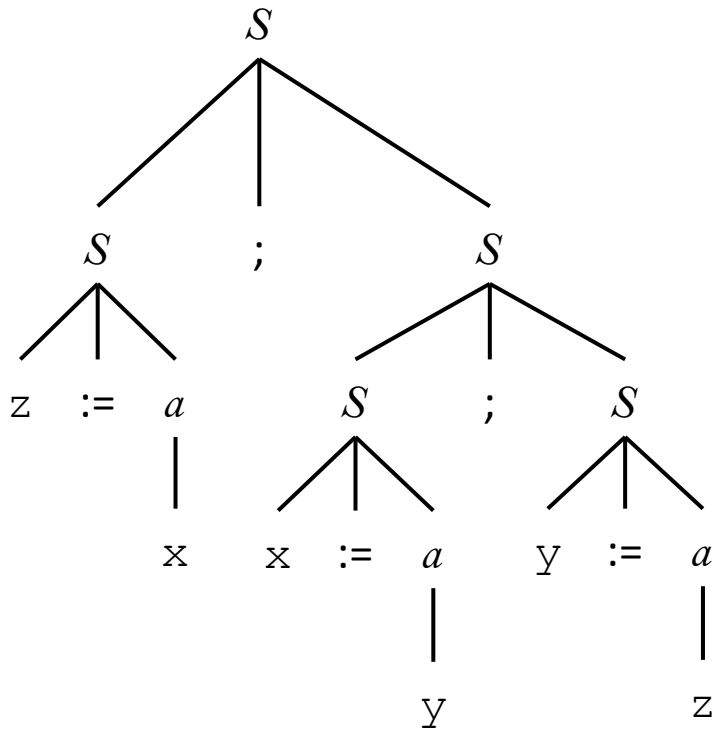
$S ::= x := a \mid \mathbf{skip} \mid S_1; S_2$

$\mid \mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2$

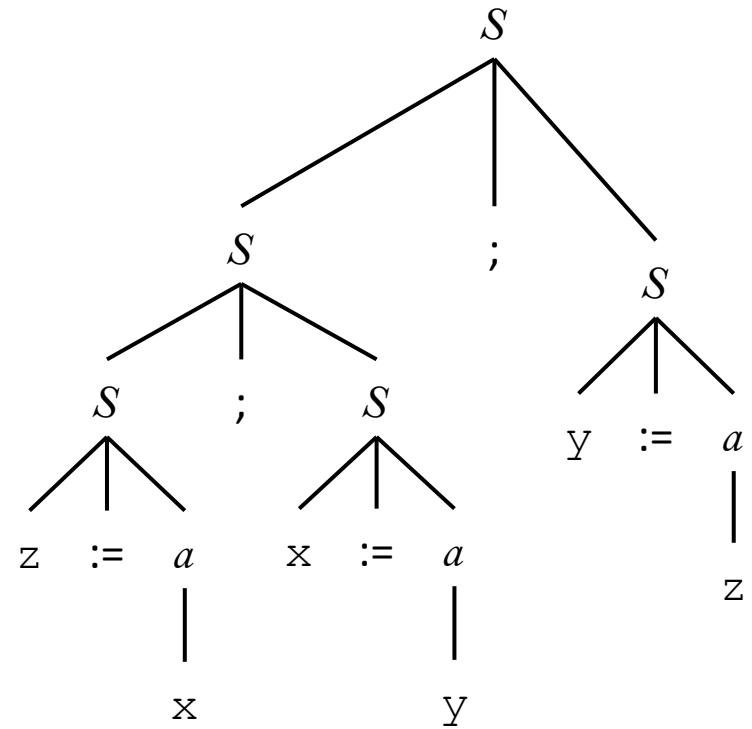
$\mid \mathbf{while } b \mathbf{ do } S$

Concrete Syntax vs. Abstract Syntax

$z := x; x := y; y := z$



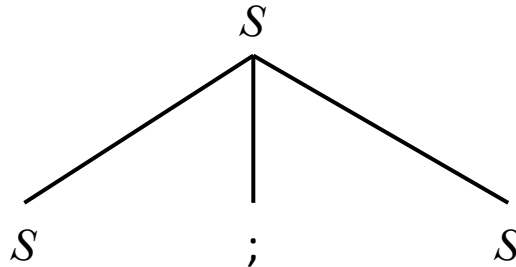
$z := x; (x := y; y := z)$



$(z := x; x := y); y := z$

Exercise: draw an AST

`y:=1; while $\neg(x=1)$ do (y:=y*x; x:=x-1)`



Syntactic categories

$n \in \mathbf{Num}$	numerals
$x \in \mathbf{Var}$	program variables
$a \in \mathbf{Aexp}$	arithmetic expressions
$b \in \mathbf{Bexp}$	boolean expressions
$S \in \mathbf{Stm}$	statements

