CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Last class

A Big step semantic

B Calling convention

C Small step semantics

Outline

- Operational semantics talks about how an expression is evaluated.
- Denotational semantics
    - Describes what a program text means in mathematical terms - constructs mathematical objects.
    - is compositional - denotation of a command is based on the denotation of its immediate sub-commands.
    - Also called: fixed-point semantics, mathematical semantics, Scott-Strachey semantics.

Operational semantics: good as specification for a compiler / interpreter.
Denotational semantics: proving equivalence of programs: equivalent programs have equal denotational models.

Denotational semantics: idea

- Assigns meanings to programs.
- $\perp$ is used to mean non-termination.
- Instance of mathematical objects:
  - A number $\in Z$
  - A boolean $\in$ {`true, false`}.
  - A state transformer: $\Sigma \to (\Sigma \cup \{\perp\})$
- Think ahead: Semantics of a loop.

Notation

- $[\![e_1]\!]$ - "means" or "denotes".
- Σ set of states. $\sigma \in \Sigma$ denotes a state.
- The meaning of an arithmetic expression e in state σ is a number. $A[\![.]\!] : Aexp \to (\Sigma \to Z)$
- The meaning of an boolean expression e in state σ is a truth value. $A[\![.]\!] : Aexp \to (\Sigma \to \{true, false\})$
- Denotational functions are *total* - defined for all (well typed) syntactic elements.
- Finds mathematical objects (called domains) that represent what programs do.

Denotational semantics of arithmetic expressions

- Inductively define $A[\![.]\!] : Aexp \to (\Sigma \to Z)$
- $$\begin{array}{lcl} A[\![n]\!]\sigma & = & \lceil n \rceil \\ A[\![x]\!]\sigma & = & \sigma(n) \\ A[\![e_1 + e_2]\!]\sigma & = & A[\![e_1]\!]\sigma + A[\![e_2]\!]\sigma \\ A[\![e_1 - e_2]\!]\sigma & = & A[\![e_1]\!]\sigma - A[\![e_2]\!]\sigma \end{array}$$

Assignment: Write denotational semantics for boolean expressions.

Denotational semantics for commands I

- Running a command c starting from a state σ yields a state σ'
- Define $C[\![c]\!]$: $C[\![.]\!] : Com \to (\Sigma \to \Sigma)$
- Q: What about non termination?
- Recall $\perp$ denotes the state of non-termination.
- Notation: $X_\perp = X \cup \{\perp\}$.
- Convention: whenever $f \in X \to X_\perp$, we extend f with $f(\perp) = \perp$ so that $f \in X_\perp \to X_\perp$. – called *strictness*

Denotational semantics for commands II

- $C[\![.]\!] : Com \to (\Sigma \to \Sigma_\perp)$
  $C[\![skip]\!]\sigma \quad = \quad \sigma$
  $C[\![x := e]\!]\sigma \quad = \quad \sigma[x := A[\![e]\!]\sigma]$
  $C[\![c_1; c_2]\!]\sigma \quad = \quad C[\![c_2]\!](C[\![c_1]\!]\sigma)$
  $C[\![$if $\texttt{b}$ then c_1 else $c_2]\!]\sigma$ =
  if $B[\![\texttt{b}]\!]$ then $C[\![c_1]\!]\sigma$ else $C[\![c_2]\!]\sigma$

Associativity of Addition

- **Theorem**: For all E_1, E_2 and E_3: $[\![E_1 + (E_2 + E_3)]\!] = [\![(E_1 + E_2) + E_3]\!]$
- **Proof**

$$
\begin{aligned}
[\![E_1 + (E_2 + E_3)]\!] &= [\![E_1]\!] + [\![(E_2 + E_3)]\!] \\
&= [\![E_1]\!] + ([\![E_2]\!] + [\![E_3]\!]) \\
&= ([\![E_1]\!] + [\![E_2]\!]) + [\![E_3]\!] \\
&= [\![(E_1 + E_2)]\!] + [\![E_3]\!] \\
&= [\![(E_1 + E_2) + E_3]\!]
\end{aligned}
$$

