1

Semantics of Programming Languages

@ 1993 John Mylopoulos

2

Semantics of Programming Languages

• What is "semantics"? The *meaning of symbol structures*, such as actions, expressions or statements

> *Semantics* applies in general to any means of communication, including actions (for example, political actions, gestures,...) and notations, formal or not, such as natural languages, programming languages, notations for architectural design etc.

• In our case, we are interested in the meaning of programs, program statements and program expressions; in particular, we are interested in:

> What effects does a program have;
>
> What is the value of an expression, given the current machine state;
>
> What does a symbol stand for.

@ 1993 John Mylopoulos

3

Semantics of Programming Languages

There are three basic approaches to giving semantics for a programming language:

Operational approach: the meaning of programs is given by specifying how programs can be executed by an abstract machine defined through an interpreter or a translator.

Axiomatic approach: meaning of programs is now given in terms of a set of axioms and inference rules, analogously to Logic.

Denotational approach: meaning now specified in terms of a mapping function from expressions to entities in the domain of discourse.

For Prolog, defining its semantics in terms of its inference engine amounts to an operational semantics; defining Prolog's semantics in terms of a set of axioms and inference rules qualifies as an axiomatic semantics.

@ 1993 John Mylopoulos

4

Why do we need Semantics?

When defining a programming language we would like to:

clearly express what programs are well-formed
clearly express what the programs mean

using a notation understandable by

language designers
language users
language implementors

A program in any language can be viewed as a string of characters chosen from some alphabet.

• Syntax versus Semantics

Syntax tells us whether a symbolic expression has the right form. Typical formalisms for syntax are BNF, regular expressions and context free grammars.

Semantics tells us how a syntactically correct program is interpreted, or what it does.

@ 1993 John Mylopoulos

5

Syntax versus Semantics

Assume we have the following statement in an Algol-like language:

for I := 1 step 10-J until 10*J do J := J + 1

Interpretation 1 (PL/I-like)

- The limit 10*J and the increment 10-J are to be evaluated once before the loop.
- The exit condition of I > limit is tested before entering the loop.

⇒ Thus if J = 5 before the loop, we would run through ten times and exit.

Interpretation 2

- The limit and increment are evaluated every time at the head of the loop.
- The exit condition of I > limit is tested before entering the loop.

⇒ Thus if J = 5, then the increment soon becomes negative driving I down, while J is increased; the loop won't terminate

@ 1993 John Mylopoulos

6

Syntax versus Semantics

Assume we have the following statement in an Algol-like language:

for I := 1 step 10-J until 10*J do J := J + 1

Interpretation 3 (Algol-like)

- The limit and increment are evaluated every time at the head of the loop.
- If the increment is positive, then the exit condition of I > limit is tested before entering the loop.
- If the increment is negative, then the exit condition of I < limit is tested before entering the loop.

fi Thus if J = 5, then when the increment becomes negative for J=11, the loop is terminated.

These interpretations describe the **semantics** of three different languages, with the same **syntax.**

How can we describe formally the semantics of a programming language?

@ 1993 John Mylopoulos

7

Operational Semantics

A machine language program "means" exactly what the computer (a concrete machine) does when the program is run.

The semantics of a concrete machine consists of rules for the execution of machine instructions that depend on the "state" of the machine as described by registers, flags, program counter and memory

The interpretive approach postulates an abstract machine consisting of set of abstract states and a set of abstract operations which effect state transitions. The semantics of a language $\mathcal{L}$ are given by an interpreter which interprets programs in $\mathcal{L}$ by specifying the effect they will have on the abstract machine. In general, an interpreter I is a function

$$I: \mathcal{M} \times \mathcal{P}_{\mathcal{L}} \rightarrow \mathcal{M}$$

where $\mathcal{M}$ - set of abstract machine states
$\mathcal{P}_{\mathcal{L}}$ - set of programs expressible in the language $\mathcal{L}$