Semantic categories

Z Integers $\{0, 1, -1, 2, -2, \dots\}$

T Truth values $\{\text{ff}, \text{tt}\}$

State **Var** \rightarrow **Z**

Example state: $s = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$

Lookup: $s \ x = 5$

Update: $s[x \mapsto 6] = [x \mapsto 6, y \mapsto 7, z \mapsto 0]$

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

- Arithmetic expressions are side-effect free
- Semantic function $\mathcal{A} \llbracket \mathbf{Aexp} \rrbracket : \mathbf{State} \rightarrow \mathbf{Z}$
- Defined by induction on the syntax tree

$$\mathcal{A} \llbracket n \rrbracket s = n$$

$$\mathcal{A} \llbracket x \rrbracket s = s.x$$

$$\mathcal{A} \llbracket a_1 + a_2 \rrbracket s = \mathcal{A} \llbracket a_1 \rrbracket s + \mathcal{A} \llbracket a_2 \rrbracket s$$

$$\mathcal{A} \llbracket a_1 - a_2 \rrbracket s = \mathcal{A} \llbracket a_1 \rrbracket s - \mathcal{A} \llbracket a_2 \rrbracket s$$

$$\mathcal{A} \llbracket a_1 * a_2 \rrbracket s = \mathcal{A} \llbracket a_1 \rrbracket s \times \mathcal{A} \llbracket a_2 \rrbracket s$$

$$\mathcal{A} \llbracket (a_1) \rrbracket s = \mathcal{A} \llbracket a_1 \rrbracket s \text{ --- not needed}$$

$$\mathcal{A} \llbracket -a \rrbracket s = 0 - \mathcal{A} \llbracket a_1 \rrbracket s$$

- Compositional
- Properties can be proved by structural induction

Arithmetic expression exercise

Suppose $s \ x = 3$

Evaluate $\mathcal{A} [\mathbf{x+1}] s$

Semantics of boolean expressions

- Boolean expressions are side-effect free
- Semantic function $\mathcal{B} \llbracket \mathbf{Bexp} \rrbracket : \mathbf{State} \rightarrow \mathbf{T}$
- Defined by induction on the syntax tree

$$\mathcal{B} \llbracket \text{true} \rrbracket s = \text{tt}$$

$$\mathcal{B} \llbracket \text{false} \rrbracket s = \text{ff}$$

$$\mathcal{B} \llbracket a_1 = a_2 \rrbracket s =$$

$$\mathcal{B} \llbracket a_1 \leq a_2 \rrbracket s =$$

$$\mathcal{B} \llbracket b_1 \wedge b_2 \rrbracket s =$$

$$\mathcal{B} \llbracket \neg b \rrbracket s =$$

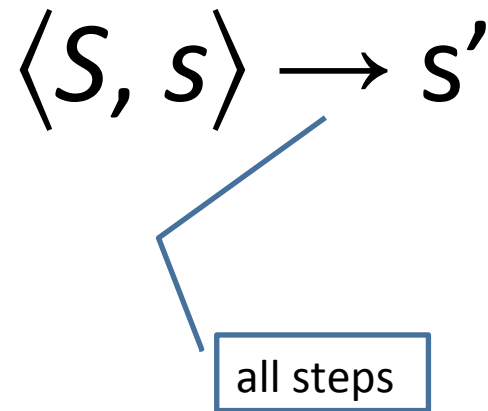
Operational semantics

- Concerned with **how** to execute programs
 - How statements modify **state**
 - Define transition relation between configurations
- Two flavors
 - **Natural semantics**: describes how the **overall** results of executions are obtained
 - So-called “big-step” semantics
 - **Structural operational semantics**: describes how the **individual steps** of a computations take place
 - So-called “small-step” semantics

Natural operating semantics (NS)

Natural operating semantics (NS)

- aka “Large-step semantics”

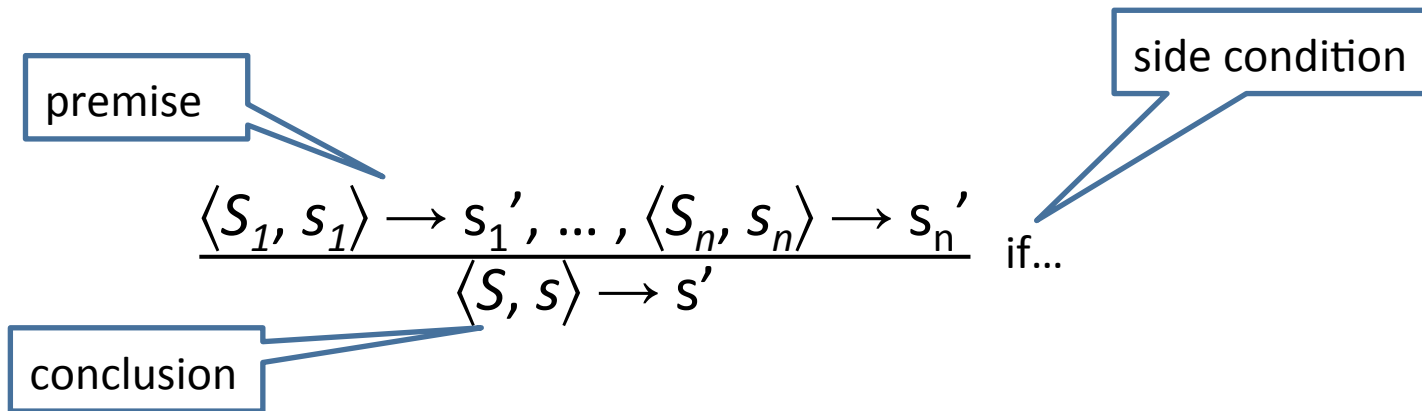


Natural operating semantics

- Developed by Gilles Kahn [[STACS 1987](#)]
- Configurations
 - $\langle S, s \rangle$ Statement S is about to execute on state s
 - s Terminal (final) state
- Transitions
 - $\langle S, s \rangle \rightarrow s'$ Execution of S from s will terminate with the result state s'
 - Ignores non-terminating computations

