Preliminary Concepts (3)

Operational Semantics, Interpreters

Woosuk Lee

CSE 6049 Program Analysis

Hanyang University, Korea

Two Styles of Definitions of Semantics

- **Denotational semantics**: The meaning is modeled by mathematical objects that represent the effect of executing the program. About the result (not about how the result is obtained)
  - So-called *compositional* style
- **Operational semantics**: The meaning is specifed by the computation steps executed on a machine. About how the result is obtained
  - So-called *transitional* style

Operational Semantics

- concerning how to execute programs and not merely what the execution results are.
    - **Big-step** operational semantics describes how overall results of executions are obtained.
    - **Small-step** operational semantics describes how individual steps of computations take place.
- Inductively defined, thus may not be compositional

Semantic Domains in Operational Semantics

- Ordinary sets; no need to be CPOs
- $S \cup T, S + T, S \times T, S \xrightarrow{\text{fin}} T$

$$S \xrightarrow{\text{fin}} T = \{f | f \in S' \to T, S' \overset{\text{fin}}{\subseteq} S\}$$

- Not to confuse $\xrightarrow{\text{fin}}$ with $\to$

The WHILE Language

$$
\begin{array}{rcl}
C & \rightarrow & \texttt{skip} \\
& | & x := E \\
& | & \texttt{if}\ E\ C\ C \\
& | & C; C \\
& | & \texttt{while}\ E\ C
\end{array}
$$

$$
\begin{array}{rcll}
E & \rightarrow & n & (n \in \mathbb{Z}) \\
& | & x & \\
& | & E \texttt{ + } E & \\
& | & \texttt{-}\ E &
\end{array}
$$

Semantics as Proofs

- Semantic domain

$$\begin{array}{rcrcl} M & \in & Memory & = & Var \xrightarrow{\text{fin}} Val \\ v & \in & Val & = & \mathbb{Z} \end{array}$$

- Program semantics: proofs using a set of inference rules
  - $M \vdash C \Rightarrow M'$: Execution of C with memory M will result in another memory M'
  - $M \vdash e \Rightarrow v$: Execution of E given memory M will result in v

Big-step Operational Semantics

$$\frac{}{M \vdash \texttt{skip} \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow v}{M \vdash x \; \texttt{:=} \; E \Rightarrow M\{x \mapsto v\}}$$

$$\frac{M \vdash C_1 \Rightarrow M_1 \quad M_1 \vdash C_2 \Rightarrow M_2}{M \vdash C_1 \; ; \; C_2 \Rightarrow M_2}$$

Big-step Operational Semantics

$$\frac{M \vdash E \Rightarrow 0 \quad M \vdash C_2 \Rightarrow M'}{M \vdash \texttt{if}\ E\ C_1\ C_2 \Rightarrow M'}$$

$$\frac{M \vdash E \Rightarrow v \quad M \vdash C_1 \Rightarrow M'}{M \vdash \texttt{if}\ E\ C_1\ C_2 \Rightarrow M'}\ v \neq 0$$

$$\frac{M \vdash E \Rightarrow 0}{M \vdash \texttt{while}\ E\ C \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow v \quad M \vdash C \Rightarrow M_1 \quad M_1 \vdash \texttt{while}\ E\ C \Rightarrow M_2}{M \vdash \texttt{while}\ E\ C \Rightarrow M_2}\ v \neq 0$$

Big-step Operational Semantics

$$\frac{}{M \vdash n \Rightarrow n}$$

$$\frac{}{M \vdash x \Rightarrow M(x)}$$

$$\frac{M \vdash E_1 \Rightarrow v_1 \quad M \vdash E_2 \Rightarrow v_2}{M \vdash E_1 \texttt{+} E_2 \Rightarrow v_1 + v_2}$$

$$\frac{M \vdash E \Rightarrow v}{M \vdash \texttt{-} E \Rightarrow -v}$$

Semantics as Proofs

More precise interpretation of the evaluation rules:

- The inference rules define a set S of triples (M, e, v). For readability, the triple was written by $M \vdash e \Rightarrow v$ in the rules.

- We say an expression e has semantics w.r.t. M iff there is a triple $(M,e,v) \in S$ for some value v.

- That is, we say an expression e has semantics w.r.t. M iff we can derive $M \vdash e \Rightarrow v$ for some value v by applying the inference rules.