Handle a loop

- Similar to operational semantics?
- $C[\![$while $\texttt{b}$ do $\texttt{c}]\!]\sigma$ = ?
- Notation: $W = C[\![$while $\texttt{b}$ do $\texttt{c}]\!]$
- while $\texttt{b}$ do $\texttt{c}$ = if $\texttt{b}$ then c; while $\texttt{b}$ do $\texttt{c}$ else skip
- $W(\sigma)$ = if $B[\![\texttt{b}]\!])\sigma$ then $W(C[\![c]\!]\sigma)$ else σ
  - Recursive definition - or no definition?
  - Not compositional
- Say $C[\![$while $\texttt{true}$ do $\texttt{skip}]\!]$
  $W(\sigma) = W(\sigma)$ – does not help.
- Say $C[\![$while $x \neq 0$ do $x = x - 2]\!]$
  $$W(\sigma) = \begin{cases} \sigma[x := 0] & \text{if } \sigma(x) \text{ even and } \sigma(x) \geq 0 \\ \sigma' & \text{otherwise.} \end{cases}$$
  for any σ'.

while k-steps semantics

- Define $W_k : \Sigma \to \Sigma_\perp$ (for $k \in N$) such that:
  $$W_k(\sigma) = \begin{cases} \sigma' & \text{if "while b do c" in state } \sigma \text{ terminates in fewer than } k \text{ iterations in state } \sigma' \\ \perp & \text{otherwise.} \end{cases}$$
- $W_0(\sigma) = \perp$
- $$W_k(\sigma) = \begin{cases} W_{k-1}(C[\![c]\!]\sigma) & \text{if } B[\![b]\!]\sigma \text{ for } k \geq 1 \\ \sigma & \text{otherwise.} \end{cases}$$

while semantics defined

- How do we get W from W_k?

$$W(\sigma) = \begin{cases} \sigma' & \text{smallest } k \text{ such that } W_k(\sigma) = \sigma' \neq \perp \\ \perp & \text{otherwise (that is, } \forall k, W_k(\sigma) = \perp). \end{cases}$$

- It is compositional.
- Has a bit of operational flavour :-(
- How to generalize it to higher order functions?

Old loops revisited:

- while `true` do `skip`; — $W_k(\sigma) = \perp$, for all k. Thus $W(\sigma) = \perp$.
- while $x \neq 0$ do x = x - 2; —

$$W(\sigma) = \begin{cases} \sigma[x := 0] & \text{if } \sigma(x) = 2 * m \text{ AND } \sigma(x) \geq 0 \\ \perp & \text{otherwise.} \end{cases}$$

Properties of while-loop

- Prove that “if $C[\![$while `b` do `c`$]\!]\sigma = \sigma'$ then $B[\![B]\!]\sigma' = \texttt{false}$.
- For any natural number n and any state σ if $W_n(\sigma) = \sigma' \neq \perp$, then $B[\![b]\!] = \texttt{false}$.

Axiomatic semantics

- Operational semantics talks about how an expression is evaluated.
- Denotational semantics - describes what a program text means in mathematical terms - constructs mathematical objects.
- Axiomatic semantics - describes the meaning of programs in terms of properties (axioms) about them.
- Usually consists of
  - A language for making assertions about programs.
  - Rules for establishing when assertions hold for different programming constructs.

Language for Assertions

- A specification language
  - Must be easy to use and expressive
  - Must have syntax and semantics.
- Requirements:
  - Assertions that characterize the state of execution.
  - Refer to variables, memory
- Examples of non state based assertions:
  - Variable x is live,
  - Lock L will be released.
  - No dependence between the values of x and y.

Assertion Language

- Specification language in first-order predicate logic
  - Terms (variables, constants, arithmetic operations)
  - Formulas:
    - `true` and `false`
    - If t_1 and t_2 are terms then, $t_1 = t_2$, $t_1 < t_2$ are formulas.
    - If ϕ is a formula, so is $\neg\phi$.
    - IF ϕ_1 and ϕ_2 are two formulas then so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ and $\phi_1 \Rightarrow \phi_2$.
    - If $\phi(x)$ is a formula (with a free variable x) then, $\forall x.\phi(x)$ and $\exists x.\phi(x)$ are formulas.

