IMPLEMENTATION OF DENOTATIONAL SEMANTICS

INFORMATICA 1/91

M. MERNIK, V. ŽUMER
Technical Faculty of Maribor
University of Maribor
Smetanova 17, 62000 MARIBOR

Keywords: formal methods, semantics, rapid prototyping, functional languages

ABSTRACT : In the paper formal methods for semantics definition and reasons for their appearance are briefly described. Denotational approach to semantics definition is more detail explained in next chapter. Implementation of denotational semantics lead us to operational approach and to rapid prototyped interpreters. We represent one of the possible implementation of denotational semantics using functional language LISP.

KEYWORDS : formal methods, semantics, rapid prototyping, functional languages

1. INTRODUCTION

A programming language is a notation for describing computations and is defined with syntax, semantics and pragmatics. The structure of statements is given by syntax. The area of syntax has been intensively studied and Backus - Naur form (BNF) is widely used for defining syntax, because there exists a close correspondence between the BNF definition and parser. The meaning of statements is given by semantics and language implementation on specific computer is described by pragmatics. In this paper we briefly describe semantics definition methods. Unfortunately, the semantic area is not as well developed as the syntax area, because semantic features are much more difficult to define and describe. Semantics of early developed programmming languages were described in natural languages. This description has the advantage that semantics was easily understood but also many disadvantages such as: ambiguity, inaccuracy, misunderstanding, etc. So we need formal semantics definitions which, ensures us :-

- precise standards for a computer implementation, which guarantes that the language implementation is exactly the same on all machines ;
- usefull user documentation ;
- a tool for design and analysis ;
- input to compiler generator, which maps semantic definitions to a guaranted correct implementation of the language.

The effort, on formal methods for semantics definition, done in 70-ties have brought us three distinct and complementary description methods : operational, denotational and axiomatic semantics [3,4,6,8].

The operational semantics method uses an interpreter to define a language. The meaning of a program is the evaluation history that the interpreter produces when it interprets the program.

The denotational approach to semantics makes use of mappings, which are called semantic valuation functions. These map syntax constructs into their abstract mathematical counter part ; thus numerals are mapped into numbers, procedures are mapped into mathematical functions, and so on. The denotational definitions are more abstract then operational.

With the axiomatic semantics method, the meaning of program is not explicitly given at all. Instead, properties about language constructs are defined. These properties are expressed with axioms and inference rules. Axiomatic definitions are more abstract then denotational and operational.

All three methods together provide a set of tools for language development. Designers might first define properties that they wish the system

to have, so axiomatic definition is constructed first. Next, a denotational semantics is defined to give the meaning of the language. Finally, the denotational definition is implemented using an operational definition. These complementary semantic definitions of a language support systematic design, development and implementation. In the paper we describe denotational approach and one of its possible implementation using functional programming language LISP.

2. DENOTATIONAL SEMANTICS

Denotational semantics [1,2,3] is a methodology for giving mathematical meaning to programming languages. To make things more understandable an example of denotational semantics for a simple language of binary numbers is given first. Its syntax is defined with :

$$\nu ::= \nu\ \delta \mid \delta$$
$$\delta ::= 0 \mid 1$$

The language of binary numbers is composed by sequence of digits 0 and 1. To define the meaning of statement a valuation semantic function V which maps sequence of digits into their meaning is constructed. We say that the domain of V is a sequence of digits and the codomain are integers which represents meanings of digits. Mathematical notation for the function V is :

V : Bin -> Integer

We must define an instruction, which maps the domain into the codomain for each BNF rule :

V[0] = 0
V[1] = 1
V[νδ] = 2 * V[ν] + V[δ]

The next example, shows the semantic valuation function V on work :

V[1001] = 2 * V[100] + V[1]
= 2 * 2 * V[10] + V[0] + 1
= 2 * 2 * 2 * V[1] + V[0] + 0 + 1
= 2 * 2 * 2 * 1 + 0 + 0 + 1
= 9

Therefore the denotational approach to languages definition must make clear several sets of quantities :

- the syntax rules,
- the definition of the various semantic functions,
- the syntax domains,
- the semantic domains.

For describing syntax rules the abstract syntax is used. It specifies the relations between logical parts of the language and can be simpler than concrete syntax, and may not contain enough information to parse the language unambiguously. On contrary, concrete syntax contains sufficient information to parse a language.

