Tentative Schedule

6/3	introduction
13/3	javascript
20/3	Haskel
27/3	No class
3/4	Operational Semantics
17/4	Denotational Semantics
24/4	Axiomatic Semantics
2/5	Exception and continuation
8/5, 15/5, 22/5	Type Systems
29/5, 5/6, 12/6	Concurrency
19/6	Domain Specific Languages
22/6	Summary class

Operational Semantics

Mooly Sagiv

Semantics with Applications

Chapter 2

H. Nielson and F. Nielson

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

The Formal Semantics of Programming Languages

An Introduction

Glynn Winskel

Syntax vs. Semantics

- The pattern of formation of sentences or phrases in a language
- Examples
  - Regular expressions
  - Context free grammars

- The study or science of meaning in language
- Examples
  - Interpreter
  - Compiler
  - Better mechanisms will be given today

Benefits of Formal Semantics

- Programming language design
  - hard- to-define= hard-to-implement=hard-to-use
- Programming language implementation
- Programming language understanding
- Program correctness
- Program equivalence
- Compiler Correctness
- Automatic generation of interpreter
- But probably not
  - Automatic compiler generation

Alternative Formal Semantics

- Operational Semantics
  - The meaning of the program is described "operationally"
  - Natural Large Step Operational Semantics
  - Structural Small Step Operational Semantics
- Denotational Semantics
  - The meaning of the program is an input/output relation
  - Mathematically challenging but complicated
- Axiomatic Semantics
  - The meaning of the program are observed properties

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x
   while (y>0)   {
        z = z * y ;
        y = y – 1;
   }
 return z
}
```

$[x \mapsto 3]$

$[x \mapsto 3, z \mapsto \perp, y \mapsto \perp]$

$[x \mapsto 3, z \mapsto 1, y \mapsto \perp]$

$[x \mapsto 3, z \mapsto 1, y \mapsto 3]$

$[x \mapsto 3, z \mapsto 1, y \mapsto 3]$

$[x \mapsto 3, z \mapsto 3, y \mapsto 3]$

$[x \mapsto 3, z \mapsto 3, y \mapsto 2]$

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x
   while (y>0)   {
        z = z * y ;
        y = y – 1;
   }
return z
}
```

[x⟼3, z⟼3, y⟼2]

[x⟼3, z⟼3, y⟼2]

[x⟼3, z⟼6, y⟼2]

[x⟼3, z⟼6, y⟼1]

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x
   while (y>0)   {
        z = z * y ;
        y = y – 1;
    }
return z
}
```

[x⟼3, z⟼6, y⟼1]

[x⟼3, z⟼6, y⟼1]

[x⟼3, z⟼6, y⟼1]

[x⟼3, z⟼6, y⟼0]

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x
   while (y>0)   {
        z = z * y ;
        y = y – 1;
    }
return z
}
```

$[x \mapsto 3, z \mapsto 6, y \mapsto 0]$ (at while (y>0))

$[x \mapsto 3, z \mapsto 6, y \mapsto 0]$ (at return z)

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x;
   while (y>0)   {
        z = z * y ;
        y = y – 1;
   }
return 6
}
```

$[x \mapsto 3, z \mapsto 6, y \mapsto 0]$

$[x \mapsto 3, z \mapsto 6, y \mapsto 0]$

Denotational Semantics

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x ;
   while (y>0)   {
        z = z * y ;
        y = y – 1;
    }
return z;
}
```

f=λx. if x = 0 then 1 else x * f(x -1)

Axiomatic Semantics

```
{x=n}
 int fact(int x) {  int z, y;
 z = 1;
{x=n ∧ z=1}
 y= x
{x=n ∧ z=1 ∧ y=n}
  while
    {x=n  ∧ y ≥0 ∧ z=n! / y!}
 (y>0)  {
       {x=n ∧ y >0 ∧  z=n! / y!}
        z = z * y ;
      {x=n ∧ y>0 ∧  z=n!/(y-1)!}
        y = y – 1;
      {x=n ∧ y ≥0 ∧  z=n!/y!}
    } return z} {x=n ∧ z=n!}
```

