Program Analysis and Verification

0368-4479

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

Noam Rinetzky

Lecture 2: Program Semantics

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

Good manners

- Mobiles

Admin

- Grades
  - 4 Assignments (30%)
    - 1 involves programming
  - 1 Lesson summary (10%)
  - Toar I: Final exam (60%)
    - Must pass
  - Toar II: Project (60%)

✓ Scribes (this week)

? Scribes (nextweek)

Today

- What does semantics mean?
  - Why do we need it?
  - How is it related to analysis/verification?

- Operational semantics
  - Natural operational semantics
  - Structural operational semantics

Motivation: Verifying absence of bugs

SECURITY

Update your iThings NOW: Apple splats scary SSL snooping bug in iOS

OS X Mavericks still VULNERABLE, millions at risk of web hijacking

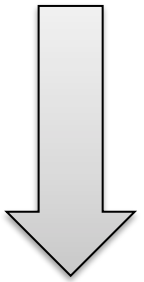
By Chris Williams, 21st February 2014 Follow 1,077 followers

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer
signedParams, uint8_t *signature, UInt16 signatureLen)
{
          OSStatus        err;
          ...

          if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                    goto fail;
          if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                    goto fail;
                    goto fail;
          if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                    goto fail;
          ...

fail:
          SSLFreeBuffer(&signedHashes);
          SSLFreeBuffer(&hashCtx);
          return err;
}
```

What can we do about it?

- Monitoring
- Testing
- Static analysis
- Formal verification
- Specification

Run time

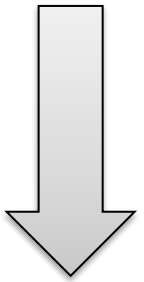


Design Time

What can we do about it?

- Monitoring
- Testing
- Static analysis
- Formal verification
- Specification

Run time



Design Time

Program analysis & verification

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

No/?/Yes

- Is assertion true?

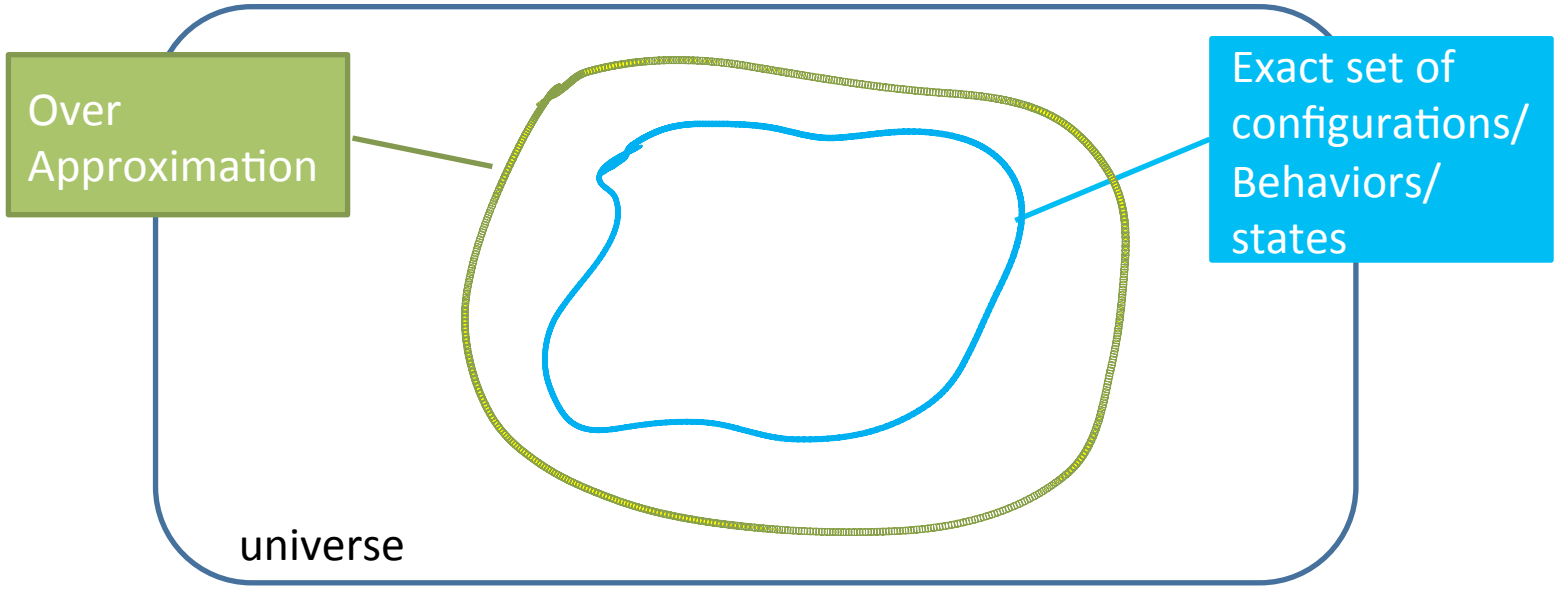
Program analysis & verification

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

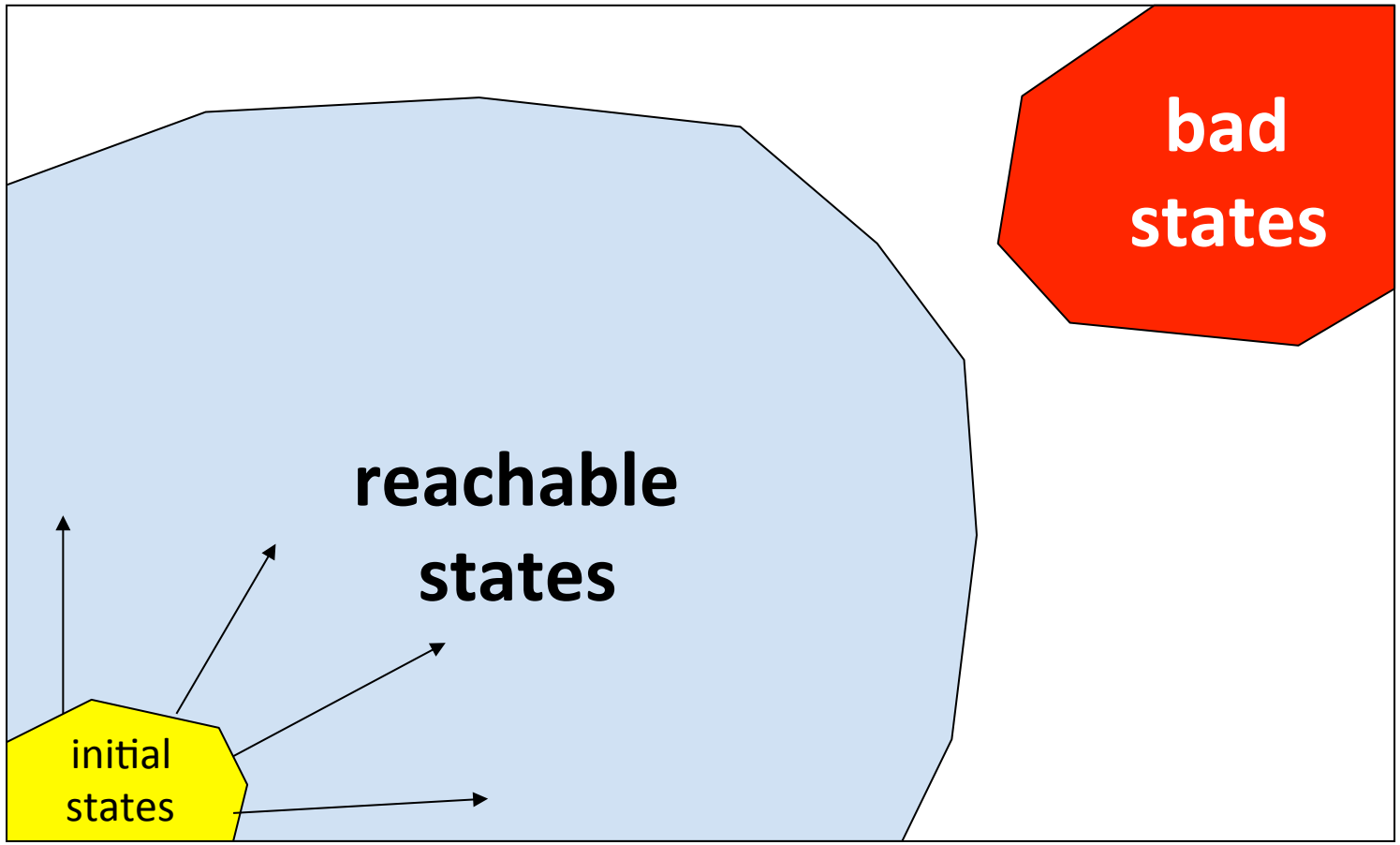
Yes/?/No

- Can we prove this? Automatically?
- Bad news: problem is generally undecidable

Main idea: use over-approximation

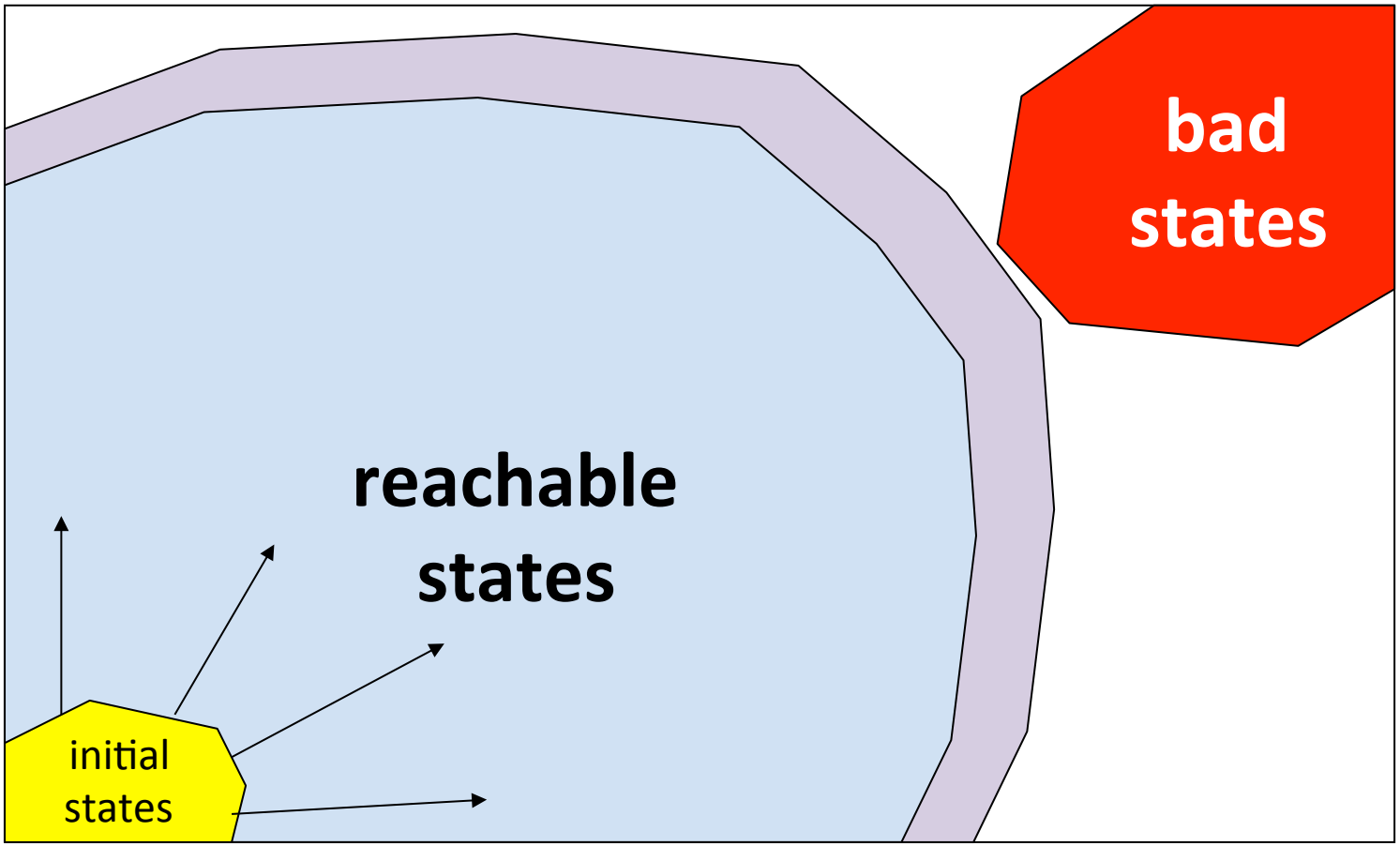


Main idea: find (properties of) all reachable states*



*or of something else …

Technique: find (properties of) more than all reachable states*



*or of something else …

Program analysis & verification

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

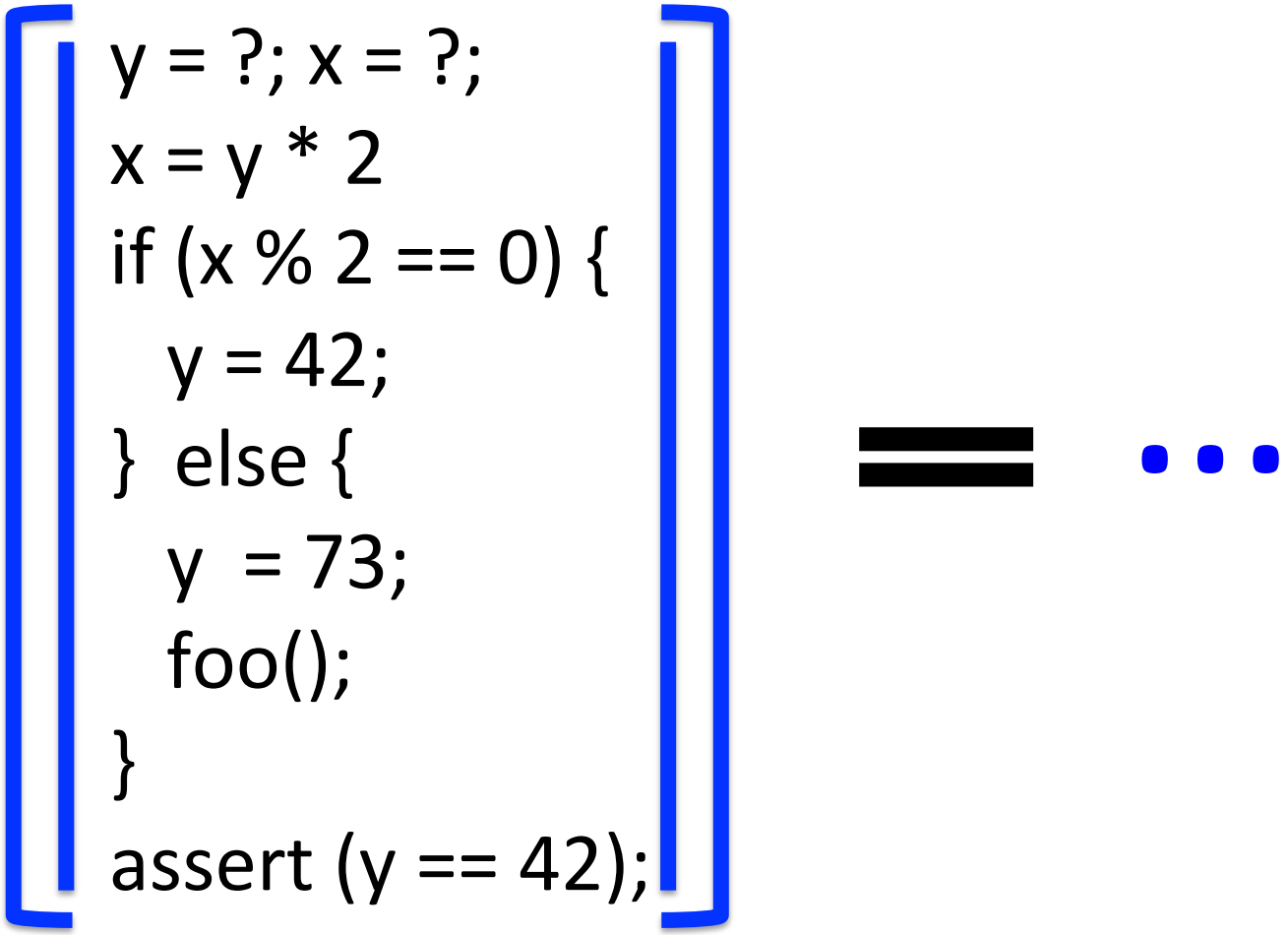
?

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?



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

y=3, x=9

…-1,0,1, … …-1,0,1,…

y **x**

“Standard” semantics

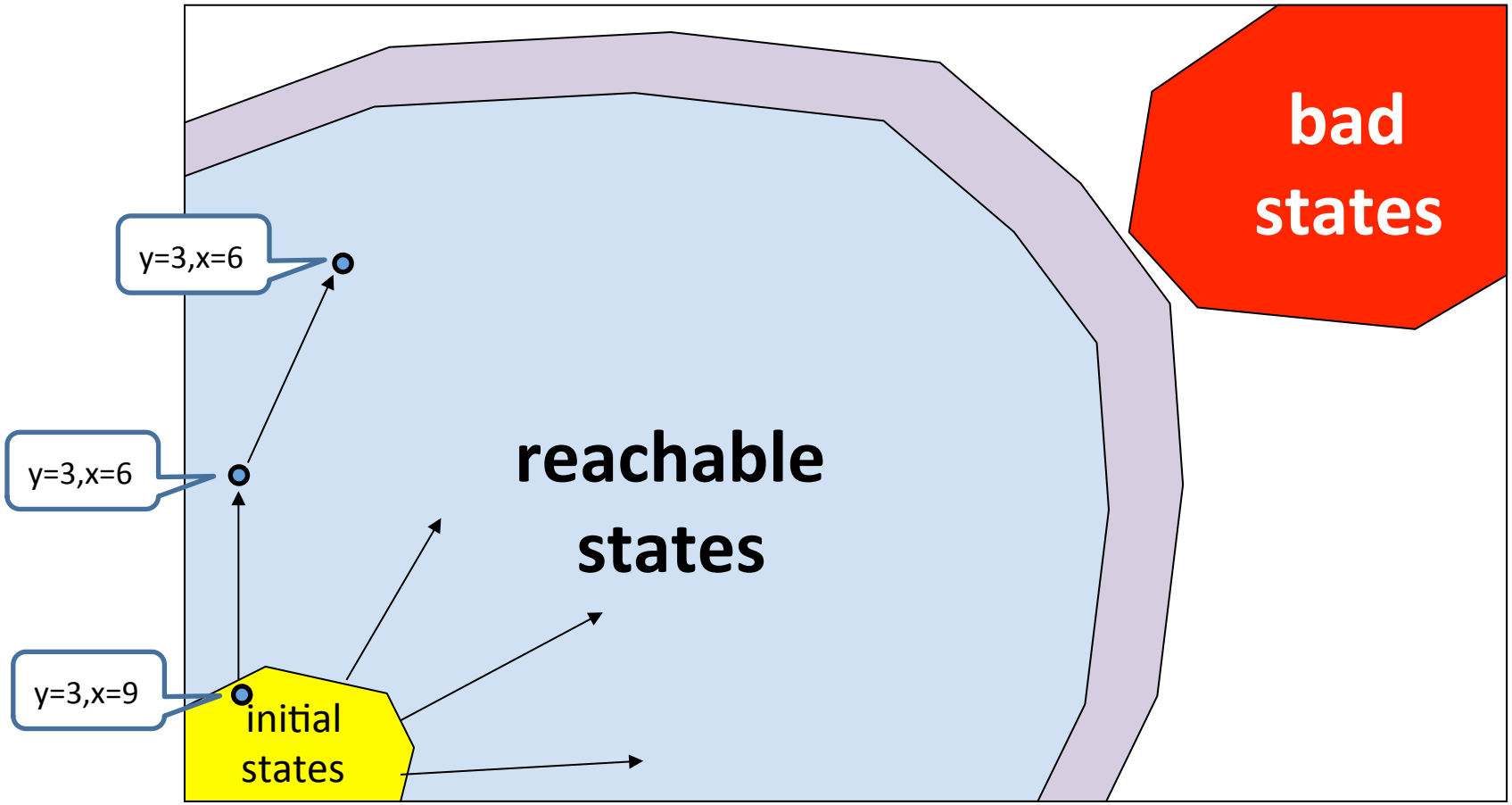
(“state transformer”)