- We say an initial expression e has semantics if $\{\} \vdash e \Rightarrow v$ for some v.

Example

$$C \stackrel{\text{let}}{=} x \texttt{ := 1 ; } y \texttt{ := } x \texttt{ + 1}$$

$$\frac{}{\emptyset \vdash C \Rightarrow \{x \mapsto 1, y \mapsto 2\}}$$

Example

$$C \stackrel{\mathsf{let}}{=} x \ \texttt{:=}\ \texttt{1}\ \texttt{;}\ y\ \texttt{:=}\ x\ \texttt{+}\ \texttt{1}$$

$$\dfrac{\dfrac{\emptyset \vdash \texttt{1} \Rightarrow 1}{\emptyset \vdash x\ \texttt{:=}\ \texttt{1} \Rightarrow \{x \mapsto 1\}} \qquad \dfrac{\dfrac{\{x \mapsto 1\} \vdash x \Rightarrow 1 \qquad \{x \mapsto 1\} \vdash \texttt{1} \Rightarrow 1}{\{x \mapsto 1\} \vdash x\ \texttt{+}\ \texttt{1} \Rightarrow 2}}{\{x \mapsto 1\} \vdash y\ \texttt{:=}\ x\ \texttt{+}\ \texttt{1} \Rightarrow \{x \mapsto 1, y \mapsto 2\}}}{\emptyset \vdash C \Rightarrow \{x \mapsto 1, y \mapsto 2\}}$$

Exercise

$$\frac{}{\{\} \vdash x := 1; \texttt{if}\ (x)\ y := 1\ y := -1 \Rightarrow ?}$$

Exercise

$$\frac{}{\{\} \vdash x := 2; \texttt{while}\ (x)\ x := x + (-1) \Rightarrow ?}$$

Execution Types

- We say the execution of a command C on a memory M
  - Terminates iff there is a memory M' such that $M \vdash C \Rightarrow M'$
  - Loops otherwise

Examples

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$\{\} \vdash x := 1; \texttt{while}\ (x)\ x := x + 1 \Rightarrow ?$$

Semantic Equivalence

- We say C_1 and C_2 are semantically equivalent (denoted $C_1 \equiv C_2$) if the following is true for all memories M, M'

$$M \vdash C_1 \Rightarrow M' \iff M \vdash C_2 \Rightarrow M'$$

- Example:
  - $\texttt{while}\ x\ C \equiv \texttt{if}\ (x)\ (C; \texttt{while}\ x\ C)\ \texttt{skip}$

Implementing Big-step Interpreter in OCaml

```
type var = string

type exp =
   | Int of int  (* n *)
   | Var of var  (* x *)
   | Plus of exp * exp  (* e1 + e2 *)
   | Minus of exp  (* -e *)

type cmd =
   | Assign of var * exp    (* x := e *)
   | Skip    (* skip *)
   | Seq of cmd * cmd      (* c1; c2 *)
   | If of exp * cmd * cmd       (* if e c1 c2 *)
   | While of exp * cmd      (* while e c *)

(* x := 10; y := 1; while (x) (y := y + y; x := x - 1*)
let pgm =
  Seq (Assign ("x", Int 10),
    Seq (Assign ("y", Int 1),
    While (Var "x",
      Seq (Assign("y", Plus (Var "y", Var "y")),
           Assign("x", Plus (Var "x", Minus (Int 1))))
       )))
```

Implementing Big-step Interpreter in OCaml

```
module Mem = struct
  type t = (var * int) list
  let empty = []
  let rec lookup m x =
      match m with
      | [] -> raise (Failure (x ^ "is not bound in state"))
      | (y,v) :: m' -> if x = y then v else lookup m' x
  let update m x v = (x,v)::m
end

let rec eval_e : exp -> Mem.t -> int
= fun e m ->
   match e with
    | Int n -> n
    | Var x -> Mem.lookup m x
    | Plus (e1, e2) -> (eval_e e1 m) + (eval_e e2 m)
    | Minus e' -> -1 * (eval_e e' m)
```