Operational Semantics

Natural Large Step Semantics

Operational Semantics of Arithmetic Expressions

Aexp → | number

| Axp PLUS Aexp

| Aexp MINUS Aexp

| Aexp MUL Aexp

| UMINUS Aexp

$A[\![\,]\!]: \text{Aexp} \rightarrow Z$

$$A[\![n]\!] = \text{val}(n)$$

$$A[\![e_1 \text{ PLUS } e_2]\!] = A[\![e_1]\!] + A[\![e_2]\!]$$

$$A[\![e_1 \text{ MINUS } e_2]\!] = A[\![e_1]\!] - A[\![e_2]\!]$$

$$A[\![e_1 \text{ MUL } e_2]\!] = A[\![e_1]\!] * A[\![e_2]\!]$$

$$A[\![\text{UMINUS } e]\!] = A[\![e]\!]$$

Handling Variables

Aexp → | number
| variable
| Aexp PLUS Aexp
| Aexp MINUS Aexp
| Aexp MUL Aexp
| UMINUS Exp

- Need the notions of states
- States State = Var $\rightarrow$ Z
- Lookup in a state s: s x
- Update of a state s: s [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
- $A[\![\text{Aexp}]\!] : \text{State} \rightarrow Z$
- Defined by induction on the syntax tree
  - $A[\![n]\!] s = n$
  - $A[\![x]\!] s = s\ x$
  - $A[\![e_1 \text{ PLUS } e_2]\!] s = A[\![e_1]\!] s + A[\![e_2]\!] s$
  - $A[\![e_1 \text{ MUL } e_2]\!] s = A[\![e_1]\!] s * A[\![e_2]\!] s$
  - $A[\![\text{UMINUS } e]\!] s = -A[\![e]\!] s$
- Compositional
- Properties can be proved by structural induction

Semantics of Boolean expressions

- Assume that Boolean expressions are side-effect free
- T={ff, tt}
- B⟦ Bexp ⟧ : State → T
- Defined by induction on the syntax tree
  - B⟦ true ⟧ s = tt
  - B⟦ false ⟧ s = ff
  - $B[\![e_1 = e_2]\!] s = \begin{cases} tt \text{ if } A[\![e_1]\!] s = A[\![e_2]\!] s \\ ff \text{ if } A[\![e_1]\!] s \neq A[\![e_2]\!] s \end{cases}$
  - $B[\![e_1 \wedge e_2]\!] s = \begin{cases} tt \text{ if } B[\![e_1]\!] s = tt \text{ and } B[\![e_2]\!] = tt \\ ff \text{ if } B[\![e_1]\!] s = ff \text{ or } B[\![e_2]\!] s = ff \end{cases}$
  - $B[\![e_1 \geq e_2]\!] s =$

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 N={0, 1, 2, …}
  - Truth values T={ff, tt}
  - States State = Var $\rightarrow$ N
  - Lookup in a state s: s x
  - Update of a state s: s [x $\mapsto$ 5]

Natural Operational Semantics

- Describe the “overall” effect of program constructs
- Ignores non terminating computations

Natural Semantics

- Notations
  - <S, s> - the program statement S is executed on input state s
  - s representing a terminal (final) state
- For every statement S, write meaning rules
  $<S, i> \rightarrow o$
  "If the statement S is executed on an input state *i*, it terminates and yields an output state *o*"
- The meaning of a program P on an input state s is the set of outputs states *o* such that $<P, i> \rightarrow o$
- The meaning of compound statements is defined using the meaning immediate constituent statements
- Inductive definitions
- Notice that $\rightarrow$ means large-step here in contrast to the first lecture where $\rightarrow$ means small-step

Natural Semantics for While

axioms

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

$[skip_{ns}]\ \langle \mathbf{skip}, s\rangle \to s$

rules

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

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

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

Natural Semantics for While (More rules)

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

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

A Derivation Tree

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

$<skip;\ x := x + 1, s_0> \to s_0[x \mapsto 1]>$

comp_{ns}

$<skip, s_0> \to s_0$ $<x := x + 1, s_0> \to s_0[x \mapsto 1]>$

skip_{ns} ass_{ns}

An Example Derivation Tree

$\langle (x := x+1;\ y := x+1)\ ;\ \ z := y),\ s0 \rangle \rightarrow s0[x \mapsto 1][y \mapsto 2][z \mapsto 2]$

comp_{ns}

$\langle x := x+1;\ y := x+1,\ s0 \rangle \rightarrow s0[x \mapsto 1][y \mapsto 2]$