Hoare Triples

- Meaning of a statement S can be described in terms of triples:
  $\{P\}S\{Q\}$
  where
- P and Q are formulas or assertions.
  - P is **a** pre-condition on S
  - Q is **a** post-condition on S.
- The triple is *valid* if
  - execution of S begins in a state satisfying P.
  - S terminates.
  - resulting state satisfies Q.

Satisfiability

- A formula in first-order logic can be used to characterize states.
  - The formula $x = 3$ characterizes all program states in which the value of the location associated with x is 3.
  - Formulas can be thought as assertions about states.
- Define $\{\sigma \in \Sigma | \sigma \models \phi\}$, where $\models$ is a satisfiability relation.
  - Let the value of a term t in state σ be t^σ
    - If t is a variable x then $t^\sigma = \sigma(x)$.
    - If t is an integer n then $t^\sigma = n$.
    - $\sigma \models t_1 = t_2$ if $t_1^\sigma = t_2^\sigma$
    - $\sigma \models t_1 \wedge t_2$ if $\sigma \models t_1$ and $\sigma \models t_2$
    - $\sigma \models \forall x.\phi(x)$ if $\sigma[x \mapsto n] \models \phi(n)$ for all integer constants n.
    - $\sigma \models \exists x.\phi(x)$ if $\sigma[x \mapsto n] \models \phi(n)$ for some integer constant n.

Examples

- $\{2 = 2\}x := 2\{x = 2\}$
  An assignment operation of x to 2 results in a state in which x is 2, assuming equality of integers!
- $\{\texttt{true}\}$ `if` B then $x := 2$ `else` $x := 1$ $\{x = 1 \vee x = 2\}$
  A conditional expression that either assigns x to 1 or 2, if executed will lead to a state in which x is either 1 or 2.
- $\{2 = 2\}x := 2\{y = 1\}$
- $\{\texttt{true}\}$ `if` B then $x := 2$ `else` $x := 1$ $\{x = 1 \wedge x = 2\}$
  Why are these invalid?

Partial Correctness

- The validity of a Hoare triple depends upon the termination of the statement S
- $\{0 \leq a \wedge 0 \leq b\}\ S\ \{z = a \times b\}$
  - If executed in a state in which $0 \leq a$ and $0 \leq b$, and
  - S terminates,
  - then $z = a \times b$.

Soundness

- Hoare rules can be seen as a proof system.
  - Derivations are proofs.
  - conclusions are theorems.
  - We write $\vdash$ {P} c {Q}, if {P} c {Q} is a theorem.
- If $\vdash$ {P} c {Q}, then $\models$ {P} c {Q}.
  - Any derivable assertion is *sound* with respect to the underlying semantics.

Proof rules

- Skip:
  $$\{P\} skip \{P\}$$
- Assignment:
  $$\{P[t/x]\} x := t \{P\}$$
  Example: Suppose $t = x + 1$
  then, $\{x+1 = 2\} x := x+1 \{x = 2\}$
- $$[Sequencing] \{P_1\} c_0 \{P_2\}\ \{P_2\} c_1 \{P_3\} \{P_1\} c_0; c_1 \{P_3\}$$
- $$[Conditionals] \{P_1 \wedge b\} c_0 \{P_2\}\ \{P_1 \wedge \neg b\} c_1 \{P_2\} \{P_1\} \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1 \{P_2\}$$

while n ¿ 0 do z := z + x; n := n - 1;

```
P = {z = x*(y-n) ∧ n ≥ 0}
```

(apply the consequence rule)

{z = x * (y-n) ∧ n ≥ 0}
while n > 0 do z := z+x; n := n-1
{z = x * (y-n) ∧ n ≥ 0 ∧ ¬ (n > 0) }

```
(any iteration)
{(z+x) = x * (y-(n-1)) ∧ (n-1) ≥ 0}
z := z+x;
{z=x*(y-(n-1)) ∧ (n-1) ≥ 0}
n := n-1
{z=x*(y-n) ∧ n ≥ 0}

z = x*(y-n) ∧ n ≥ 0 ∧ n > 0 ⇒
        {(z+x) = x * (y-(n-1)) ∧ (n-1) ≥ 0}

(consequence)
{z = x*(y-n) ∧ n ≥ 0 ∧ n > 0}
```

```
z := z+x; n := n-1
{z=x*(y-n) ∧ n ≥ 0}
```