Implementing Big-step Interpreter in OCaml

```
let rec eval_c : cmd -> Mem.t -> Mem.t
= fun c m ->
  match c with
  | Assign (x, e) -> Mem.update m x (eval_e e m)
  | Skip -> m
  | Seq (c1, c2) -> eval_c c2 (eval_c c1 m)
  | If (e, c1, c2) ->
    eval_c (if (eval_e e m) <> 0 then c1 else c2) m
  | While (e, c) ->
    if (eval_e e m) <> 0 then
      eval_c (While (e,c)) (eval_c c m)
    else m

let _ =
  print_int (Mem.lookup (eval_c pgm Mem.empty) "y");
  print_newline ()
```

Small-step Operational Semantics

- Another alternative is to define semantics as a transition system
    - $\mathcal{S}$: the set of states
    - $(\rightarrow) \subseteq \mathcal{S} \times \mathcal{S}$: transition relation
- In our case, a state is a pair of a command and a memory $\langle C, M \rangle$

$$\langle C, m \rangle \rightarrow \langle C', m' \rangle$$

"Execution of C from m will result in C' and m'."

Small-step Operational Semantics

- Semantics of expressions is defined as a function:

$$[\![E]\!] : Memory \rightarrow Val$$

$$\begin{array}{rcl} [\![n]\!](M) & = & n \\ [\![x]\!](M) & = & M(x) \\ [\![E_1{+}E_2]\!](M) & = & [\![E_1]\!](M) + [\![E_2]\!](M) \\ [\![-E]\!](M) & = & -[\![E]\!](M) \end{array}$$

Small-step Operational Semantics

$$\frac{\langle C_1, m\rangle \to \langle C_1', m'\rangle}{\langle C_1;\, C_2, m\rangle \to \langle C_1';\, C_2, m'\rangle}$$

$$\frac{}{\langle \texttt{skip};\, C_2, m\rangle \to \langle C_2, m\rangle}$$

$$\frac{[\![E]\!](m) = n}{\langle x := E, m\rangle \to \langle \texttt{skip}, m\{x \mapsto n\}\rangle}$$

$$\frac{[\![E]\!](M) \neq 0}{\langle \texttt{if}\ E\ C_1\ C_2, M\rangle \to \langle C_1, M\rangle}$$

$$\frac{[\![E]\!](M) = 0}{\langle \texttt{if}\ E\ C_1\ C_2, M\rangle \to \langle C_2, M\rangle}$$

$$\frac{}{\langle \texttt{while}\ B\ C, m\rangle \to \langle \texttt{if}\ B\ \texttt{then}\ (C;\ \texttt{while}\ B\ C)\ \texttt{else}\ \texttt{skip}, m\rangle}$$

Exercise

x := 1 ; y := x + 1

Exercise

$$x := 1; \texttt{if}\ (x)\ y := 1\ y := -1$$

Exercise

$$x := 2; \texttt{while}\ (x)\ x := x + (-1)$$

Implementing Small-Step Interpreter in OCaml

```
type conf =
    | NonTerminated of cmd * Mem.t
    | Terminated of Mem.t

let rec eval_e : exp -> Mem.t -> int
= fun e m ->
    match e with
    | Int n -> n
    | Var x -> Mem.lookup m x
    | Plus (e1, e2) -> (eval_e e1 m) + (eval_e e2 m)
    | Minus e' -> -1 * (eval_e e' m)

let rec next : conf -> conf
= fun conf ->
  match conf with
    | Terminated _ -> raise (Failure "impossible")
    | NonTerminated (c, s) ->
      (match c with
       | Assign (x, e) -> Terminated (Mem.update s x (eval_e e s))
       | Skip -> Terminated s
       | Seq (c1, c2) -> (
          match (next (NonTerminated (c1,s))) with
```

Implementing Small-Step Interpreter in OCaml

```
          | NonTerminated (c', s') -> NonTerminated (Seq (c', c2), s')
          | Terminated s' -> NonTerminated (c2, s')
      )
      | If (e, c1, c2) ->
          if (eval_e e s) <> 0 then NonTerminated (c1, s)
          else NonTerminated (c2, s)
      | While (e, c) ->
          NonTerminated (If (e, Seq (c, While (e, c)), Skip), s)
      )

let rec next_trans : conf -> Mem.t
= fun conf ->
    match conf with
    | Terminated s -> s
    | _ -> next_trans (next conf)

let _ =
    print_int (Mem.lookup (next_trans (NonTerminated (pgm,Mem.empty))) "y");
    print_newline ()
```