Natural operating semantics

- \rightarrow defined by rules of the form



- The meaning of compound statements is defined using the meaning immediate constituent statements

Natural semantics for **While**

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

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

axioms

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

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

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

Natural semantics for **While**

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

Non-compositional

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

Example

- Let s_0 be the state which assigns zero to all program variables

$$\langle x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]$$

$$\langle \text{skip}, s_0 \rangle \rightarrow s_0$$

$$\frac{\langle \text{skip}, s_0 \rangle \rightarrow s_0, \langle x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]}{\langle \text{skip}; x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]}$$

$$\frac{\langle x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]}{\langle \text{if } x=0 \text{ then } x := x+1 \text{ else skip}, s_0 \rangle \rightarrow s_0[x \mapsto 1]}$$

Derivation trees

- Using axioms and rules to derive a transition $\langle S, s \rangle \rightarrow s'$ gives a derivation tree
 - Root: $\langle S, s \rangle \rightarrow s'$
 - Leaves: axioms
 - Internal nodes: conclusions of rules
 - Immediate children: matching rule premises

Derivation tree example 1

- Assume $s_0 = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$
 $s_1 = [x \mapsto 5, y \mapsto 7, z \mapsto 5]$
 $s_2 = [x \mapsto 7, y \mapsto 7, z \mapsto 5]$
 $s_3 = [x \mapsto 7, y \mapsto 5, z \mapsto 5]$

[ass_{ns}]

$$\langle z := x, s_0 \rangle \rightarrow s_1$$

[ass_{ns}]

$$\langle x := y, s_1 \rangle \rightarrow s_2$$

[comp_{ns}]

$$\langle (z := x; x := y), s_0 \rangle \rightarrow s_2$$

[ass_{ns}]

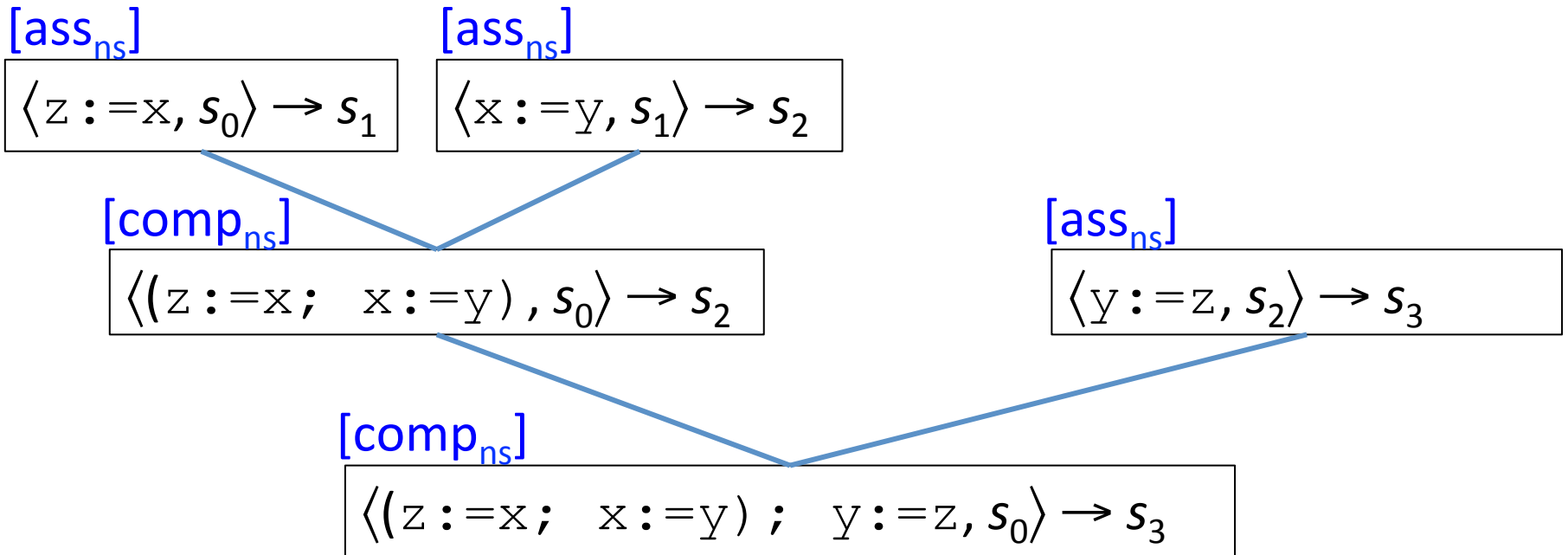
$$\langle y := z, s_2 \rangle \rightarrow s_3$$