@ 1993 John Mylopoulos

8

Operational Semantics

Consider the language *Mini:*

<program> ::= <statement> {; <statement>}.
<statement> ::= <identifier> := <expression>
<identifier> ::= ...
<expression> ::= ... (arithmetic expressions only)

Assume that we want to write an interpreter with respect to an abstract machine defined by Lisp; assume Mini programs are represented as lists:

```
((:= X 52) (:= Y (+ X 21)))
```

```
(defun interpreter (p)
 (cond  ((null p) nil)
   (t ((set (cadar p) (eval (caddar p)))
        (interpreter (cdr p))))))
```

What about expressions that don't have an effect on machine states?

@ 1993 John Mylopoulos

9

Operational Semantics: Another Example

The syntactic analysis of a program (say using a BNF formalism) can be used to transform the program into an internal representation that can be more easily analyzed.

For example the simple language consisting only of arithmetic expressions in infix notation where every leaf is a number:

((1 + 8) / (4 - 1)) has a value of 3

can be represented in terms of the following list in Polish (postfix) notation:

(1 8 + 4 1 - /)

@ 1993 John Mylopoulos

10

An Interpreter for Polish Notation

This list in Polish notation can be given a meaning using the following Interpreter (written in Lisp):

```
(defun pn (code stack)
  (if (null code) (car stack)
     (let ((instr (car code))
           (coden (cdr code)))
        (pn coden
        (if  (numberp instr)
         (cons instr stack))
           (let ((LHS (car stack))
              (RHS (cadr stack))
              (ST (cddr stack)))
           (cons  (cond
              ((eq instr +) (+ LHS RHS))
              ((eq instr *) (* LHS RHS))
              ((eq instr -) (- LHS RHS))
              ((eq instr /) (/ LHS RHS)))
                      ST)
             ))))))

-> pn('( 1 8  + 4 1 - /) ())
   3
```


@ 1993 John Mylopoulos

11

An Interpreter for Polish Notation

This list in Polish notation can be given a meaning using the following Interpreter (written in Prolog):

```
pn( [], [V | _], V).

pn ([+ | Coden],  [LHS, RHS | ST ], V) :-
   X is LHS + RHS, pn(Coden , [X | ST], V).

pn([ * | Coden],[LHS, RHS | ST ],V) :-
   X is LHS * RHS, pn(Coden, [X | ST], V).

pn([ - | Coden],[LHS, RHS | ST ],V) :-
   X is LHS - RHS,  pn(Coden, [X | ST], V).

pn([ / | Coden],[LHS,RHS | ST ],V) :-
  X is LHS // RHS, pn(Coden,[X | ST],V).


pn( [Num | Coden],ST, V) :-
   pn(Coden, [Num | ST ], V).

|?- pn([1, 8 , +, 4, 1, -, /], [], Result)
Result = 3
yes
```


@ 1993 John Mylopoulos

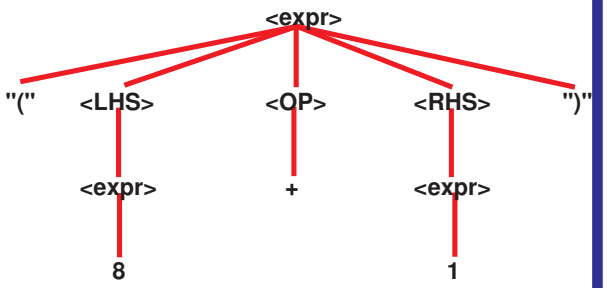
12

Parse Trees (a reminder)

Assume we have the BNF definition of infix expressions:

```
<expr>    ::= <number>
            |   "(" <LHS> <OP> <RHS> ")"
<LHS>     ::= <expr>
<RHS>     ::= <expr>
<OP>      ::= "+" | "-" | "*" | "/"
```

and we parse (8 + 1) then the parse tree is :



@ 1993 John Mylopoulos

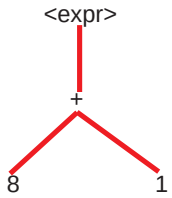
13

Abstract Syntax Trees (ASTs)