```
y = ?;                    y=3, x=9
x = y * 2                 y=3, x=6
if (x % 2 == 0) {         y=3, x=6
   y = 42;                y=42, x=6
}  else {
   y  = 73;               ...
   foo();                 ...
}
assert (y == 42); y=42, x=6
```

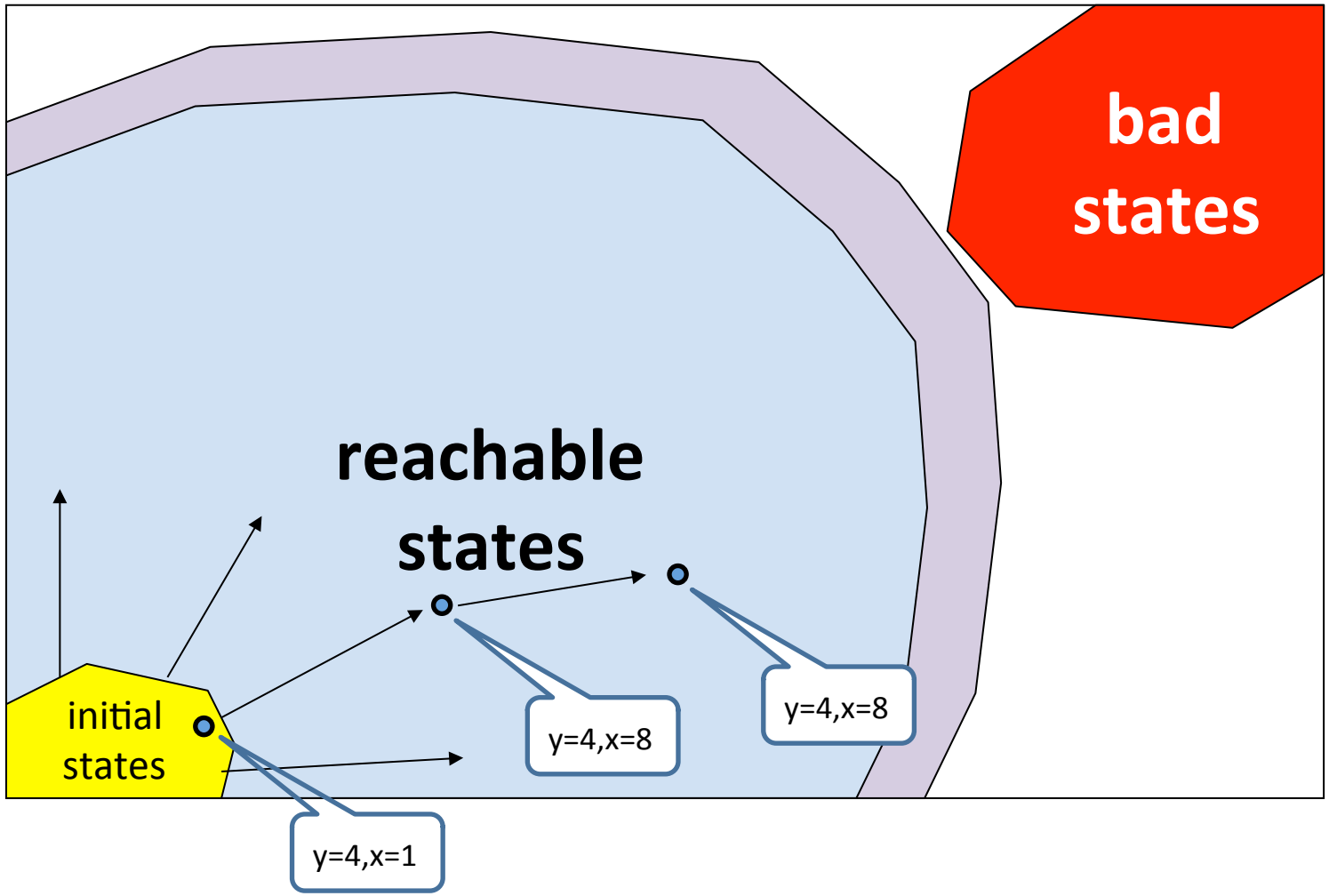
…-1,0,1, … …-1,0,1,…

y **x**

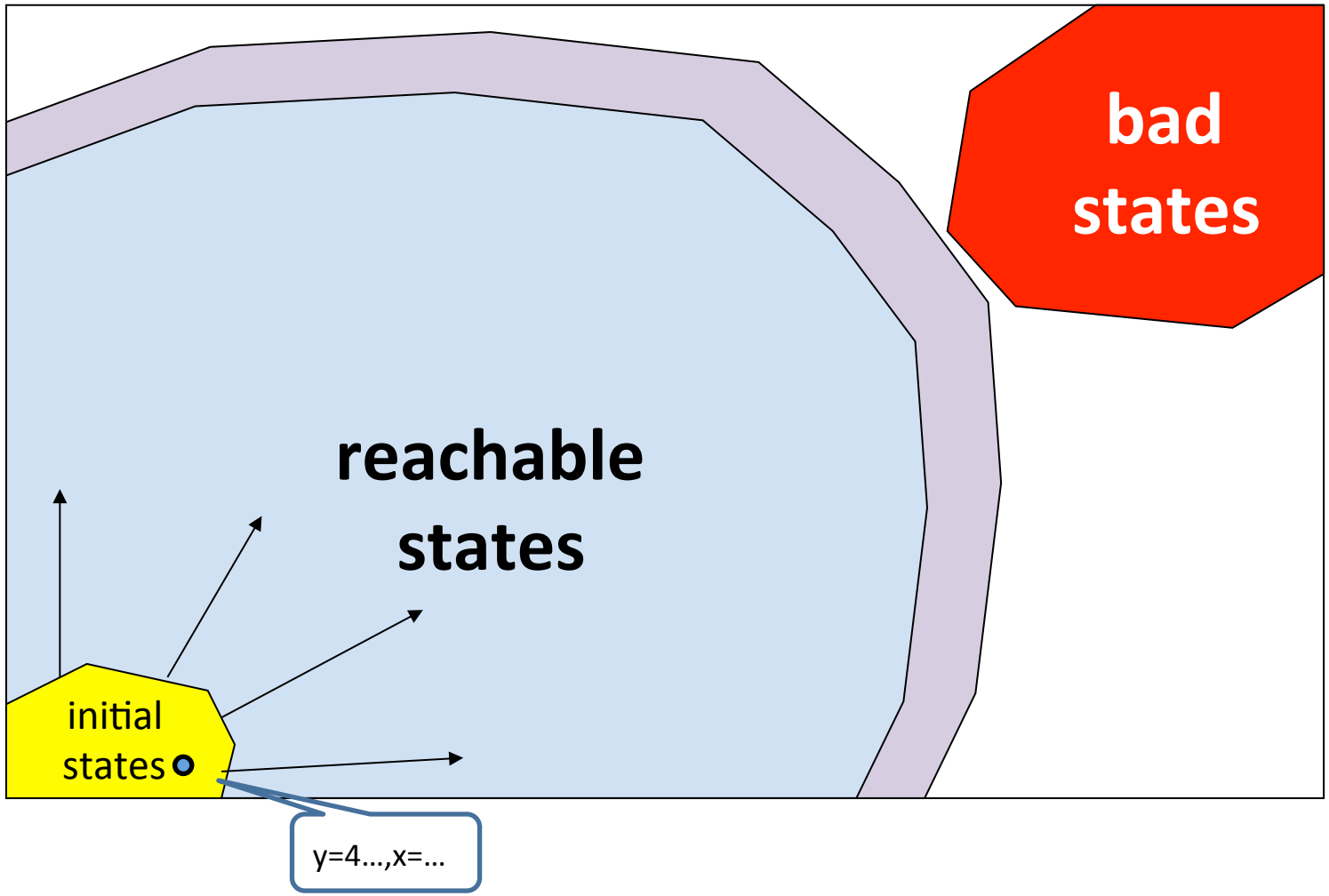
“State transformer” semantics



“State transformer” semantics

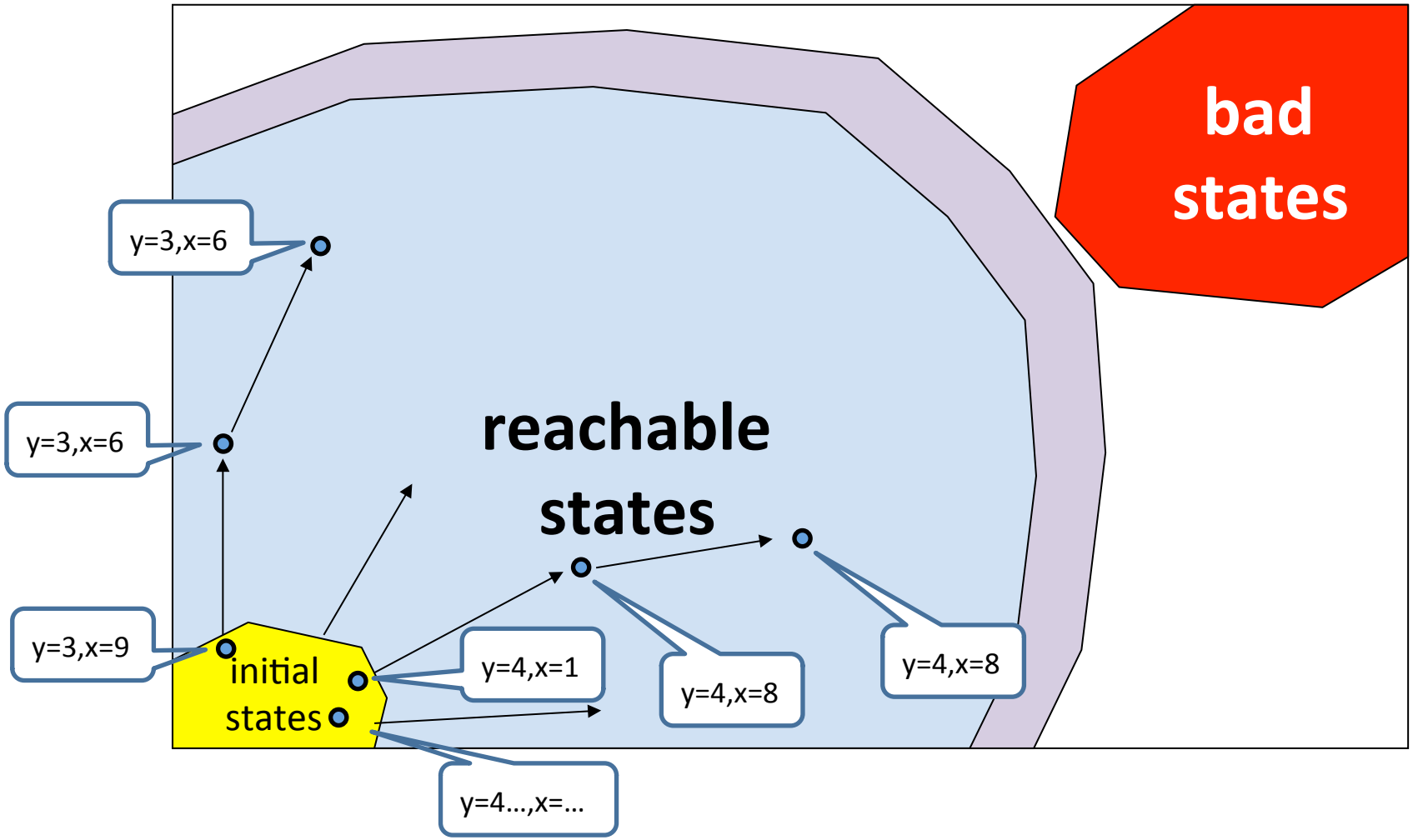


“State transformer” semantics



“State transformer” semantics

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



“Standard” (collecting) semantics

(“sets-of states-transformer”)

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

{(y,x) | y,x ∈ Nat}

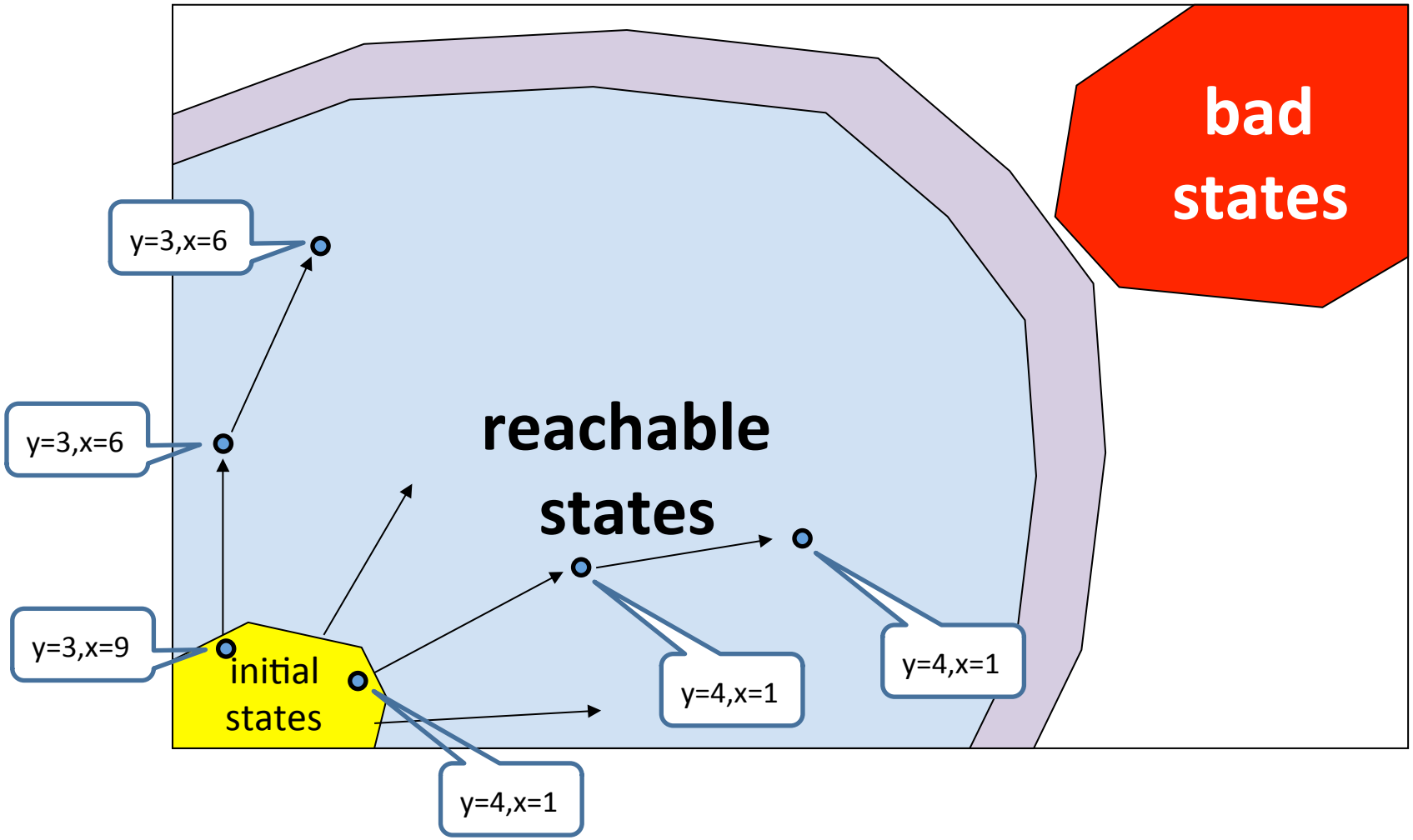
“Standard” (collecting) semantics

(“sets-of states-transformer”)