$\langle z := y, s0[x \mapsto 1][y \mapsto 2] \rangle \rightarrow s0[x \mapsto 1][y \mapsto 2][z \mapsto 2]$

comp_{ns}

$\langle x := x+1;\ s0 \rangle \rightarrow s0[x \mapsto 1]$

$\langle y := x+1,\ s0[x \mapsto 1] \rangle \rightarrow s0[x \mapsto 1][y \mapsto 2]$

ass_{ns}

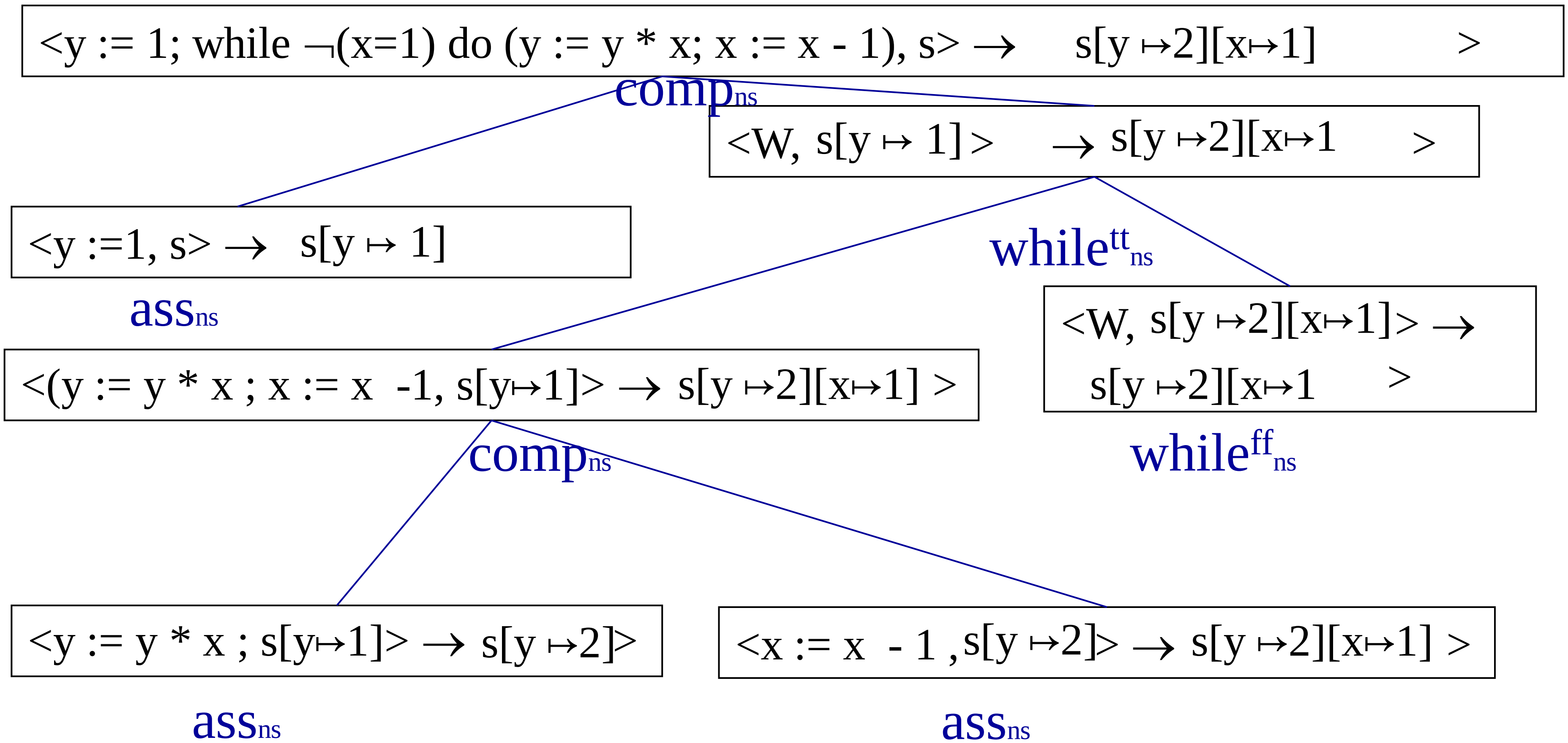
ass_{ns}

Top Down Evaluation of Derivation Trees

- Given a program S and an input state s
- Find an output state s' such that
  $<S, s> \rightarrow s'$
- 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