Assume we have the BNF definition of infix expressions:

```
<expr>    ::= <number>
          |   "(" <LHS> <OP> <RHS> ")"
<LHS>     ::= <expr>
<RHS>     ::= <expr>
<OP>      ::= "+" | "-" | "*" | "/"
```

The parse tree contains redundant information which when removed yields the (abstract) syntax tree. For example, the expression (8 + 1) becomes:



This tree could be internally represented as a compact data structure. This tree representation lends itself to easy restructuring during optimization phases.

@ 1993 John Mylopoulos

14

Operational Semantics

The translation approach also postulates an abstract machine consisting of set of abstract states and a set of abstract operations which effect state transitions. The translation approach assigns meaning to programs in $\mathcal{L}$ by translating them to programs in the abstract machine language $\mathcal{L}(\mathcal{M})$. In general, a translator $\mathcal{T}$ is a function

$$\mathcal{T}: \mathcal{P}_{\mathcal{L}} \longrightarrow \mathcal{P}_{\mathcal{L}(\mathcal{M})}$$

Compilers define the semantics of programming languages operationally, using the translation approach.

For Mini, let $\mathcal{L}(\mathcal{M})$ be an assembly language. Then $\mathcal{T}$ maps the program

```
X := 52;
Y := X + 21.
```

to something like

```
load FIFTYTWO          % to accumulator
store X                % X ← accumulator
load TWENTYONE
add X                  % acc ← X + accumulator
store Y
```


@ 1993 John Mylopoulos

15

Approaches to Specifying Semantics

Extensible Definition

Define the semantics of certain primitive operators of a language (using some other means), and then define the rest of the language in terms of the primitive operations.

For example in Lisp, it is possible to define the semantics of **eval** using eleven primitives:

cons, car, cdr
eq, atom
read, print
getplist, putplist
quote, cond

(See pp 125-172 of Functional Programming in Lisp)

This is generally possible in extensible languages such as Lisp or Prolog.

@ 1993 John Mylopoulos

16

Axiomatic Approach

How does it work for Logic? Axioms, inference rules, provability relation (|-)

For example, for Propositional Logic,

- Axioms
  $A \Rightarrow (B \Rightarrow A)$
  $((A \Rightarrow (B \Rightarrow \Gamma)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow \Gamma)))$
  $((\neg A \Rightarrow \neg B) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B))$

- Inference rule
  modus ponens: $A, A \Rightarrow B \vdash B$

- Provability relation
  transitive closure of |-

How can we adopt this for programs?

@ 1993 John Mylopoulos

17

Axiomatic Approach

The behaviour of a program is specified by a precondition/postcondition pair (α, β). A program p satisfies (α, β)

$\{\alpha\}$ p $\{\beta\}$

if and only if when α holds before execution of p, β holds after execution (assuming that p terminates).

For example, consider again

```
X := 52;
Y := X + 21.
```

and let

$\alpha \equiv$ `true`

$\beta \equiv X = 52 \wedge Y = X + 21$

• Exercise

```
{true} X := 5 { X = 5}        ??
{Y = 5} X := 5 {Y = X}        ??
```

How do we prove such formulas?

@ 1993 John Mylopoulos

18

Axioms and Inference Rules

We need axioms and inference rules.

Assignment axiom

$\{\alpha\theta\}$ X := t $\{\alpha\}$ where $\theta = \{X/t\}$

Inference rules

$$\frac{\{\alpha\}\ s_1\ \{\beta\},\ \{\beta\}\ s_2\ \{\gamma\}}{\{\alpha\}\ s_1; s_2\ \{\gamma\}}$$ inference rule for linear sequences

$$\frac{\{\alpha \wedge g\}\ s_1\ \{\beta\},\ \{\alpha \wedge \neg g\}\ s_2\ \{\beta\}}{\{\alpha\}\ \text{if g then } s_1 \text{ else } s_2\ \{\beta\}}$$ inference rule for conditional

$$\frac{\{\alpha \wedge g\}\ s\ \{\alpha\}}{\{\alpha\}\ \text{while g do s od}\ \{\alpha \wedge \neg g\}}$$ inference rule for iteration