```
y = ?;                          {(y=3, x=9),(y=4,x=1),(y=…, x=…)}
x = y * 2                       {(y=3, x=6),(y=4,x=8),(y=…, x=…)}
if (x % 2 == 0) {               {(y=3, x=6),(y=4,x=8),(y=…, x=…)}
   y = 42;                      {(y=42, x=6),(y=42,x=8),(y=42, x=…)}

} else {
   y  = 73;                     { }
   foo();                       { }
}
assert (y == 42);               {(y=42, x=6),(y=42,x=8),(y=42, x=…)}
```

Yes

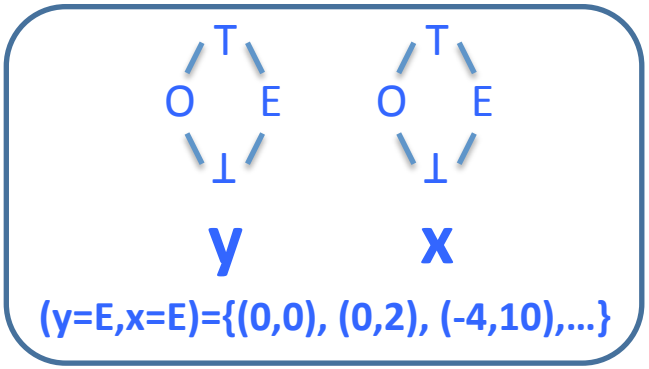
“Set-of-states transformer” semantics



“Abstract-state transformer” semantics

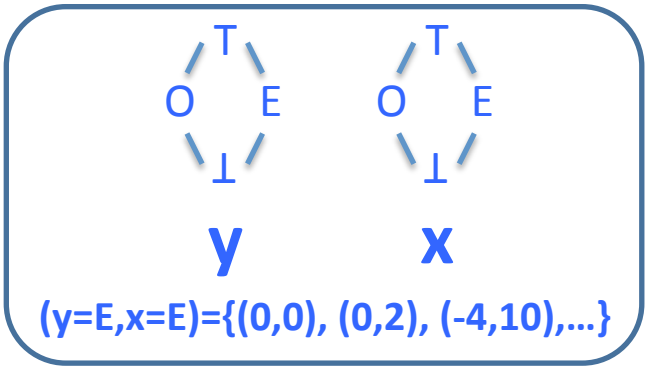
```
y = ?;
x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

y=T, x=T



“Abstract-state transformer” semantics

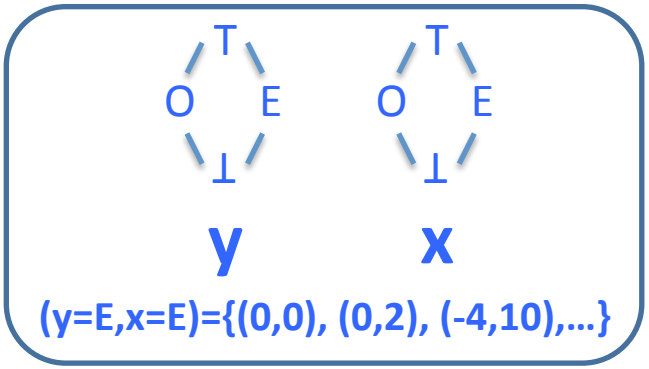
```
y = ?;                    y=T, x=T
x = y * 2                 y=T, x=E
if (x % 2 == 0) {         y=T, x=E
   y = 42;                y=T, x=E
}  else {
   y  = 73;               …
   foo();                 …
}
assert (y == 42); y=E, x=E
```



Yes/?/No

“Abstract-state transformer” semantics

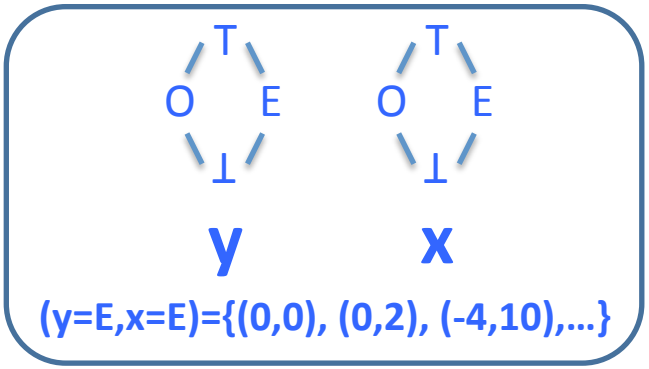
```
y = ?;                    y=T, x=T
x = y * 2                 y=T, x=E
if (x % 2 == 0) {         y=T, x=E
   y = 42;                y=T, x=E
}  else {
   y  = 73;               …
   foo();                 …
}
assert (y == 42); y=E, x=E
```



Yes/?/No

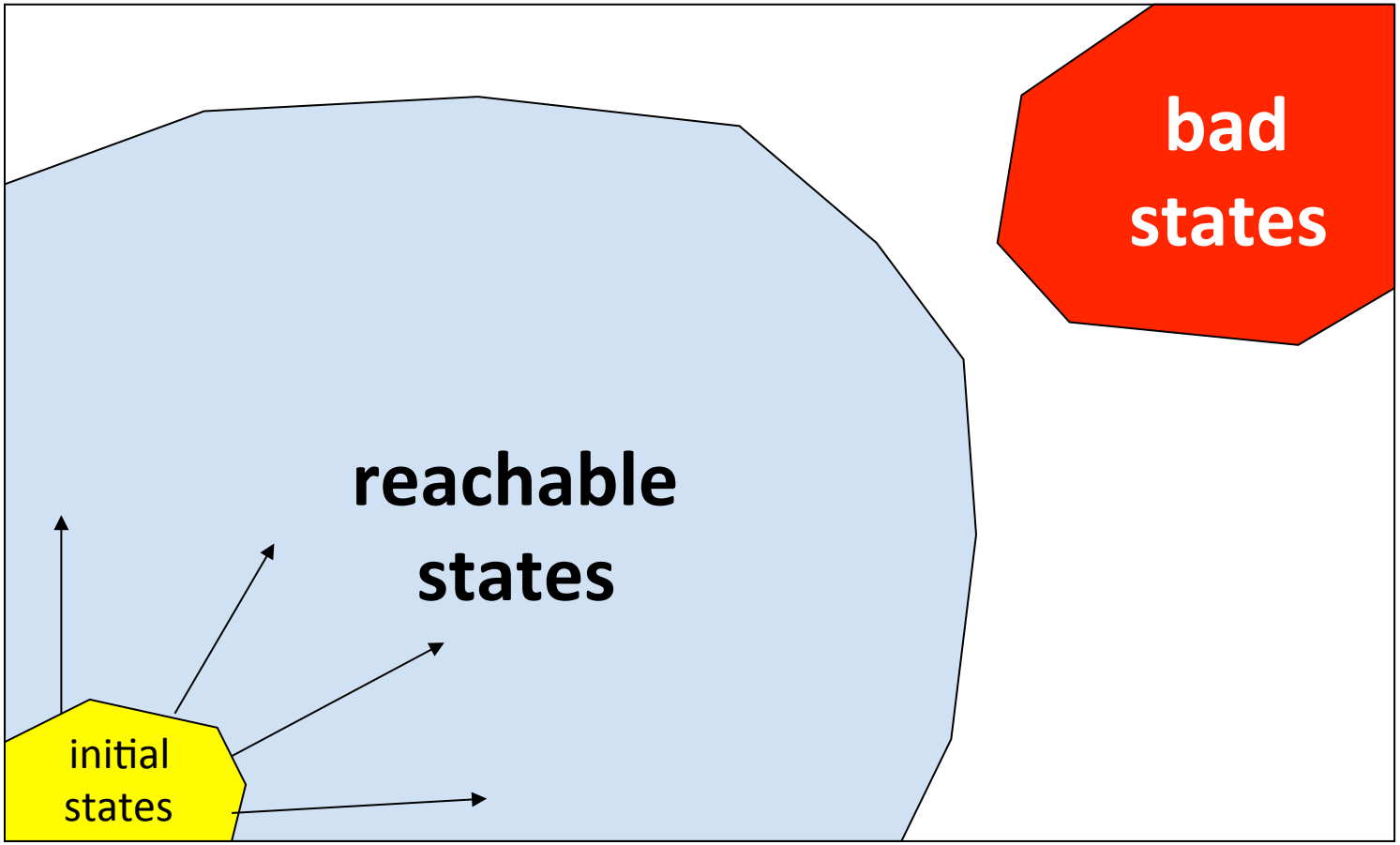
“Abstract-state transformer” semantics

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

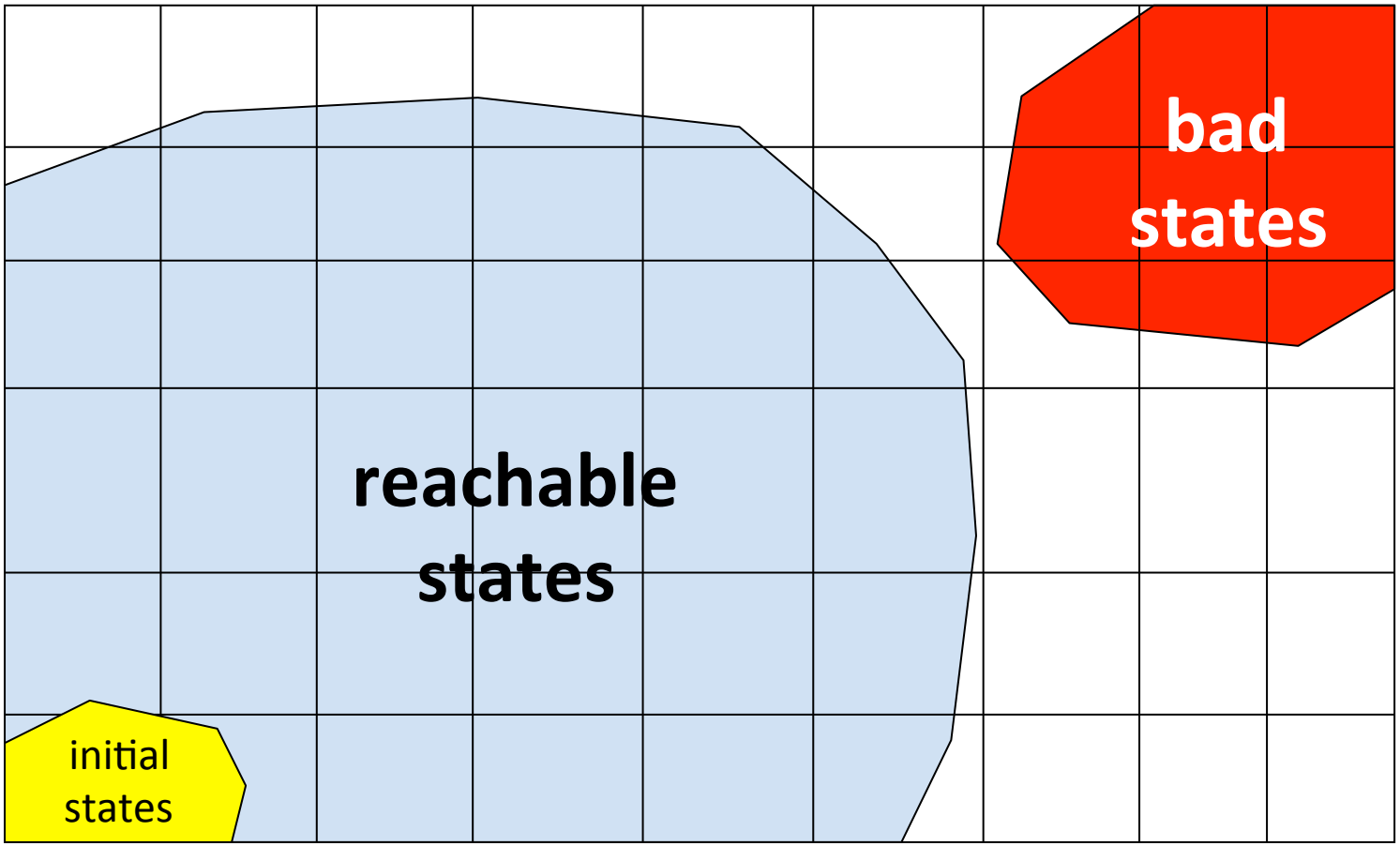


?

