Formal Semantics of Programming Languages

Mooly Sagiv

Reference: Semantics with Applications

Chapter 2

H. Nielson and F. Nielson

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

Benefits of formal definitions

- Intellectual
- Better understanding
- Formal proofs
- Mechanical checks by computer
- Tool generations

What is a good formal definition?

- Natural
- Concise
- Easy to understand
- Permits effective mechanical reasoning

Programming Languages

- Syntax
  - Which string is a legal program?
  - Usually defined using context free grammar+ contextual constraints
- Semantics
  - What does a program mean?
  - What is the output of the program on a given run?
  - When does a runtime error occur?
  - A formal definition

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 in the course

Who need formal semantics for PL?

- Language designers
- Compiler designers
- [Programmers]

Example C++

- Designed with a source to source compiler to C
- Many issues
  - Especially later

Type Safety

- A programming language is type safe if every well typed program has no undefined semantics
- No runtime surprise
- Is C type safe?
- How about Java?

Breaking Safety in C

```
void foo(s) {
    char c[100];
    strcpy(c, s);
}
```

A Pathological C Program

```
a =  malloc(…) ;
b = a;
free (a);
c = malloc (…);
if  (b == c)  printf(“unexpected equality”);
```

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
}
```

$[x \mapsto 3]$

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

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

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

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

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

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