[comp_{ns}]

$$\langle (z := x; x := y); y := z, s_0 \rangle \rightarrow s_3$$

Derivation tree example 1

- Assume $s_0 = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$
 $s_1 = [x \mapsto 5, y \mapsto 7, z \mapsto 5]$
 $s_2 = [x \mapsto 7, y \mapsto 7, z \mapsto 5]$
 $s_3 = [x \mapsto 7, y \mapsto 5, z \mapsto 5]$

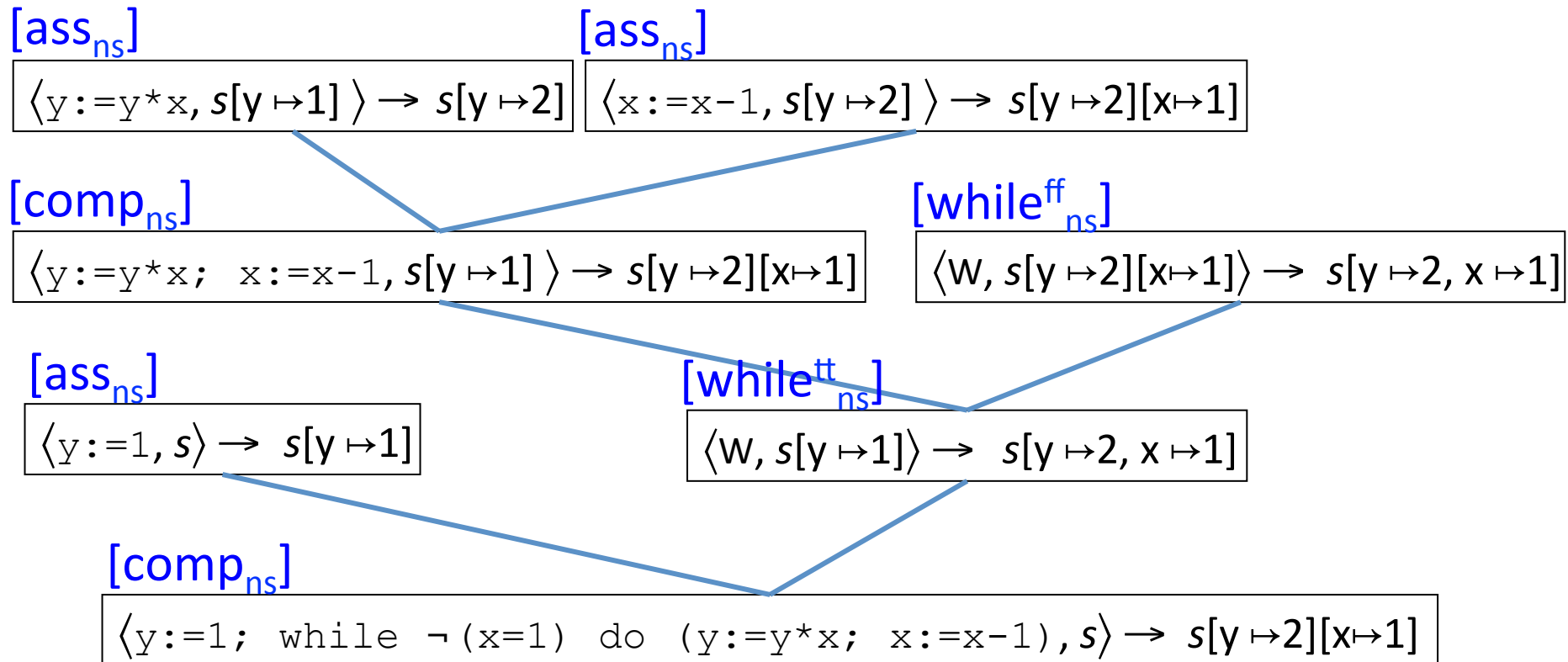


Top-down evaluation via derivation trees

- Given a statement S and an input state s
find an output state s' such that $\langle S, s \rangle \rightarrow s'$
- Start with the root and repeatedly apply rules
until the axioms are reached
 - Inspect different alternatives in order
- In While s' and the derivation tree is unique

Top-down evaluation example

- Factorial program with $s \ x = 2$
- Shorthand: $w = \text{while } \neg(x=1) \text{ do } (y := y * x; x := x - 1)$



Program termination

- Given a statement S and input s
 - S **terminates** on s if there exists a state s' such that $\langle S, s \rangle \rightarrow s'$
 - S **loops** on s if there is no state s' such that $\langle S, s \rangle \rightarrow s'$
- Given a statement S
 - S **always terminates** if
for every input state s , S terminates on s
 - S **always loops** if
for every input state s , S loops on s

Semantic equivalence

- S_1 and S_2 are **semantically equivalent** if for all s and s'
 $\langle S_1, s \rangle \rightarrow s'$ if and only if $\langle S_2, s \rangle \rightarrow s'$
- Simple example
`while b do S`
is semantically equivalent to:
`if b then (S ; while b do S) else skip`
– Read proof in pages 26-27