Now we can prove properties of programs

@ 1993 John Mylopoulos

19

Proofs

Prove:

{true} X := 5 { X = 5 }

Start with {αθ} X := t {α} and let t = 5, α = (X = 5)

{ (X = 5)θ } X := 5 { X = 5 }

But θ = {X/5}, so

{ (5 = 5) } X := 5 { X = 5 }

or

|- {true} X := 5 { X = 5 }

So, we have proved this theorem in the axiomatization just presented

Likewise, choose α = (Y = X), t = 5 in the substitution axiom:

{ (Y = X)θ } X := 5 { Y = X }

Again, θ = {X/5} so

|- { Y = 5 } X := 5 { Y = X }

@ 1993 John Mylopoulos

20

Proofs

Prove

{true} X := 52; Y := X + 21. {X = 52 ∧ Y = X + 21}

start with {αθ} X := t {α} and let α = (X = 52 ∧ Y = X + 21), t = X + 21:

|- { (X = 52 ∧ X + 21 = X + 21)} Y := X + 21 {(X = 52 ∧ Y = X + 21)}

or

|- { (X = 52)} Y := X + 21 {(X = 52 ∧ Y = X + 21)}

Also,

|- {true} X := 52 { X = 52}

Now use the inference rule for sequencing

{true}X:=52{X=52},{X=52}Y:=X+21{(X=52∧Y=X+21)}

{true} X := 52; Y := X + 21 {(X=52 ∧ Y=X+21)}

|- {true} X := 52; Y := X + 21 {(X=52 ∧ Y=X+21)}

@ 1993 John Mylopoulos

21

Denotational Approach

Map symbols to their "meanings"

• Use semantic valuation functions to map symbols to their meanings

• Compositionality: the meaning of the whole is a function of the meaning of the parts

Meaning characterized independently of implementation

@ 1993 John Mylopoulos

22

Denotational Approach

• How does the denotational approach work for Logic?

Try an example

friendOf(john,father(mary)) $\vee$ brotherOf(eve,jim)

Syntactic elements:

john, mary, eve, jim	constant symbols
father(.)	function symbol
friendOf(.,.), brotherOf(.,.)	predicate symbols
friendOf(...) $\vee$ brotherOf(...)	well-formed formula (wff)

Model: $(\mathcal{U}, \mathcal{R}, \mathcal{F})$

$\mathcal{U}$ - universe of discourse

$\mathcal{R}$ - set of relations over $\mathcal{U}$

$\mathcal{F}$ - set of functions over $\mathcal{U}$

For our example

$\mathcal{U}$ = {a, b, c, d}

$\mathcal{R}$ = { R_f = {(a,b), (a, c)}, R_b = {(b,c)}}

$\mathcal{F}$ = { F_f = {(b,d), (c,d)}}

@ 1993 John Mylopoulos

23

Denotational Approach

- The valuation function ν will be defined in stages:
  - σ: *Symbols* $\rightarrow \mathcal{U} \cup \mathcal{R} \cup \mathcal{F}$
  - τ: *Terms* $\rightarrow \mathcal{U}$
  - ν: *Wffs* $\rightarrow$ *{true, false}*

σ(john) = a, σ(mary) = b, σ(eve) = c, σ(jim) = d
σ(friendOf) = R_f, σ(brotherOf) = R_b
σ(father) = F_f

τ on terms:
τ(c) = σ(c) for constant symbols c
τ(f(t_1, t_2,..., t_n)) = σ(f)(τ(t_1), τ(t_2),..., τ(t_n))

ν on atomic literals:
ν(p(t_1, t_2,..., t_n)) = (τ(t_1), τ(t_2),..., τ(t_n)) $\in$ σ(p)

ν on wffs (including no variables and quantifiers):
ν(α ∨ β) = true if ν(α) or ν(β) is true, false otherwise
ν(friendOf(john,father(mary))) = ??
ν(brotherOf(eve,jim)) = ??
ν(friendOf(john,father(mary)) ∨ brotherOf(eve,jim)) = ??