Example of Top Down Tree Construction

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



Semantic Equivalence

- S_1 and S_2 are semantically equivalent if
  for all s and s'
  $<S_1, s> \rightarrow s'$ if and only if $<S_2, s> \rightarrow s'$
- Simple example
  "while b do S"
  is semantically equivalent to:
  "if b then (S ; while b do S) else skip"

Deterministic Semantics for While (Theorem 2.9, page 39)

- If <S, s> $\rightarrow s_1$ and <S, s> $\rightarrow s_2$ then $s_1 = s_2$
- The proof uses induction on the shape of derivation trees
  - 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)**
- $S_{ns}[\![S]\!]s = s'$ if $<S, s> \rightarrow s'$ and otherwise $S_{ns}[\![S]\!]s$ is undefined
- Examples
  - $S_{ns}[\![\text{skip}]\!]s = s$
  - $S_{ns}[\![x := 1]\!]s = s\,[x \mapsto 1]$
  - $S_{ns}[\![\text{while true do skip}]\!]s = \text{undefined}$

Structural Operational Semantics

- Emphasizes the individual steps
- For every statement S, write meaning rules <S, *i*> $\Rightarrow \gamma$
  "If the **first** step of executing the statement S on an input state *i* leads to γ"
- Two possibilities for γ
  - γ = <S', s'> The execution of S is not completed, S' is the remaining computation which need to be performed on s'
  - γ = o The execution of S has terminated with a final state o
  - γ is a stuck configuration when there are no transitions
- The meaning of a program P on an input state s is the set of final states that can be executed in arbitrary finite steps
- $\Rightarrow$ means small step as $\rightarrow$ in the first lecture

Structural Semantics for While

axioms

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

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

rules

$$[comp^1_{sos}]\ \frac{\langle S_1, s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S'_1; S_2, s'\rangle}$$

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

Structural Semantics for While
if construct

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

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

Structural Semantics for While while construct

$[\text{while}_{sos}]$ <while b do S, s> $\Rightarrow$
<if b then (S; while b do S) else skip, s>

Derivation Sequences

- A finite derivation sequence starting at <S, s>
  $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$ such that
  - $\gamma_0 = \langle S, s \rangle$
  - $\gamma_i \Rightarrow \gamma_{i+1}$
  - γ_k is either stuck configuration or a final state
- An infinite derivation sequence starting at <S, s>
  $\gamma_0, \gamma_1, \gamma_2 \ldots$ such that
  - $\gamma_0 = \langle S, s \rangle$
  - $\gamma_i \Rightarrow \gamma_{i+1}$
- $\gamma_0 \Rightarrow^i \gamma_i$ in i steps
- $\gamma_0 \Rightarrow^* \gamma_i$ in finite number of steps
- For each step there is a derivation tree

Example

- Let s_0 such that
  $s_0\,x = 5$
  and
  $s_0\,y = 7$
- S = (z:=x; x := y); y := z

Factorial Program

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

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

$< y := 1\ ;\ W, s >$

$\Rightarrow\ < W, s[y \mapsto 1] >$

$\Rightarrow\ < \text{if}\ \neg\ \neg\ (x = 1)\ \text{then skip else}\ ((y := y * x\ ;\ x := x - 1); W), s[y \mapsto 1] >$

$\Rightarrow\ < ((y := y * x\ ;\ x := x - 1); W), s[y \mapsto 1] >$

$\Rightarrow\ < (x := x - 1\ ;\ W), s[y \mapsto 3] >$

$\Rightarrow\ < W\ , s[y \mapsto 3][x \mapsto 2] >$

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

$\Rightarrow\ < ((y := y * x\ ;\ x := x - 1); W), s[y \mapsto 3]\ [x \mapsto 2] >$

$\Rightarrow\ < (x := x - 1\ ;\ W)\ , s[y \mapsto 6]\ [x \mapsto 2] >$

$\Rightarrow\ < W, s[y \mapsto 6][x \mapsto 1] >$

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

$\Rightarrow\ < \text{skip}, s[y \mapsto 6][x \mapsto 1] > \ \Rightarrow\ s[y \mapsto 6][x \mapsto 1]$