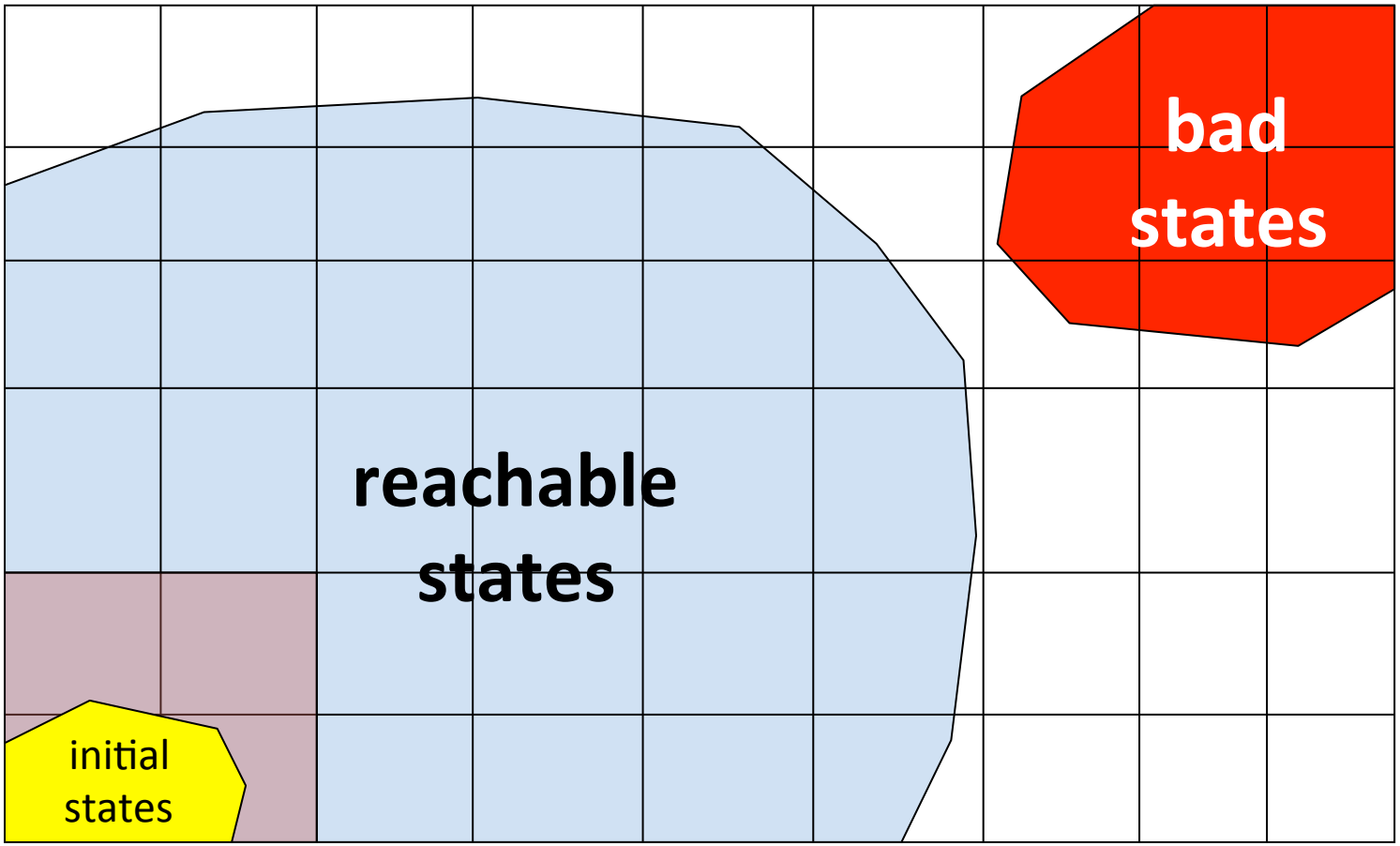
“Abstract-state transformer” semantics



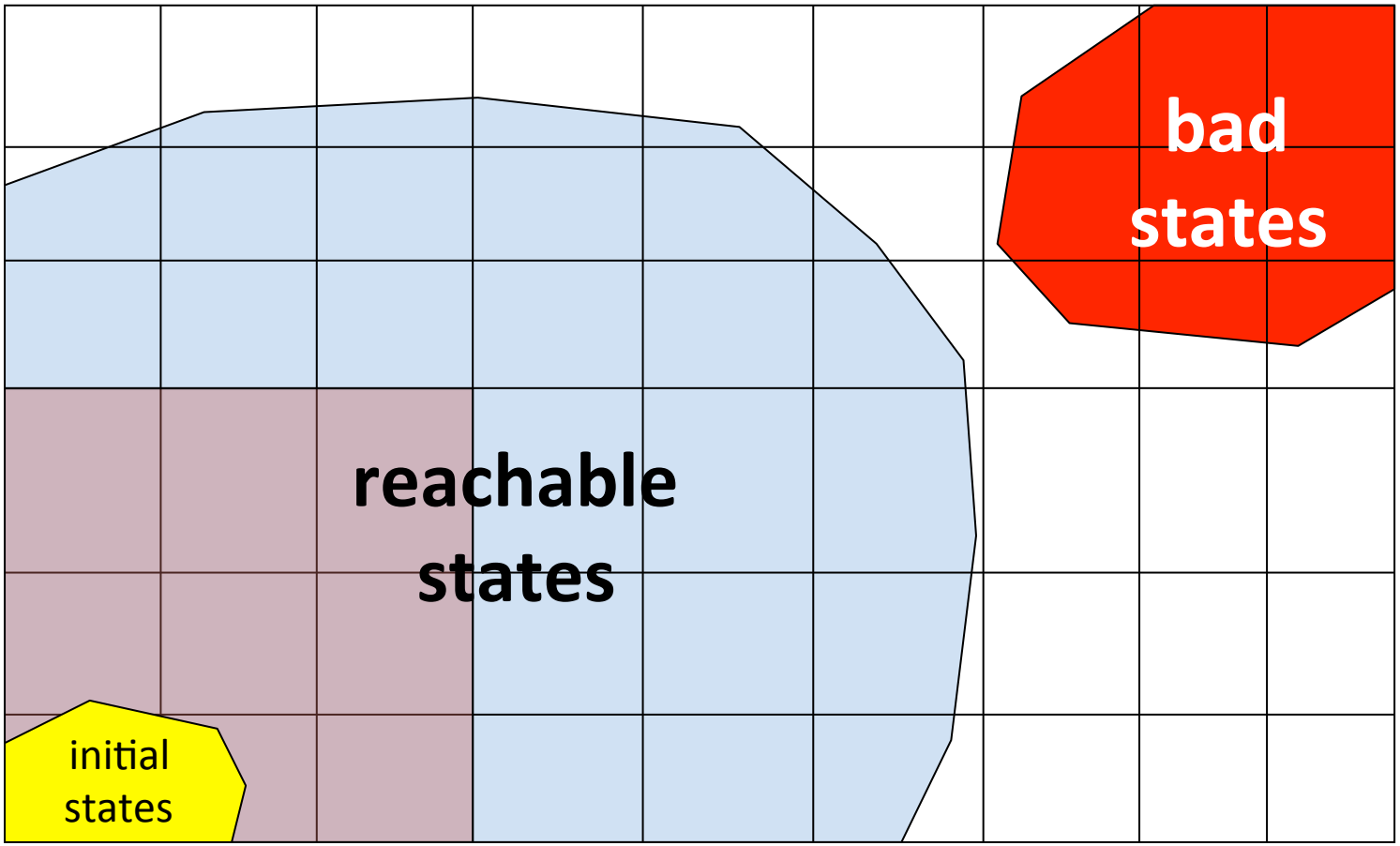
“Abstract-state transformer” semantics



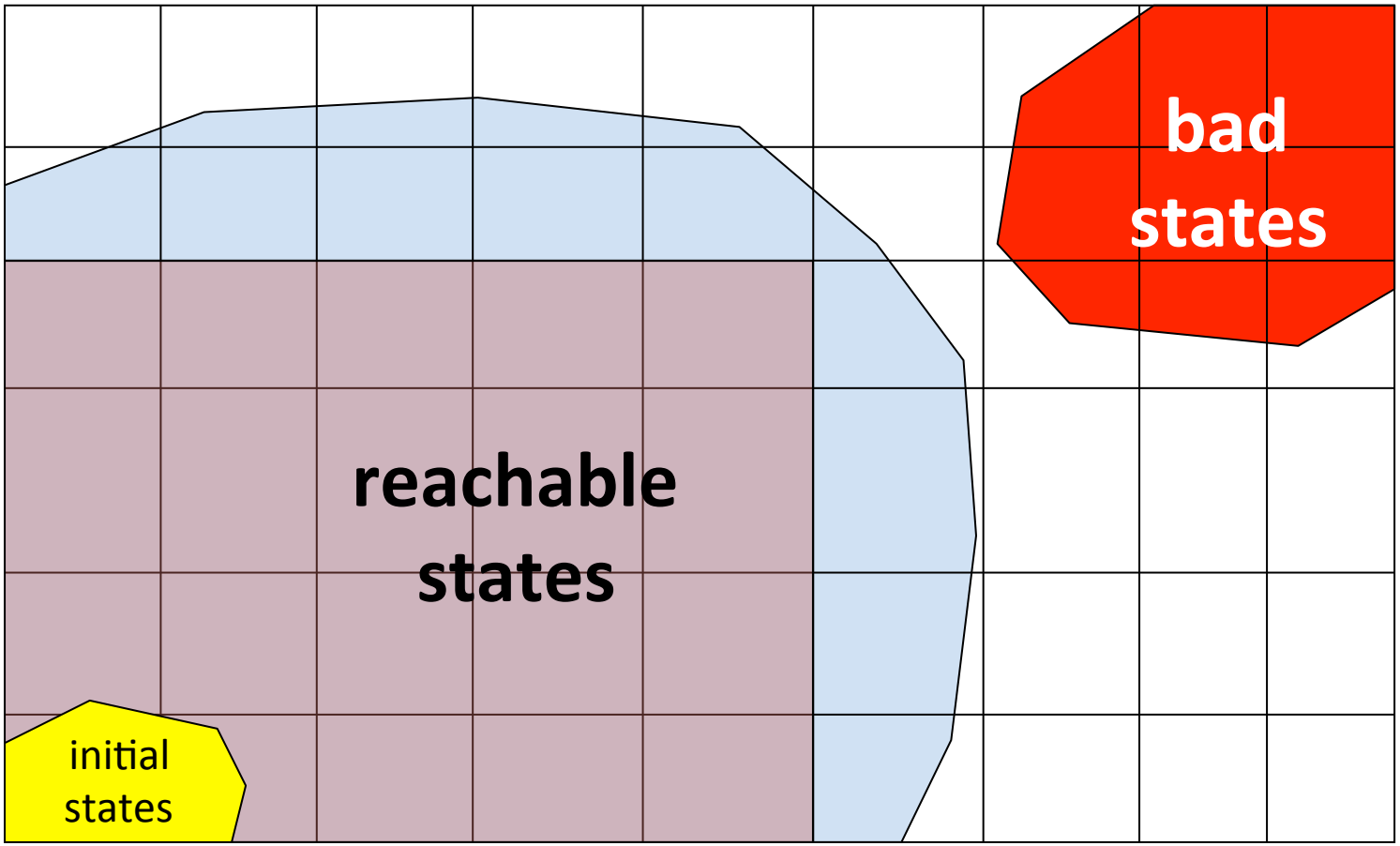
“Abstract-state transformer” semantics



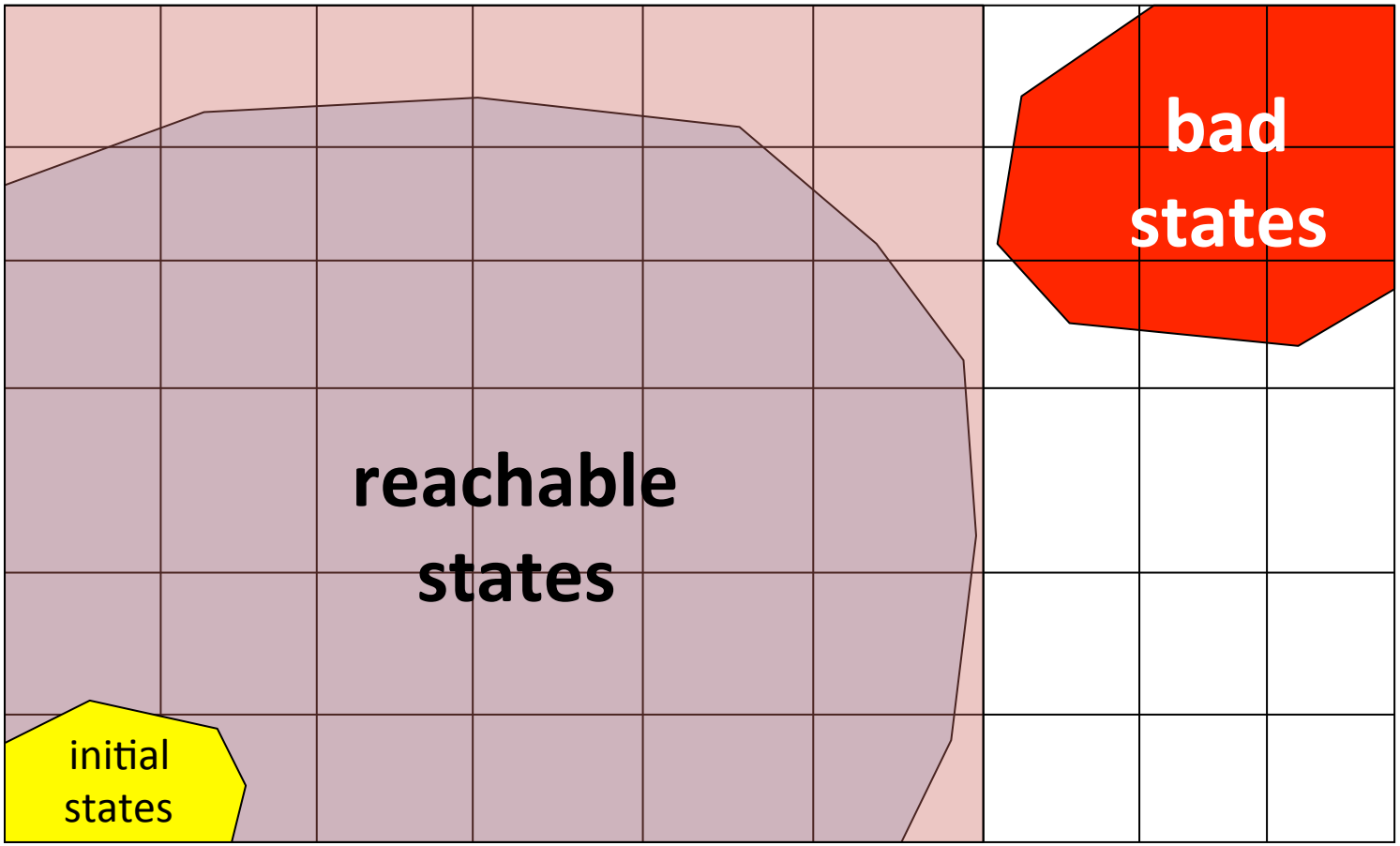
“Abstract-state transformer” semantics



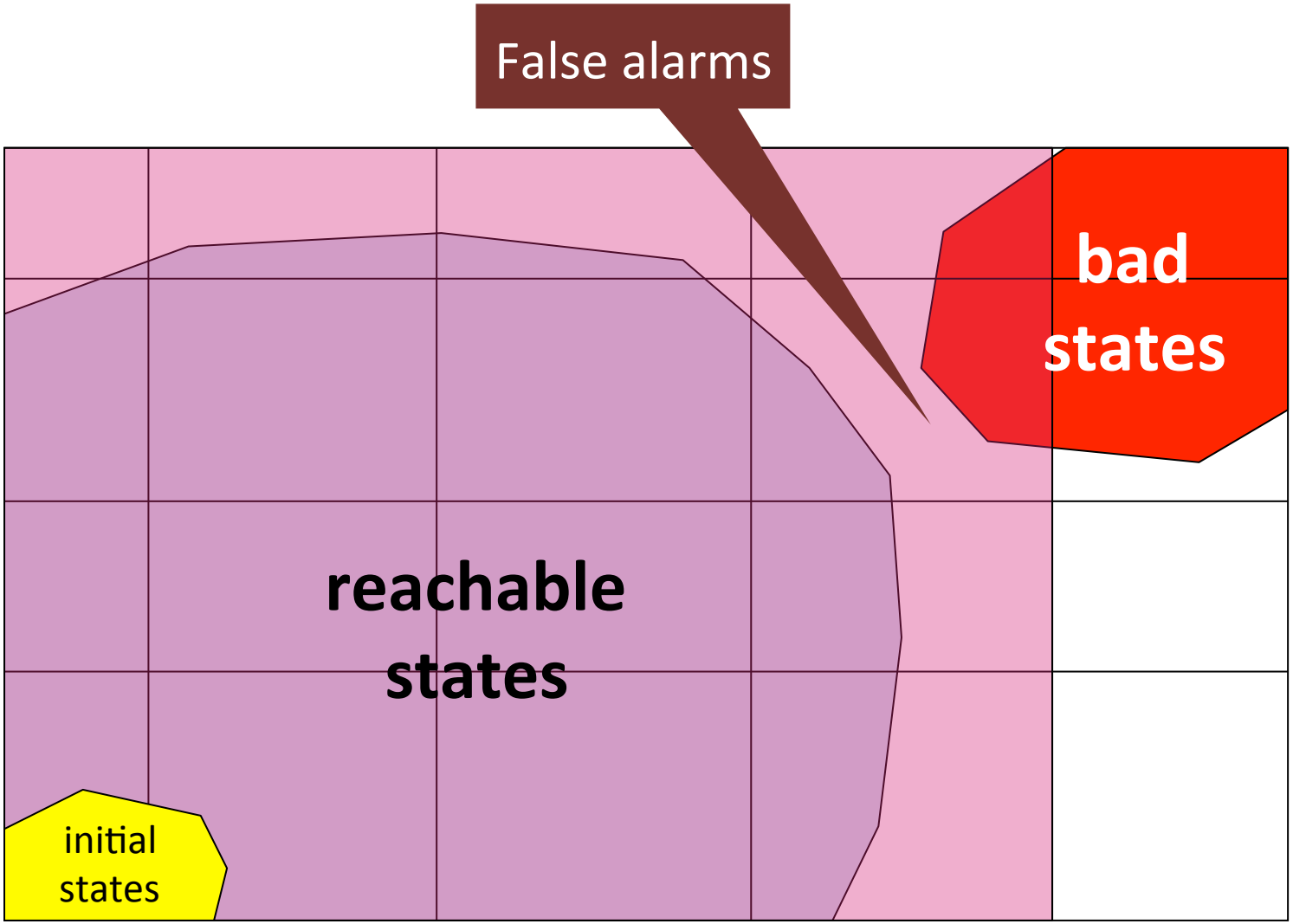
Technique: explore abstract states



Sound: cover all reachable states

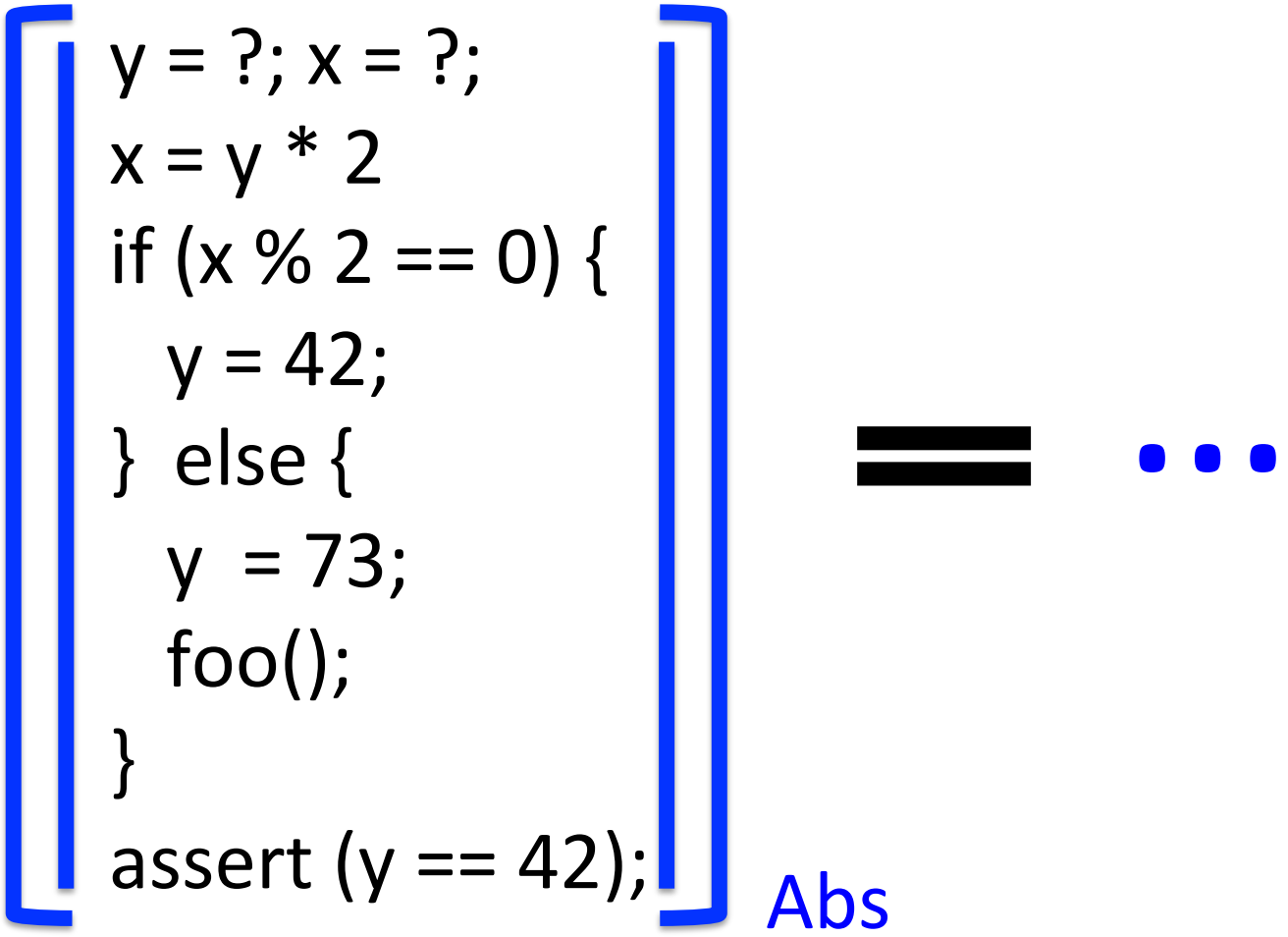


Imprecise abstraction



37

What does P mean?



Programming Languages

- Syntax
  - "how do I write a program?"
  - BNF
  - "Parsing"
- Semantics
  - "What does my program mean?"
  - …