Step II - constructing the proof in reverse order

```
(pre-loop code)
{z = x*(y-y) ∧ y ≥ 0}
n := y
{z = x*(y-n) ∧ n ≥ 0}

{0 = x*(y-y) ∧ y ≥ 0}
z := 0
{z = x*(y-y) ∧ y ≥ 0}


{y ≥ 0}
z := 0; n := y
{z = x*(y-n) ∧ n ≥ 0}
{y ≥ 0} above-program {z = x * y}
```


Useless assignment

```
while (x != y) do
if (x <= y)
then
y := y-x
else
x := x-y
```

Derive that
$\vdash$ `{x = m ∧ y = n} above-program {x = gcd(m, n)}`

Hint: Start with the loop invariant to be `{gcd(x, y) = gcd(m, n)}`

Last Class

- Axiomatic Semantics
- Proof rules
- Proving the semantics of the multiplication routine.

Equivalence of Denotational and Operational semantics

- Statement: $\sigma \triangleright e \vdash n$ iff $A[\![e]\!]\sigma = n$
  $\sigma \triangleright e \vdash t$ iff $B[\![e]\!]\sigma = t$
  $\sigma \triangleright c \vdash \sigma'$ iff $C[\![c]\!]\sigma = \sigma' \neq \perp$
- Arithmetic and boolean expressions - straight forward.
- We will study commands.

Equivalence proof - if (I)

IF: If we have a derivation $\sigma \triangleright c \vdash \langle v, \sigma' \rangle$ then $C[\![c]\!]\sigma = \sigma'$.

proof
(By induction on the structure of the derivation (let us call it D).)
Say, the last rule in the derivation D is a while-loop.
(other cases are easier and left for self study).

We will reuse the old notation

- $C[\![\texttt{while } b \texttt{ do } c]\!] = W$.

To prove that $W(\sigma) = \sigma'$.

Equivalence proof -if (II)

Case: Given- we have a derivation $\sigma \triangleright c \vdash \sigma'$ and the last rule is a while-false.

$$[D ::]D_1 :: \sigma \triangleright b \vdash \langle false, \sigma \rangle \sigma \triangleright \texttt{while } b \texttt{ do } c \vdash \sigma$$

- σ' must be σ
- From D_1 and using the equivalence for booleans we have that $B[\![b]\!] = false$.

  $W_1(\sigma) = \sigma$

  Therefor $W(\sigma) = \sigma$.

$$[D ::]$$

$$[D ::]\forall P, Q$$

body end the program either does not terminate or it terminates in a state that satisfies Q.

$\forall \sigma, P, Q, c \models \{P\}c\{Q\}$
if
$\forall \sigma'$:
 $\sigma \triangleright P \vdash \langle true, \sigma \rangle \wedge$
 $\sigma \triangleright c \vdash \sigma'$
then
$\sigma' \triangleright Q \vdash \langle true, \sigma' \rangle$

Validity

Validity via total correctness

- $[P]c[Q]$: Whenever we start the execution of command c in a state that satisfies P, the program terminates in a state that satisfies Q.
- $\forall \sigma, P, Q, c \ \models [P]c[Q]$
  if $\sigma \triangleright P \vdash \langle true, \sigma \rangle$
  then
  $\exists \sigma'$:
  $\sigma \triangleright c \vdash \sigma' \wedge$
  $\sigma' \triangleright Q \vdash \langle true, \sigma' \rangle$

Soundness

- All derived triples are valid.
- If $\vdash$ {P} c {Q}, then $\models$ {P} c {Q}.
- Any derivable assertion is *sound* with respect to the underlying operational semantics.

Completeness

- All derived triples are derivable from empty set of assumptions.
- If $\models$ {P} c {Q}, then
  $\exists \sigma'$
  $\textit{init-state} \ \triangleright \{P\}c\{Q\} \vdash \langle true, \sigma' \rangle$.

Acknowledgements

- Suresh Jagannathan
- George Necula
- Internet.