Properties of natural semantics

- Equivalence of program constructs
 - **skip**; **skip** is semantically equivalent to **skip**
 - $((S_1; S_2); S_3)$ is semantically equivalent to $(S_1; (S_2; S_3))$
 - $(x := 5; y := x * 8)$ is semantically equivalent to $(x := 5; y := 40)$

Equivalence of $(S_1; S_2); S_3$ and $S_1; (S_2; S_3)$

Equivalence of $(S_1; S_2); S_3$ and $S_1; (S_2; S_3)$

Assume $\langle (S_1; S_2); S_3, s \rangle \rightarrow s'$ then the following unique derivation tree exists:

$$\frac{\langle S_1, s \rangle \rightarrow s_1, \langle S_2, s_1 \rangle \rightarrow s_{12}}{\frac{\langle (S_1; S_2), s \rangle \rightarrow s_{12}, \langle S_3, s_{12} \rangle \rightarrow s'}{\langle (S_1; S_2); S_3, s \rangle \rightarrow s'}}$$

Using the rule applications above, we can construct the following derivation tree:

$$\frac{\langle S_1, s \rangle \rightarrow s_1, \frac{\langle S_2, s_1 \rangle \rightarrow s_{12}, \langle S_3, s_{12} \rangle \rightarrow s'}{\langle (S_2; S_3), s_{12} \rangle \rightarrow s'}}{\langle (S_1; S_2); S_3, s \rangle \rightarrow s'}$$

And vice versa.

Deterministic semantics for **While**

- **Theorem:** for all statements S and states s_1, s_2
if $\langle S, s \rangle \rightarrow s_1$ and $\langle S, s \rangle \rightarrow s_2$ then $s_1 = s_2$
- The proof uses induction on the shape of derivation trees (pages 29-30)
 - Prove that the property holds for all simple derivation trees by showing it holds for axioms
 - Prove that the property holds for all composite trees:
 - For each rule assume that the property holds for its premises (induction hypothesis) and prove it holds for the conclusion of the rule

single
node

#nodes > 1

The semantic function S_{ns}

- The meaning of a statement S is defined as a partial function from **State** to **State**

$$S_{ns}: \mathbf{Stm} \rightarrow (\mathbf{State} \hookrightarrow \mathbf{State})$$

$$S_{ns} \llbracket S \rrbracket s = \begin{cases} s' & \text{if } \langle S, s \rangle \rightarrow s' \\ \text{undefined} & \text{otherwise} \end{cases}$$

- Examples:

$$S_{ns} \llbracket \text{skip} \rrbracket s = s$$

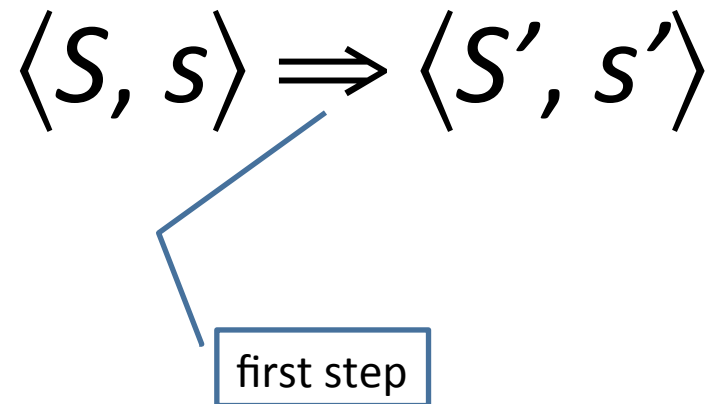
$$S_{ns} \llbracket x := 1 \rrbracket s = s [x \mapsto 1]$$

$$S_{ns} \llbracket \text{while true do skip} \rrbracket s = \text{undefined}$$

Structural operating semantics (SOS)

Structural operating semantics (SOS)

- aka “Small-step semantics”



Structural operational semantics

- Developed by Gordon Plotkin
- Configurations: γ has one of two forms:
 - $\langle S, s \rangle$ Statement S is about to execute on state s
 - s Terminal (final) state
- Transitions $\langle S, s \rangle \Rightarrow \gamma$
 - $\gamma = \langle S', s' \rangle$ Execution of S from s is **not** completed and remaining computation proceeds from intermediate configuration γ
 - $\gamma = s'$ Execution of S from s has **terminated** and the final state is s'
- $\langle S, s \rangle$ is **stuck** if there is no γ such that $\langle S, s \rangle \Rightarrow \gamma$



first step

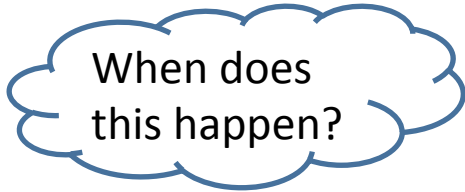
Structural semantics for **While**

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

$$[\text{skip}_{\text{sos}}] \quad \langle \text{skip}, s \rangle \Rightarrow s$$

$$[\text{comp}^1_{\text{sos}}] \quad \frac{\langle S_1, s \rangle \Rightarrow \langle S_1', s' \rangle}{\langle S_1; S_2, s \rangle \Rightarrow \langle S_1'; S_2, s' \rangle}$$

$$[\text{comp}^2_{\text{sos}}] \quad \frac{\langle S_1, s \rangle \Rightarrow s' \circ \circ \circ}{\langle S_1; S_2, s \rangle \Rightarrow \langle S_2, s' \rangle}$$



$$[\text{if}^{\text{tt}}_{\text{sos}}] \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_1, s \rangle \quad \text{if } \mathcal{B}[[b]]s = \mathbf{tt}$$

$$[\text{if}^{\text{ff}}_{\text{sos}}] \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_2, s \rangle \quad \text{if } \mathcal{B}[[b]]s = \mathbf{ff}$$