```
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
}
```

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

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

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

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

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

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

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

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

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

Denotational Semantics

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

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

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

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]

Example State Manipulations

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

Semantics of arithmetic expressions

- Assume that arithmetic expressions are side-effect free
- $A[\![\text{Aexp}]\!] : \text{State} \rightarrow N$
- Defined by structural induction on the syntax tree
  - $A[\![n]\!] s = n$
  - $A[\![x]\!] s = s\ x$
  - $A[\![e_1 + e_2]\!] s = A[\![e_1]\!] s + A[\![e_2]\!] s$
  - $A[\![e_1 * e_2]\!] s = A[\![e_1]\!] s * A[\![e_2]\!] s$
  - $A[\![(e_1)]\!] s = A[\![e_1]\!] s$ --- not needed
  - $A[\![- e_1]\!] s = -A[\![e_1]\!] s$

Properties of arithmetic expressions

- The semantics is compositional
  - $A[\![e_1 \text{ op } e_2]\!] = [\![\text{op}]\!](A[\![e_1]\!], A[\![e_2]\!])$
  - Properties can be proved by structural induction
- We say that e_1 is semantically equivalent to e_2 ($e_1 \approx e_2$) when $A[\![e_1]\!] = A[\![e_2]\!]$

Commutativity of expressions

- Theorem: for every expressions e_1, e_2: $e_1 + e_2 \approx e_2 + e_1$
- Proof:
  $\mathcal{A}[\![e_1 + e_2]\!]s = \mathcal{A}[\![e_1]\!]s + \mathcal{A}[\![e_2]\!]s = \mathcal{A}[\![e_2]\!]s + \mathcal{A}[\![e_1]\!]s = \mathcal{A}[\![e_2 + e_1]\!]s$

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
  - $B[\![\text{true}]\!] s = tt$
  - $B[\![\text{false}]\!] s = ff$
  - $B[\![e_1 = e_2]\!] s = \begin{cases} tt \text{ if } A[\![e_1]\!] s = A[\![e_2]\!] s \\ ff \text{ if } A[\![e_1]\!] s \neq A[\![e_2]\!] s \end{cases}$
  - $B[\![e_1 \wedge e_2]\!] s = \begin{cases} tt \text{ if } B[\![e_1]\!] s = tt \text{ and } B[\![e_2]\!] = tt \\ ff \text{ if } B[\![e_1]\!] s = ff \text{ or } B[\![e_2]\!] s = ff \end{cases}$
  - $B[\![e_1 \geq e_2]\!] s =$

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

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
  $\langle S, i\rangle \rightarrow o$
  "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 i is the set of outputs states *o* such that $\langle P, i\rangle \rightarrow o$
- The meaning of compound statements is defined using the meaning immediate constituent statements

Natural Semantics for While

axioms

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

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

rules

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

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

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

Natural Semantics for While (More rules)

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

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

Simple Examples

- Let s_0 be the state which assigns zero to all program variables
- Assignments
  $[ass_{ns}]$ $\langle x := x+1, s_0 \rangle \to s_0[x \mapsto 1]$
- Skip statement
  $[skip_{ns}]$ $\langle skip, s_0 \rangle \to s_0$
- Composition

$$[comp_{ns}] \ \frac{\langle skip, s_0 \rangle \to s_0, \ \langle x := x+1, s_0 \rangle \to s_0[x \mapsto 1]}{\langle skip;\ x := x+1, s_0 \rangle \to s_0[x \mapsto 1]}$$

Simple Examples (Cont)

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

$$[\text{if}^{tt}_{ns}] \quad \frac{\langle \text{skip}, s_0\rangle \to s_0}{\langle \text{if } x=0 \text{ then skip else } x := x+1,\ s_0\rangle \to s_0}$$

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

$$\dfrac{\langle \text{skip}, s_0\rangle \to s_0 \qquad \langle x := x + 1, s_0\rangle \to s_0[x \mapsto 1]>}{\langle \text{skip}; x := x + 1, s_0\rangle \to s_0[x \mapsto 1]>}\ \text{comp}_{\text{ns}}$$

An Example Derivation Tree

<(x :=x+1; y :=x+1) ; z := y), s0> →s0[x ↦1][y ↦2][z ↦2]

comp$_{ns}$

<x :=x+1; y :=x+1, s0> →s0[x ↦1][y ↦2]

<z :=y,s0[x ↦1][y ↦2]>→s0[x↦1][y↦2][z ↦2]

ass$_{ns}$

comp$_{ns}$

ass$_{ns}$ <x :=x+1; s0> →s0[x ↦1]

ass$_{ns}$ <y :=x+1, s0[x ↦1]> →s0[x ↦1][y ↦2]

Top Down Evaluation of Derivation Trees

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

$\langle y := 1;\ \text{while} \neg(x=1)\ \text{do}\ (y := y * x;\ x := x - 1), s\rangle \rightarrow\ s[y \mapsto 2][x \mapsto 1]$

comp_{ns}

$\langle y :=1, s\rangle \rightarrow\ s[y \mapsto 1]$

ass_{ns}

$\langle W, s[y \mapsto 1]\rangle \rightarrow s[y \mapsto 2][x \mapsto 1$

while^{tt}_{ns}

$\langle (y := y * x\ ;\ x := x\ -1, s[y \mapsto 1]\rangle \rightarrow s[y \mapsto 2][x \mapsto 1]\ \rangle$

comp_{ns}

$\langle y := y * x\ ;\ s[y \mapsto 1]\rangle \rightarrow s[y \mapsto 2]\rangle$

ass_{ns}

$\langle x := x\ -1\ , s[y \mapsto 2]\rangle \rightarrow s[y \mapsto 2][x \mapsto 1]$

ass_{ns}

$\langle W, \rightarrow\ s[y \mapsto 2][x \mapsto 1]$
$s[y \mapsto 2][x \mapsto 1$

while^{ff}_{ns}

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

Example Programs

- skip
- If true then skip else C
- while false do C
- while true do skip

Semantic Equivalence

- S_1 and S_2 are semantically equivalent if for all s and s' $(s \approx s')$
  $\langle S_1, s\rangle \rightarrow s'$ if and only if $\langle S_2, s\rangle \rightarrow s'$

Example of Semantic Equivalence

- skip ; skip $\approx$ skip
- $(S_1 ; S_2) ; S_3) \approx (S_1 ;(S_2 ; S_3))$
- x := 5 ; y := x * 8 $\approx$ x :=5; y := 40
- while b do S $\approx$
  if b then (S ; while b do S) else skip

Deterministic Semantics for While

- If <S, s> $\to s_1$ and <S, s> $\to s_2$ 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}: $\mathbf{Stm} \rightarrow (\mathbf{State} \hookrightarrow \mathbf{State})$
- $S_{ns} [\![S]\!]s = s'$ if $<S, s> \rightarrow s'$ and otherwise
  $S_{ns} [\![S]\!]s$ is undefined
- Examples
  - $S_{ns} [\![\text{skip}]\!]s = s$
  - $S_{ns} [\![x := 1]\!]s = s\,[x \mapsto 1]$
  - $S_{ns} [\![\text{while true do skip}]\!]s = \text{undefined}$

Summary Natural Semantics

- Simple
- Useful
- Enables simple proofs of language properties
- Automatic generation of interpreters
- But limited
  - Concurrency