Program Termination

- Given a statement S and input s
  - S terminates on s if there exists a finite derivation sequence starting at <S, s>
  - S terminates successfully on s if there exists a finite derivation sequence starting at <S, s> leading to a final state
  - S loops on s if there exists an infinite derivation sequence starting at <S, s>

Properties of the Semantics

- S_1 and S_2 are semantically equivalent if:
  - for all s and γ which is either final or stuck $<S_1, s> \Rightarrow^* \gamma$ if and only if $<S_2, s> \Rightarrow^* \gamma$
  - there is an infinite derivation sequence starting at $<S_1, s>$ if and only if there is an infinite derivation sequence starting at $<S_2, s>$
- Deterministic
  - If $<S, s> \Rightarrow^* s_1$ and $<S, s> \Rightarrow^* s_2$ then $s_1=s_2$
- The execution of $S_1; S_2$ on an input can be split into two parts:
  - execute S_1 on s yielding a state s'
  - execute S_2 on s'

Sequential Composition

- If $<S_1; S_2, s> \Rightarrow^k s''$ then there exists a state s' and numbers k_1 and k_2 such that
  - $<S_1, s> \Rightarrow^{k1} s'$
  - $<S_2, s'> \Rightarrow^{k2} s''$
  - and $k = k_1 + k_2$
- The proof 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** → **(State ↪ State)**
- $S_{sos}[\![S]\!]s = s'$ if $\langle S, s\rangle \Rightarrow^* s'$ and otherwise $S_{sos}[\![S]\!]s$ is undefined

An Equivalence Result

- For every statement S of the While language
  - $\mathcal{S}_{nat}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]$

Extensions to While

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

The **While** Programming Language with Abort

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S| **abort**
- Abort terminates the execution
- No new rules are needed in natural and structural operational semantics
- Statements
  - if x = 0 then abort else y := y / x
  - skip
  - abort
  - while true do skip

Conclusion

- The natural semantics cannot distinguish between looping and abnormal termination (unless the states are modified)
- In the structural operational semantics looping is reflected by infinite derivations and abnormal termination is reflected by stuck configuration

The **While** Programming Language with Non-Determinism

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S| S_1 or S_2
- Either S_1 or S_2 is executed
- Example
  - x := 1 or (x :=2 ; x := x+2)

The While Programming Language with Non-Determinism Natural Semantics

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

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

The While Programming Language with Non-Determinism Structural Semantics

The While Programming Language with Non-Determinism Examples

- x := 1 or (x :=2 ; x := x+2)
- (while true do skip) or (x :=2 ; x := x+2)

Conclusion

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

The **While** Programming Language with Parallel Constructs

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S| S_1 par S_2
- All the interleaving of S_1 or S_2 are executed
- Example
  - x := 1 par (x :=2 ; x := x+2)

The **While** Programming Language with Parallel Constructs Structural Semantics

$$[\text{par}^1_{\text{sos}}]\ \frac{\langle S_1, s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S'_1 \text{ par } S_2, s'\rangle}$$

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

$$[\text{par}^3_{\text{sos}}]\ \frac{\langle S_2, s\rangle \Rightarrow \langle S'_2, s'\rangle}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S_1 \text{ par } S'_2, s'\rangle}$$

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

The **While** Programming Language with Parallel Constructs Natural Semantics

Conclusion

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

The **While** Programming Language with local variables and procedures

- Abstract syntax

  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S|
  begin D_v D_p S **end | call p**

  D_v ::= **var** x := a ; D_v | ε

  D_p ::= **proc** p **is** S ; D_p | ε

Conclusions Local Variables

- The natural semantics can “remember” local states
- Need to introduce stack or heap into state of the structural semantics

Summary

- Operational Semantics is useful for:
  - Language Designers
  - Compiler/Interpreter Writer
  - Programmers
- Natural operational semantics is a useful abstraction
  - Can handle many PL features
  - No stack/ program counter
  - Simple
  - “Mostly” compositional
- Other abstractions exist

Further Reading

- Ankur Taly: Operational Semantics for JavaScript
- Pietro Cenciarelli*?, Alexander Knapp*, *Bernhard Reus*, *and Martin Wirsing*: An Event-Based Structural Operational Semantics of Multi-threaded Java
  Alan Jeffrey and Julian Rathke:Java Jr.: Fully abstract trace semantics for a core Java language