Structural semantics for **While**

[while_{sos}]

$$\langle \text{while } b \text{ do } S, s \rangle \Rightarrow$$
$$\langle \text{if } b \text{ then}$$
$$\quad S; \text{while } b \text{ do } S \rangle$$
$$\text{else}$$
$$\quad \text{skip}, s \rangle$$

Derivation sequences

- A derivation sequence of a statement S starting in state s is either
- A **finite** sequence $\gamma_0, \gamma_1, \gamma_2 \dots, \gamma_k$ such that
 1. $\gamma_0 = \langle S, s \rangle$
 2. $\gamma_i \Rightarrow \gamma_{i+1}$
 3. γ_k is either stuck configuration or a final state
- An **infinite** sequence $\gamma_0, \gamma_1, \gamma_2, \dots$ such that
 1. $\gamma_0 = \langle S, s \rangle$
 2. $\gamma_i \Rightarrow \gamma_{i+1}$
- Notations:
 - $\gamma_0 \Rightarrow^k \gamma_k$ γ_0 derives γ_k in k steps
 - $\gamma_0 \Rightarrow^* \gamma$ γ_0 derives γ in a finite number of steps
- For **each** step there is a corresponding derivation tree

Derivation sequence example

- Assume $s_0 = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$
 - $\langle (z := x; x := y); y := z, s_0 \rangle$
 - $\Rightarrow \langle x := y; y := z, s_0[z \mapsto 5] \rangle$
 - $\Rightarrow \langle y := z, (s_0[z \mapsto 5])[x \mapsto 7] \rangle$
 - $\Rightarrow ((s_0[z \mapsto 5])[x \mapsto 7])[y \mapsto 5]$

- Derivation tree for first step:

$$\frac{\langle z := x, s_0 \rangle \Rightarrow s_0[z \mapsto 5]}{\langle z := x; x := y, s_0 \rangle \Rightarrow \langle x := y, s_0[z \mapsto 5] \rangle}$$

$$\langle (z := x; x := y); y := z, s_0 \rangle \Rightarrow \langle x := y; y := z, s_0[z \mapsto 5] \rangle$$

Evaluation via derivation sequences

- For any **While** statement S and state s it is always possible to find at least one derivation sequence from $\langle S, s \rangle$
 - Apply axioms and rules forever or until a terminal or stuck configuration is reached
- **Proposition:** there are no stuck configurations in **While**

Factorial ($n!$) example

- Input state s such that $s \ x = 3$

$y := 1; \text{ while } \neg(x=1) \text{ do } (y := y * x; x := x - 1)$

$\langle y := 1; W, s \rangle$
 $\Rightarrow \langle W, s[y \mapsto 1] \rangle$
 $\Rightarrow \langle \text{if } \neg(x=1) \text{ then } ((y := y * x; x := x - 1); W \text{ else skip}), s[y \mapsto 1] \rangle$
 $\Rightarrow \langle ((y := y * x; x := x - 1); W), s[y \mapsto 1] \rangle$
 $\Rightarrow \langle (x := x - 1; W), s[y \mapsto 3] \rangle$
 $\Rightarrow \langle W, s[y \mapsto 3][x \mapsto 2] \rangle$
 $\Rightarrow \langle \text{if } \neg(x=1) \text{ then } ((y := y * x; x := x - 1); W \text{ else skip}), s[y \mapsto 3][x \mapsto 2] \rangle$
 $\Rightarrow \langle ((y := y * x; x := x - 1); W), s[y \mapsto 3][x \mapsto 2] \rangle$
 $\Rightarrow \langle (x := x - 1; W), s[y \mapsto 6][x \mapsto 2] \rangle$
 $\Rightarrow \langle W, s[y \mapsto 6][x \mapsto 1] \rangle$
 $\Rightarrow \langle \text{if } \neg(x=1) \text{ then } ((y := y * x; x := x - 1); W \text{ else skip}), s[y \mapsto 6][x \mapsto 1] \rangle$
 $\Rightarrow \langle \text{skip}, s[y \mapsto 6][x \mapsto 1] \rangle$
 $\Rightarrow s[y \mapsto 6][x \mapsto 1]$

Program termination

- Given a statement S and input s
 - S **terminates** on s if there exists a finite derivation sequence starting at $\langle S, s \rangle$
 - S **terminates successfully** on s if there exists a finite derivation sequence starting at $\langle S, s \rangle$ leading to a final state
 - S **loops** on s if there exists an infinite derivation sequence starting at $\langle S, s \rangle$

