Operational Semantics

Mooly Sagiv

Reference: Semantics with Applications

Chapter 2

H. Nielson and F. Nielson

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

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 Operational Semantics
  - Structural 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
}
```

$[x \mapsto 3]$

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

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

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

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

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

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

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

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

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

$[x \mapsto 3, z \mapsto 6, y \mapsto 2]$

$[x \mapsto 3, z \mapsto 6, y \mapsto 1]$

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

$[x \mapsto 3, z \mapsto 6, y \mapsto 1]$

$[x \mapsto 3, z \mapsto 6, y \mapsto 1]$

$[x \mapsto 3, z \mapsto 6, y \mapsto 1]$

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

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

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

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

```
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;
}
```

$f=\lambda x.$ if $x = 0$ then 1 else $x * f(x -1)$

Axiomatic Semantics

$\{x=n\}$

int fact(int x) { int z, y;

z = 1;

$\{x=n \wedge z=1\}$

y= x

$\{x=n \wedge z=1 \wedge y=n\}$

while

$\{x=n \wedge y \geq 0 \wedge z=n! / y!\}$

(y>0) {

$\{x=n \wedge y > 0 \wedge z=n! / y!\}$

z = z * y ;

$\{x=n \wedge y>0 \wedge z=n!/(y-1)!\}$

y = y – 1;

$\{x=n \wedge y \geq 0 \wedge z=n!/y!\}$

} return z} $\{x=n \wedge z=n!\}$

Operational Semantics

Natural Semantics

Operational Semantics of Arithmetic Expressions

Exp → | number

| Exp PLUS Exp

| Exp MINUS Exp

| Exp MUL Exp

| UMINUS Exp

$A[\![\,]\!]: Exp \rightarrow Z$

$$A[\![n]\!] = val(n)$$

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

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

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

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

Handling Variables

Exp → | number

| variable

| Exp PLUS Exp

| Exp MINUS Exp

| Exp MUL Exp

| UMINUS Exp

- Need the notions of states
- States State = Var $\to$ 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
- $B[\![\text{Bexp}]\!] : \text{State} \rightarrow T$
- Defined by induction on the syntax tree
  - $B[\![\text{true}]\!]\, s = tt$
  - $B[\![\text{false}]\!]\, s = ff$
  - $B[\![e_1 = e_2]\!]\, s =$

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

  - $B[\![e_1 \wedge e_2]\!]\, s =$

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

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

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

Natural Semantics for While

axioms

$[\text{ass}_{ns}]\ <x := a, s> \to s[x \mapsto \mathbf{A}[\![a]\!]s]$

$[\text{skip}_{ns}]\ <\mathbf{skip}, s> \to s$

rules

$$[\text{comp}_{ns}]\ \frac{<S_1, s> \to s', <S_2, s'> \to s''}{<S_1; S_2, s> \to s''}$$

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

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

Natural Semantics for While (More rules)

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

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

A Derivation Tree

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

$$\dfrac{\langle skip, s_0\rangle \to s_0 \;\; (skip_{ns}) \qquad \langle x := x+1, s_0\rangle \to s_0[x \mapsto 1] \;\; (ass_{ns})}{\langle skip;\ x := x+1, s_0\rangle \to s_0[x \mapsto 1]} \; comp_{ns}$$

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}

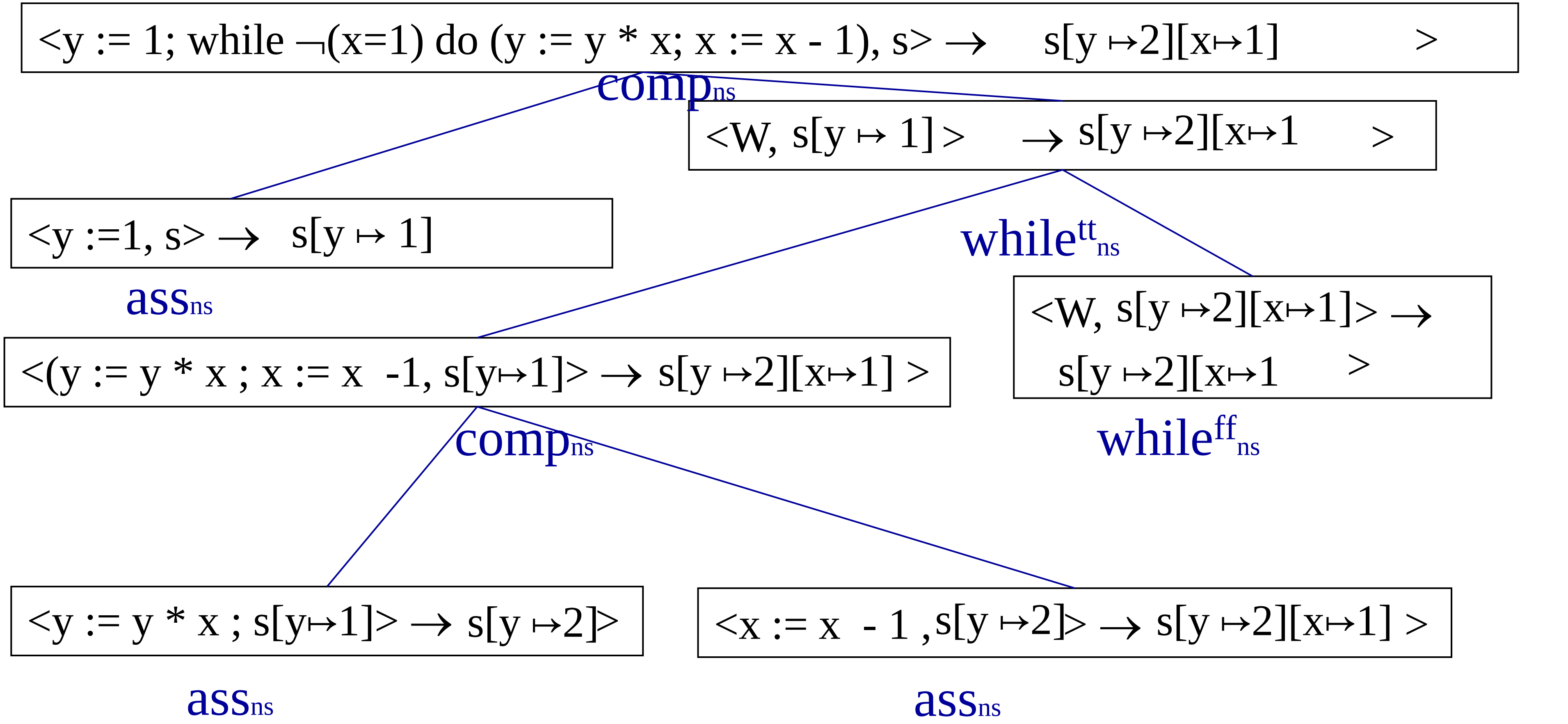
ass_{ns}

Top Down Evaluation of Derivation Trees

- Given a program S and an input state s
- Find an output state s' such that
  $\langle S, s\rangle \to s'$
- 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'
  $\langle S_1, s\rangle \rightarrow s'$ 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"

Deterministic Semantics for While

- 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
  - $S_{ns}\,[\![\text{skip}]\!]s = s$
  - $S_{ns}\,[\![x := 1]\!]s = s\,[x \mapsto 1]$
  - $S_{ns}\,[\![\text{while true do skip}]\!]s =$ undefined

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 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)

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

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

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

The While Programming Language with Non-Determinism 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)

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)

Conclusion

- In the natural semantics immediate constituent is an atomic entity so we cannot express interleaving of computations

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

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