@ 1993 John Mylopoulos

24

Validity of Formulas

A formula α is valid, |= α iff for every interpretation ν(α) = true.

Clearly,

|= α ∨ ¬ α

for any α

Soundness and completeness theorem: |- α ιff |= α

In other words, a formula can be inferred with respect to the axioms and inference rules of (Predicate) Logic if and only if it is true in all interpretations.

Soundness and completess theorem justifies the provability relation chosen

@ 1993 John Mylopoulos

25

Denotational Approach Applied to Mini

How do we apply this to Mini? Use the same general approach as for Logic:

σ: *Symbols* $\rightarrow \mathcal{U} \cup \mathcal{R} \cup \mathcal{F}$
τ: *Expressions* $\times \Sigma \rightarrow (\mathcal{U} \cup \Sigma)$
ν: *Wffs* $\rightarrow$ *{true, false}*

What is the universe of discourse? Numbers, variables with values (therefore machine states). Let's represent the latter with lists of (variable-value) pairs
e.g., ((X, 6), (Y, 5), (Z, 190))

$$\mathcal{U} = \mathcal{N} \cup \Sigma \cup \{true, false\}$$

$\mathcal{N}$ – set of numbers
Σ – set of machine states (lists of pairs)

What functions and relations do we need? Arithmetic functions, state functions

σ: Numerals $\rightarrow \mathcal{N}$
$\sigma(+)$ = addition function for numbers...

@ 1993 John Mylopoulos

26

Denotational Approach

τ: Arithmetic Expressions $\times \Sigma \rightarrow \mathcal{N}$
τ: Arithmetic Predicates $\times \Sigma \rightarrow$ *{true, false}*

$\tau(n) = \sigma(n)$ for numeral n
$\tau(X, s) = V$ if there is a tuple (X,V) in s
$\tau(f(t_1, t_2,..., t_n)) = \sigma(f)(\tau(t_1), \tau(t_2),..., \tau(t_n))$

$\tau(\text{ id := e, s}) \in \Sigma$ or, equivalently,
$\tau(\text{ id := e}) \in \Sigma \rightarrow \Sigma$

For example,

$\tau(\text{ X := 21}) = f_{X:=21}: \Sigma \rightarrow \Sigma$ such that

$f_{X:=21}((X_1,V_1), (X_2, V_2), ..., (X_m, V_m))) =$
if $X \neq X_j$ for all j then $((X_1,V_1), (X_2, V_2), ..., (X_m, V_m))$
else if $X = X_j$ $((X_1,V_1), (X_2, V_2), ...,(X, 21),..., (X_m, V_m))$

$\nu(\{ \alpha \} p \{ \beta \},s) =$
true if for a given state $s \in \Sigma$, if $\tau(\alpha, s)$ is true and p terminates, then $\tau(\beta, s)$ is true

The denotational approach explains what are expressions in the language about.

@ 1993 John Mylopoulos

27

Relating the Axiomatic and the Denotational Approach

A wff is valid, $\models \{ \alpha \} p \{ \beta \}$, iff
$\nu(\{ \alpha \} p \{ \beta \}, s) = \text{true}$ for all states $s \in \Sigma$

Soundness: If $\vdash \{ \alpha \} p \{ \beta \}$ then $\models \{ \alpha \} p \{ \beta \}$

Proof: Use induction on the number of assignment statements in p.

Completeness: If $\models \{ \alpha \} p \{ \beta \}$ then $\vdash \{ \alpha \} p \{ \beta \}$.

Proof: Again, use induction on the number of assignment statements in p.

@ 1993 John Mylopoulos