Properties of structural operational semantics

- S_1 and S_2 are **semantically equivalent** if:
 - for all s and γ which is either final or stuck,
 $\langle S_1, s \rangle \Rightarrow^* \gamma$ if and only if $\langle S_2, s \rangle \Rightarrow^* \gamma$
 - for all s , there is an infinite derivation sequence starting at $\langle S_1, s \rangle$ if and only if there is an infinite derivation sequence starting at $\langle S_2, s \rangle$
- **Theorem: While** is deterministic:
 - If $\langle S, s \rangle \Rightarrow^* s_1$ and $\langle S, s \rangle \Rightarrow^* s_2$ then $s_1 = s_2$

Sequential composition

- **Lemma:** If $\langle S_1; S_2, s \rangle \Rightarrow^k s''$ then there exists s' and $k=m+n$ such that $\langle S_1, s \rangle \Rightarrow^m s'$ and $\langle S_2, s' \rangle \Rightarrow^n s''$
- The proof (pages 37-38) uses induction on the length of derivation sequences
 - Prove that the property holds for all derivation sequences of length 0
 - Prove that the property holds for all other derivation sequences:
 - Show that the property holds for sequences of length $k+1$ using the fact it holds on all sequences of length k (induction hypothesis)

The semantic function S_{sos}

- The meaning of a statement S is defined as a partial function from **State** to **State**

$$S_{\text{sos}}: \mathbf{Stm} \rightarrow (\mathbf{State} \hookrightarrow \mathbf{State})$$

$$S_{\text{sos}} \llbracket S \rrbracket s = \begin{cases} s' & \text{if } \langle S, s \rangle \Rightarrow^* s' \\ \text{undefined} & \text{else} \end{cases}$$

- Examples:

$$S_{\text{sos}} \llbracket \text{skip} \rrbracket s = s$$

$$S_{\text{sos}} \llbracket x := 1 \rrbracket s = s [x \mapsto 1]$$

$$S_{\text{sos}} \llbracket \text{while true do skip} \rrbracket s = \text{undefined}$$

An equivalence result

- For every statement in **While**

$$S_{\text{ns}} \llbracket S \rrbracket = S_{\text{sos}} \llbracket S \rrbracket$$

- Proof in pages 40-43

Language Extensions

- `abort` statement (like C's `exit` w/o return value)
- Non-determinism
- Parallelism
- Local Variables
- Procedures
 - Static Scope
 - Dynamic scope

While + abort

- Abstract syntax

$$\begin{aligned} S ::= & x := a \mid \mathbf{skip} \mid S_1; S_2 \\ & \mid \mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2 \\ & \mid \mathbf{while } b \mathbf{ do } S \\ & \mid \mathbf{abort} \end{aligned}$$

- Abort terminates the execution
 - In “**skip**; S ” the statement S executes
 - In “**abort**; S ” the statement S should never execute
- Natural semantics rules: ...?
- Structural semantics rules: ...?

Comparing semantics

Statement	Natural semantics	Structural semantics
<code>abort</code>		
<code>abort; S</code>		
<code>skip; S</code>		
<code>while true do skip</code>		
<code>if x = 0 then abort else y := y / x</code>		

Conclusions

- The natural semantics cannot distinguish between looping and abnormal termination
 - Unless we add a special error state
- In the structural operational semantics looping is reflected by infinite derivations and abnormal termination is reflected by stuck configuration

While + non-determinism

- Abstract syntax

$$\begin{aligned} S ::= & x := a \mid \mathbf{skip} \mid S_1; S_2 \\ & \mid \mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2 \\ & \mid \mathbf{while } b \mathbf{ do } S \\ & \mid S_1 \mathbf{ or } S_2 \end{aligned}$$

- Either S_1 is executed or S_2 is executed
- Example: $x := 1 \text{ or } (x := 2; x := x + 2)$
 - Possible outcomes for x : 1 and 4

While + non-determinism: natural semantics

$$[\text{or}_{\text{ns}}^1] \quad \frac{\langle S_1, s \rangle \rightarrow s'}{\langle S_1 \text{ or } S_2, s \rangle \rightarrow s'}$$

$$[\text{or}_{\text{ns}}^2] \quad \frac{\langle S_2, s \rangle \rightarrow s'}{\langle S_1 \text{ or } S_2, s \rangle \rightarrow s'}$$

While + non-determinism: structural semantics

$[or^1_{sos}]$

?

$[or^2_{sos}]$

?

While + non-determinism

- What about the definitions of the semantic functions?

$$- S_{ns} \llbracket S_1 \text{ or } S_2 \rrbracket s$$

$$- S_{sos} \llbracket S_1 \text{ or } S_2 \rrbracket s$$

Comparing semantics

Statement	Natural semantics	Structural semantics
<code>x:=1 or (x:=2; x:=x+2)</code>		
<code>(while true do skip) or (x:=2; x:=x+2)</code>		

Conclusions

- In the natural semantics non-determinism will suppress non-termination (looping) if possible
- In the structural operational semantics non-determinism does not suppress non-terminating statements