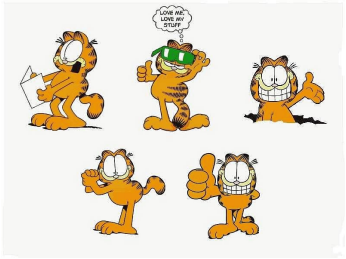
Program semantics

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

Program semantics

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

- Cat-transformer



What semantics do we want?

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

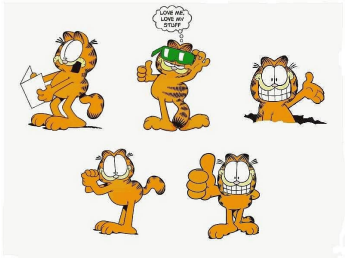
Operational Semantics

Recap

Program semantics

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

- Cat-transformer



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." / page 1

Formal semantics

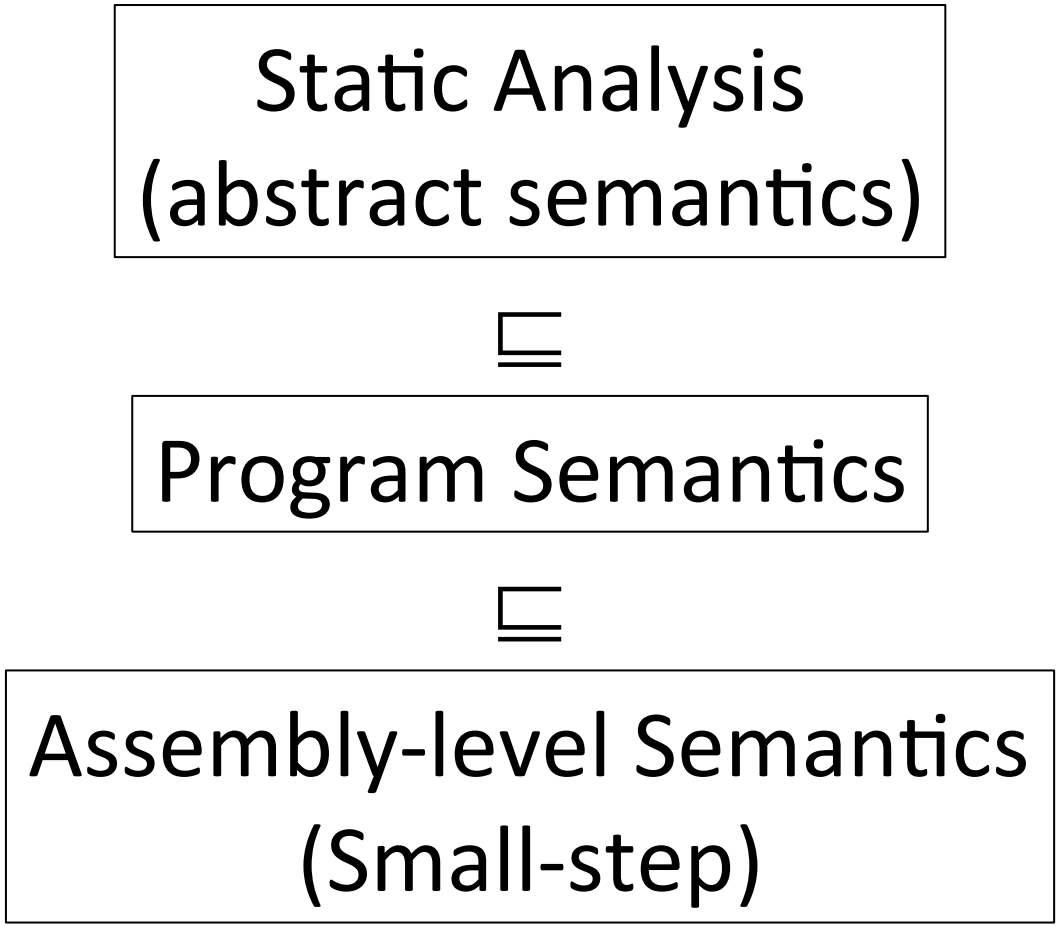
"This theory allows a program to be manipulated like a formula – that is to say, its properties can be calculated."

