Semantic Interpretation
Contents

8.1 State Transformations and Partial Functions
8.2 Semantics of Clite
8.3 Semantics with Dynamic Typing
8.4 A Formal Treatment of Semantics
Semantics of a PL

Defines the meaning of a program

- *Syntactically valid*
- *Static type checking valid*
Historical Problem

Valid program had different meanings on different machines

- More than (e.g.) size of an int or float

Problem was lack of precision in defining meaning
Methods

Compiler C on Machine M
  – *Ex: Fortran on IBM 709/7090*
  – *Ex: PL/1 (F) on IBM 360 series*

Operational Semantics – Ch. 7
Axiomatic Semantics – Ch. 18
Denotational Semantics – Ch. 8.4
Example

Environment

- \( i, j \) at memory locations 154, 155

\[ \{ <i, 154>, <j, 155> \} \]

State

- \( i \) has value 13, \( j \) has value -1

\[ \{ ..., <154, 13>, <155, -1>, ... \} \]
Simple State

Ignore environment
Set of identifier – value pairs
Ex: \{<i, 13>, <j, -1>\}

Special value *undefined* for each type
8.1 State Transformations

**Defn:** The *denotational semantics* of a language defines the meanings of abstract language elements as a collection of state-transforming functions.

**Defn:** A *semantic domain* is a set of values whose properties and operations are independently well-understood and upon which the rules that define the semantics of a language can be based.
Meaningless Program

for (i = 1; i > -1; i++)
    i--;

Meaningless Expression

Are all expressions meaningful?
C++Lite Semantics

*State* – a mapping from program variables to values

A *meaning* function $M$ is a mapping:

\[ M: \text{Program} \rightarrow \text{State} \]

\[ M: \text{Statement} \times \text{State} \rightarrow \text{State} \]

\[ M: \text{Expression} \times \text{State} \rightarrow \text{Value} \]
C++Lite Semantics in OCaml

type var = string

type value = I of int | B of bool

module type StateSig = sig

  type state

  val empty: state

  val update: state → var → value → state

  val get_value: state → var → value

end
Meaning Rule 8.1

The meaning of a *Program* is defined to be the meaning of the *body* when given an initial state consisting of the variables of the *decpart* initialized to the *undef* value corresponding to the variable's type.

```plaintext
let p_meaning (decl_list, stat_list) =
   s_meaning (Block stat_list) State.empty
```
Statements

s\_meaning: statement → state → state

type statement =

  Skip                       (* ; *)
  | Block of statement list   (* {…} *)
  | Assign of string * expr   (* x = a *)
  | Cond of expr * statement * statement (* if b s1 else s2 *)
  | Loop of expr * statement   (* while (b) s *)
Meaning Rule 8.2 - Skip

The meaning of a *Skip* is the identity function on the state; that is, the state is unchanged.

```plaintext
let rec s_meaning (s:statement) (q:state) =
  match s with
  | Skip → q
  | ...
Meaning Rule 8.3 - Assignment

The output state is computed from the input state by replacing the value of the target variable by the computed value of the source expression.

```latex
let rec s\_meaning (s:statement) (q:state) =
    match s with ...
    | Assign (x, e) -> let v = e\_meaning e q in
        State.update q x v
    | ...
```
Meaning Rule 8.4 - Conditional

The meaning of a conditional is:

- *If the meaning of the test is true, the meaning of the then-branch*;

- *Otherwise, the meaning of the else-branch*
Meaning Rule 8.4 - Conditional

Let rec s_meaning (s:statement) (q:state) =
    match s with ...
    | Cond (e, s1, s2) ->
        (match (e_meaning e q) with
            B true -> s_meaning s1 q
            | ... )
    | ...
    | ...
Meaning Rule - Loop

let rec s_meaning (s:statement) (q:state) =
    match s with ...
    | Loop (e, s’) ->
        (match (e_meaning e q) with
            B false -> q
            | B true -> s_meaning (Block [s’; s]) q
            | _ -> failwith "Impossible case" )
    | ...

Meaning Rule - Block

let rec s_meaning (s:statement) (q:state) =
match s with ...
    | Block [] -> q
    | Block s':::t ->
        let q' = s_meaning s' q in
        s_meaning (Block t) q'
Expression Semantics

**Defn:** A *side effect* occurs during the evaluation of an expression if, in addition to returning a value, the expression alters the state of the program.

Ignore for now.
Expressions

e\_meaning: expr → state → value

type expr =

    IntC of int

    | BoolC of bool

    | Var of var

    | Unary of unaryOp * expr

    | Binary of binaryOp * expr * expr
Meaning Rule 8.7

The meaning of an expression in a state is a value defined by:

1. *If a value, then the value. Ex: 3*
2. *If a variable, then the value of the variable in the state.*
3. *If a Binary:*
   a) Determine meaning of term1, term2 in the state.
   b) Apply the operator according to rule 8.8

...
Meaning Rules - Expressions

let rec e_meaning (e:expr) (q:state) =
  match e with
  IntC n → I n
  | BoolC b → B b
  | Var x → State.get_value q x
  | ...

Meaning Rules - Expressions

let rec e_meaning (e:expr) (q:state) =

match s with ...

| Binary (op, e1, e2) →
|     let v1 = e_meaning e1 q in
|     let v2 = e_meaning e2 q in
|     b_meaning op v1 v2
|...