While + parallelism

Abstract syntax

$$\begin{aligned} S ::= & x := a \mid \mathbf{skip} \mid S_1; S_2 \\ & \mid \mathbf{if} \ b \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \\ & \mid \mathbf{while} \ b \ \mathbf{do} \ S \\ & \mid S_1 \parallel S_2 \end{aligned}$$

- All the interleaving of S_1 and S_2 are executed
- Example: $x := 1 \parallel (x := 2; x := x + 2)$
 - Possible outcomes for x : 1, 3, 4

While + parallelism: structural semantics

$$[\text{par}^1_{\text{sos}}] \quad \frac{\langle S_1, s \rangle \Rightarrow \langle S_1', s' \rangle}{\langle S_1 \parallel S_2, s \rangle \Rightarrow \langle S_1' \parallel S_2, s' \rangle}$$

$$[\text{par}^2_{\text{sos}}] \quad \frac{\langle S_1, s \rangle \Rightarrow s'}{\langle S_1 \parallel S_2, s \rangle \Rightarrow \langle S_2, s' \rangle}$$

$$[\text{par}^3_{\text{sos}}] \quad \frac{\langle S_2, s \rangle \Rightarrow \langle S_2', s' \rangle}{\langle S_1 \parallel S_2, s \rangle \Rightarrow \langle S_1 \parallel S_2', s' \rangle}$$

$$[\text{par}^4_{\text{sos}}] \quad \frac{\langle S_2, s \rangle \Rightarrow s'}{\langle S_1 \parallel S_2, s \rangle \Rightarrow \langle S_1, s' \rangle}$$

While + parallelism: natural semantics

Challenge problem:

Give a formal proof that
this is in fact impossible.

Idea: try to prove on a
restricted version of **While**
without loops/conditions

Example: derivation sequences of a parallel statement

$\langle x := 1 \parallel (x := 2; x := x + 2), s \rangle \Rightarrow$

Conclusion

- In the structural operational semantics we concentrate on small steps so interleaving of computations can be easily expressed
- In the natural semantics immediate constituent is an atomic entity so we cannot express interleaving of computations

While + memory

Abstract syntax

$$\begin{aligned} S ::= & x := a \mid \mathbf{skip} \mid S_1; S_2 \\ & \mid \mathbf{if} \ b \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \\ & \mid \mathbf{while} \ b \ \mathbf{do} \ S \\ & \mid x := \mathbf{malloc} \ (a) \\ & \mid x := [y] \\ & \mid [x] := y \end{aligned}$$

~~State : Var \rightarrow Z~~

State : Stack \times Heap

Stack : Var \rightarrow Z

Heap : Z \rightarrow Z

Integers as memory addresses

From states to traces

Trace semantics

- Low-level (conceptual) semantics
- Add program counter (pc) with states
 - $\Sigma = \mathbf{State} + \text{pc}$
- The meaning of a program is a relation
$$\tau \subseteq \Sigma \times \mathbf{Stm} \times \Sigma$$
- Execution is a finite/infinite sequence of states
- A useful concept in defining static analysis as we will see later

Example

```
1: y := 1;  
   while 2:  $\neg(x=1)$  do (  
       3: y := y * x;  
       4: x := x - 1  
   )  
5:
```

Traces

```
1: y := 1;
   while 2: ¬(x=1) do (
     3: y := y * x;
     4: x := x - 1
   )
5:
```

Set of traces is infinite therefore trace semantics is incomputable in general

$\langle \{x \mapsto 2, y \mapsto 3\}, 1 \rangle [y := 1] \langle \{x \mapsto 2, y \mapsto 1\}, 2 \rangle [\neg(x=1)] \langle \{x \mapsto 2, y \mapsto 1\}, 3 \rangle [y := y * x]$
 $\langle \{x \mapsto 2, y \mapsto 2\}, 4 \rangle [x := x - 1] \langle \{x \mapsto 1, y \mapsto 2\}, 2 \rangle [\neg(x=1)] \langle \{x \mapsto 1, y \mapsto 2\}, 5 \rangle$

$\langle \{x \mapsto 3, y \mapsto 3\}, 1 \rangle [y := 1] \langle \{x \mapsto 3, y \mapsto 1\}, 2 \rangle [\neg(x=1)] \langle \{x \mapsto 3, y \mapsto 1\}, 3 \rangle [y := y * x]$
 $\langle \{x \mapsto 3, y \mapsto 3\}, 4 \rangle [x := x - 1] \langle \{x \mapsto 2, y \mapsto 3\}, 2 \rangle [\neg(x=1)] \langle \{x \mapsto 2, y \mapsto 3\}, 3 \rangle$
 $[y := y * x] \langle \{x \mapsto 2, y \mapsto 6\}, 4 \rangle [x := x - 1] \langle \{x \mapsto 1, y \mapsto 6\}, 2 \rangle [\neg(x=1)] \langle \{x \mapsto 1, y \mapsto 6\}, 5 \rangle$

...

Operational semantics summary

- SOS is powerful enough to describe imperative programs
 - Can define the set of traces
 - Can represent program counter implicitly
 - Handle **goto** statements and other non-trivial control constructs (e.g., exceptions)
- Natural operational semantics is an abstraction
- Different semantics may be used to justify different behaviors
- Thinking in concrete semantics is essential for a analysis writer

The End