Gérard 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?

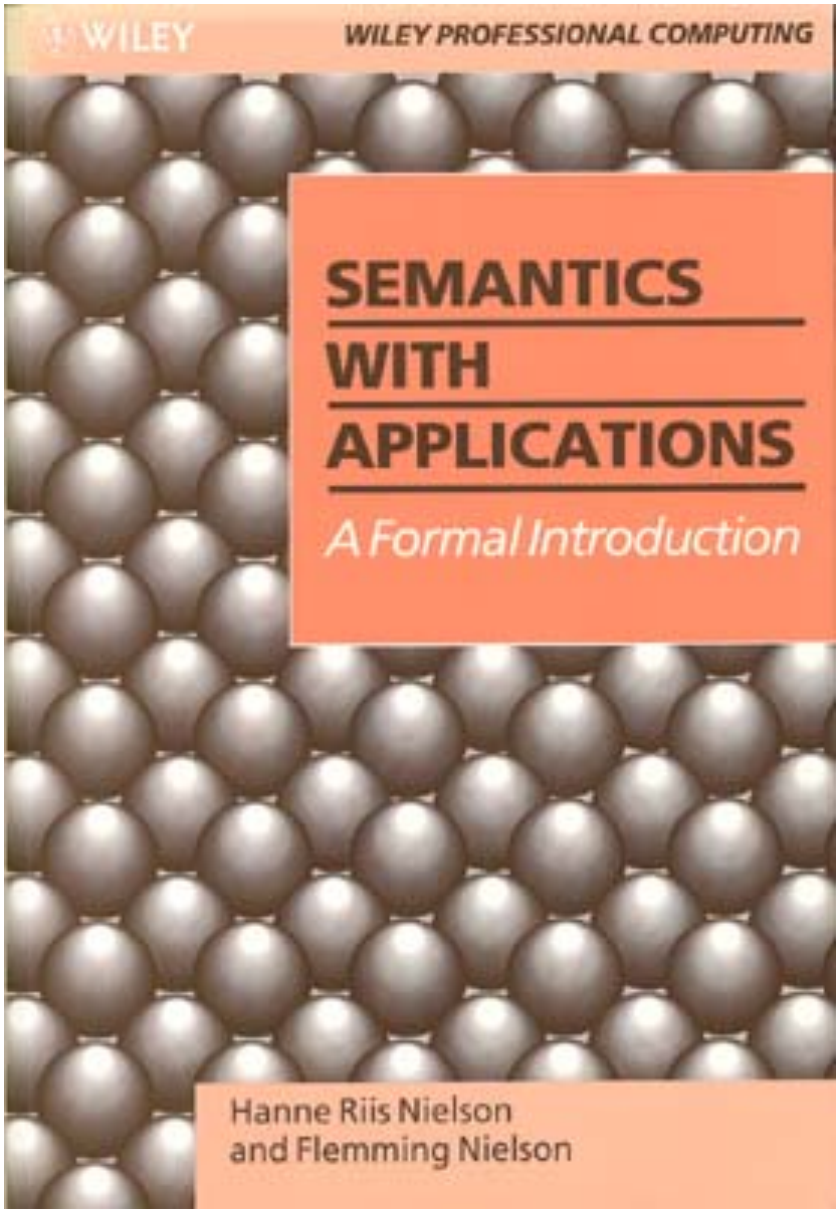
Levels of abstractions and applications



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]

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



Operational Semantics

A simple imperative language: **While**

Abstract syntax:

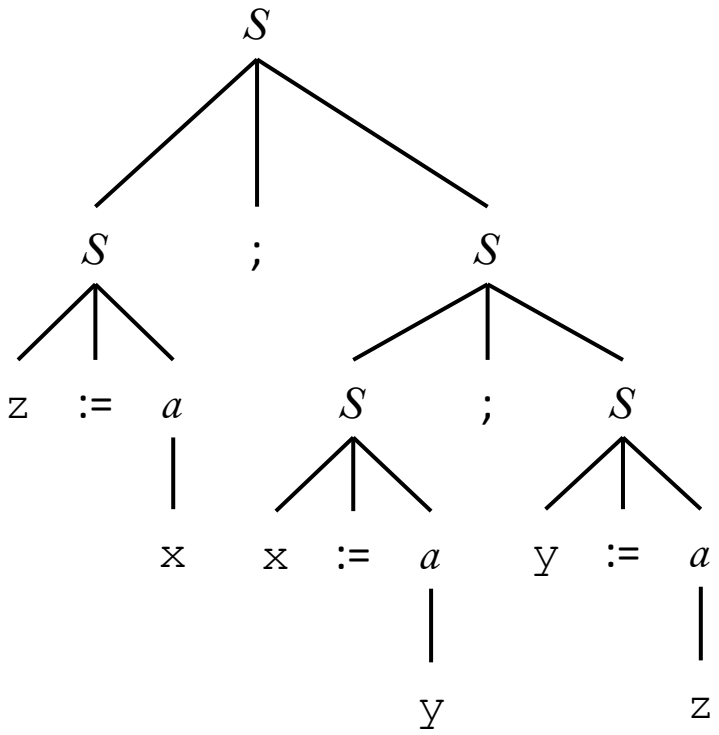
$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$$

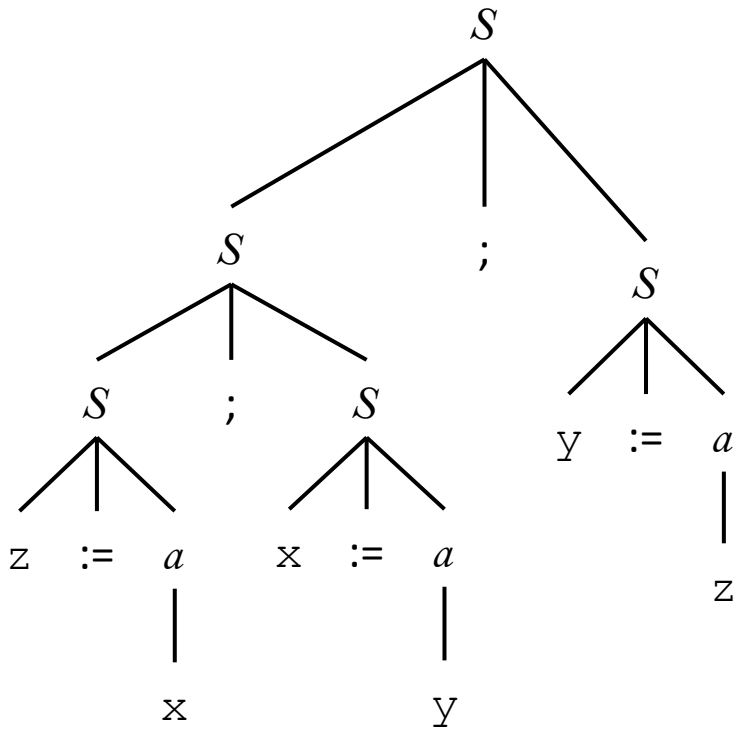
$$b ::= \texttt{true} \mid \texttt{false}$$
$$\mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

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

Concrete Syntax vs. Abstract Syntax

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



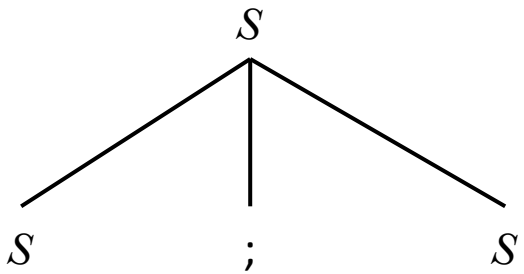


z:=x; (x:=y; y:=z)

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

Exercise: draw an AST

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



Syntactic categories

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

Semantic categories

$\mathbf{Z}$	Integers {0, 1, -1, 2, -2, …}
$\mathbf{T}$	Truth values $\{\mathbf{ff}, \mathbf{tt}\}$
State	**Var → Z**
Example state:	s=[x↦5, y↦7, z↦0]
Lookup:	s x = 5
Update:	s[x↦6] = [x↦6, y↦7, z↦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 $\mathcal{A}[\![\ \textbf{Aexp}\ ]\!] : \textbf{State} \rightarrow \textbf{Z}$
- Defined by induction on the syntax tree

  $\mathcal{A}[\![\ n\ ]\!]\ s = n$

  $\mathcal{A}[\![\ x\ ]\!]\ s = s\ x$

  $\mathcal{A}[\![\ a_1 + a_2\ ]\!]\ s = \mathcal{A}[\![\ a_1\ ]\!]\ s\ + \mathcal{A}[\![\ a_2\ ]\!]\ s$

  $\mathcal{A}[\![\ a_1 - a_2\ ]\!]\ s = \mathcal{A}[\![\ a_1\ ]\!]\ s - \mathcal{A}[\![\ a_2\ ]\!]\ s$

  $\mathcal{A}[\![\ a_1 * a_2\ ]\!]\ s = \mathcal{A}[\![\ a_1\ ]\!]\ s \times \mathcal{A}[\![\ a_2\ ]\!]\ s$

  $\mathcal{A}[\![\ (a_1)\ ]\!]\ s = \mathcal{A}[\![\ a_1\ ]\!]\ s$ --- not needed

  $\mathcal{A}[\![\ -a\ ]\!]\ s = 0 - \mathcal{A}[\![\ a_1\ ]\!]\ s$
- Compositional
- Properties can be proved by structural induction

Arithmetic expression exercise

Suppose s x = 3

Evaluate $\mathcal{A}\,[\![\texttt{x+1}]\!]$ s

Semantics of boolean expressions

- Boolean expressions are side-effect free
- Semantic function $\mathcal{B}[\![\textbf{Bexp}]\!] : \textbf{State} \rightarrow \textbf{T}$
- Defined by induction on the syntax tree

  $\mathcal{B}[\![\text{true}]\!] \text{ s} = \text{tt}$

  $\mathcal{B}[\![\text{false}]\!] \text{ s} = \text{ff}$

  $\mathcal{B}[\![a_1 = a_2]\!] \text{ s} =$

  $\mathcal{B}[\![a_1 \leq a_2]\!] \text{ s} =$

  $\mathcal{B}[\![b_1 \wedge b_2]\!] \text{ s} =$

  $\mathcal{B}[\![\neg b]\!] \text{ s} =$

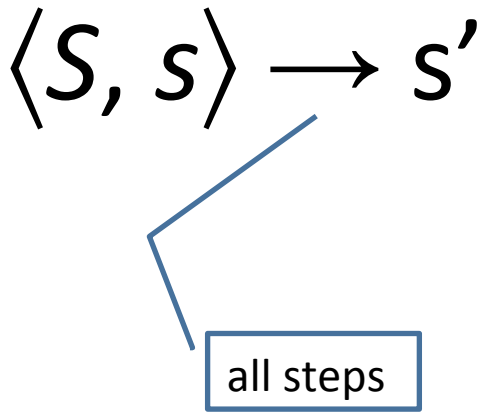
Operational semantics

- 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

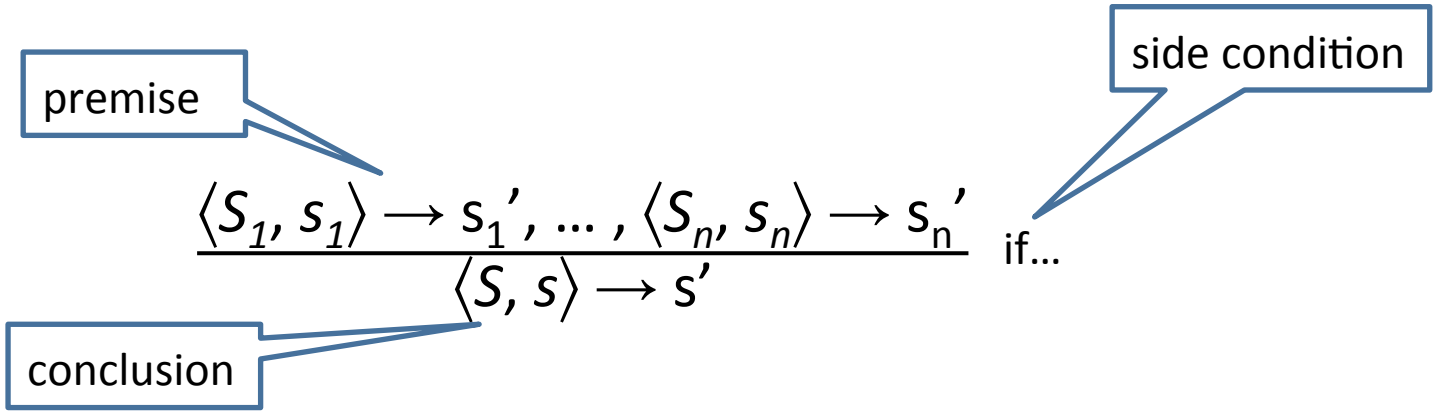
- Transitions

  $\langle S, s \rangle \rightarrow s'$ 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**

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

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

axioms

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

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

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

Natural semantics for **While**

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

Non-compositional

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

Example

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

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

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

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

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

Derivation trees

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

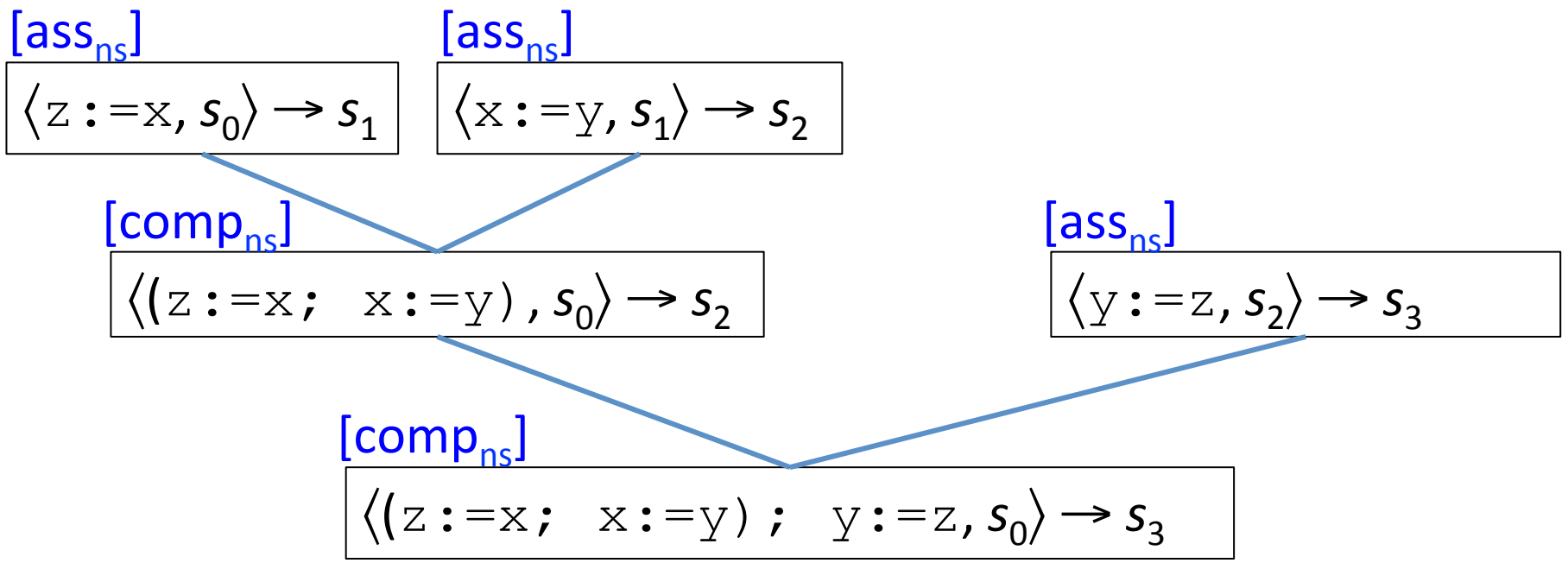
Derivation tree example 1

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

$$\dfrac{\dfrac{[\text{ass}_{ns}]\quad \langle z:=x, s_0\rangle \to s_1 \qquad [\text{ass}_{ns}]\quad \langle x:=y, s_1\rangle \to s_2}{[\text{comp}_{ns}]\quad \langle (z:=x;\ x:=y), s_0\rangle \to s_2} \qquad [\text{ass}_{ns}]\quad \langle y:=z, s_2\rangle \to s_3}{[\text{comp}_{ns}]\quad \langle (z:=x;\ x:=y);\ y:=z, s_0\rangle \to s_3}$$

Derivation tree example 1

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

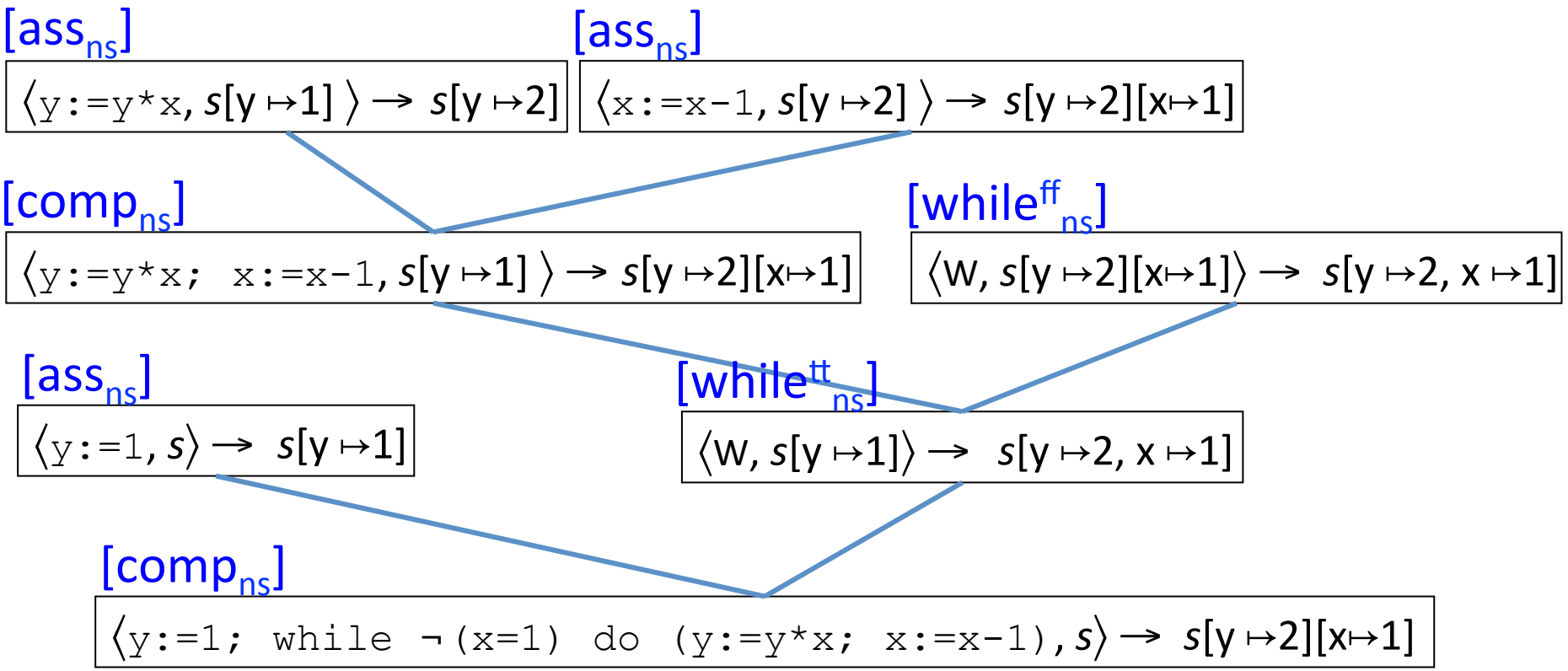


Top-down evaluation via derivation trees

- 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=`while ¬(x=1) 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:

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

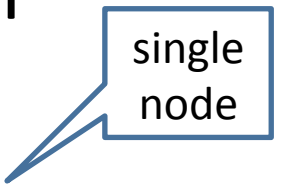
Using the rule applications above, we can construct the following derivation tree:

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

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



The semantic function S_{ns}

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

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

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

- Examples:

  $S_{ns}[\![\texttt{skip}]\!]s = s$

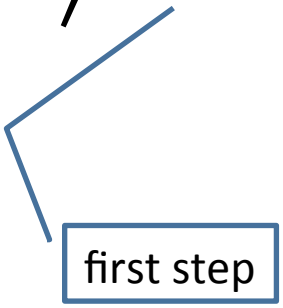
  $S_{ns}[\![\texttt{x:=1}]\!]s = s\,[x \mapsto 1]$

  $S_{ns}[\![\texttt{while true do skip}]\!]s$ = undefined

Structural operating semantics (SOS)

- aka “Small-step semantics”

$$\langle S, s \rangle \Rightarrow \langle S', s' \rangle$$

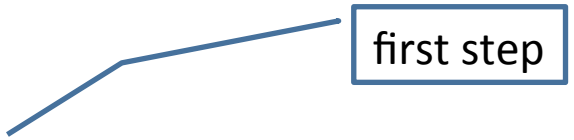


Structural operational semantics

- 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 $\langle S, s \rangle \Rightarrow \gamma$

Structural semantics for **While**

$[ass_{sos}]$ $\langle x{:=}a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[skip_{sos}]$ $\langle \texttt{skip}, s\rangle \Rightarrow s$

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

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

When does this happen?

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

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

Structural semantics for **While**

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

[while$_{sos}$]

Derivation sequences

- A derivation sequence of a statement S starting in state s is either
- A **finite** sequence $\gamma_0, \gamma_1, \gamma_2 \ldots, \gamma_k$ 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 $\gamma_0, \gamma_1, \gamma_2, \ldots$ 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↦5, y↦7, z↦0]

$$\langle (\texttt{z:=x; x:=y}); \texttt{y:=z}, s_0\rangle$$
$$\Rightarrow \langle \texttt{x:=y; y:=z}, s_0[\texttt{z}\mapsto 5]\rangle$$
$$\Rightarrow \langle \texttt{y:=z}, (s_0[\texttt{z}\mapsto 5])[\texttt{x}\mapsto 7]\rangle$$
$$\Rightarrow ((s_0[\texttt{z}\mapsto 5])[\texttt{x}\mapsto 7])[\texttt{y}\mapsto 5]$$

- Derivation tree for first step:

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

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; while ¬(x=1) 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_{sos}: **Stm** $\rightarrow$ (**State** $\hookrightarrow$ **State**)

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

- Examples:

  $S_{sos}[\![\texttt{skip}]\!]\, s = s$

  $S_{sos}[\![\texttt{x:=1}]\!]\, s = s\,[x \mapsto 1]$

  $S_{sos}[\![\texttt{while true do skip}]\!]\, s =$ undefined

An equivalence result

- For every statement in **While**

$$S_{ns}[\![S]\!] = S_{sos}[\![S]\!]$$

- Proof in pages 40-43

The End