λ-notations is used for describing semantic functions. To show the advantage of a λ-notation following definitions must be stated first. Function f(x) = y is equivalently described in λ-notation with f = λx.y . In this notation all functions usually have only one argument or one parameter. Consider a function f with two variables x and y. A function g with the property g(x)(y) = f(x,y) can be defined and is in some sense equivalent to f. The function g is called the curried version of f. Function which produced another function on its output is called the high order function. λ-notation has following advantages :

- a high order function is easier to describe ;
  example :
  add : ℕ -> ℕ -> ℕ
  standard notation :
  add(n) = f , where f(m) = n+m
  λ-notation :
  add = λn.λm.n+m

- a function can be defined without giving it a name. Such function is called the anonymous function.

Several additional notations may be used in expressions of semantic equations, such as "update" expressions of the form [a ↦ b]f, which denotes a function f which is the same as the function f except, that it maps the argument a into the value b. Conditional expressions are written in the form "t -> e_1 □ e_2" where the value of the expression e_1 is evaluated if and only if the boolean expression t is true and the value e_2 is evaluated otherwise.

Syntax domains of imperative languages are usually identifiers, expression, commands and definitions. Semantic domains are most conventionally described with an environment and a store. The environment is a finite set of associations of identifiers with values they denote (denotable values) and the store is a

finite set of locations with values they contain (storable values). The store will be sufficient for simple languages where equal identifier can not be used for different objects.

And now, let us define the meaning of identifiers. Identifiers may stands for location, label, procedure and theirs meaning are represent by semantic domain environment which is a function from identifiers to denotable values. If identifier stands for location , a second semantic domain is used to get value they contain. The store is therefore a function from locations to storable values.

The meaning of commands is to change the store, the meaning of a definition is to change the environment and the meaning of an expression is to produce a value (expressible value).

3. IMPLEMENTING DENOTATIONAL SEMANTICS USING LISP

A great advantage of the denotational approach is that can be mechanized and interpreted on a computer [1,2,7]. In this chapter a small and a simple example is described with the denotational semantics first and then implemented using functional language LISP. LISP is selected because high order functions are relatively simple to express using it.

The abstract syntax for a simple imperative language is given first :

P ∈ Program
B ∈ Block
D ∈ Declaration
C ∈ Command
E ∈ Expression
I ∈ Identifier
N ∈ Numeral

P ::= B.
B ::= begin D & C end
D ::= D_1 & D_2 | var I
C ::= C & C_2 | I:=E |
if E then C_1 else C_2 | B
E ::= E_1 + E_2 | I | N

Program is a block of statements and each block consists of a declaration part and a command part. Commands are : assignment statement, if statement and block of statements. Expressions are : identifiers, numerals and composed expressions. Delimiter symbol between statements is a character &.

Semantics are described as follows :
Domains and operations on it are :

I. Natural numbers n ∈ N
II. Identifiers i ∈ Id
III. Location l ∈ Loc
reserve_loc : Loc
reserve_loc = random 1000
IV. Denotable values d ∈ D_values = Loc
V. Environment e ∈ Env = Id -> D_values
emptyenv : Env
emptyenv = λi.0
accessenv : Id -> Env -> D_values
accessenv = λi.λe.e(i)
updateenv : Id -> Value -> Env -> Env
updateenv = λi.λd.λe.[i ↦ v]e
VI. Storable values v ∈ S_value = N
VII. Store s ∈ Store = Loc -> S_value
emptystore : Store
emptystore = λi.0
access : Loc -> Store -> S_value
access = λi.λs.s(i)
update : Loc -> S_value ->
Store -> Store
update = λi.λv.λs.[i ↦ v]s
VIII. Expressible values x ∈ Exp_vaules = N

Semantic valuation functions is given next :
P: Program -> Store
P[B] = B[B] emptyenv emptystore

The meaning of a program is defined with a store, which is the output from function B. The function B needs environment and store on the input. The function B is applied on initialized environment and store first.

B: Block -> Env -> Store -> Store
B[begin D & C end] = λe.λs.C[C] (D[D] e) s

The meaning of a block is the meaning of commands, which is evaluated on environment and store. The environment is changed by function D first.

D: Declarations -> Env -> Env
D[D_1 & D_2] = λe.D[D_2] (D[D_1] e)
D[var I] = λe.updateenv [I] reserve_loc e

The meaning of a declaration is to change the environment. When an identifier declaration is evaluated, new location is produced first and then new updated environment is returned. This updated environment has a property that it maps an identifier to a new location.

```
C: Command -> Env -> Store -> Store
C[B] = λe.λs.B[B] e s
C[C1 & C2] = λe.λs.C[C2] e (C[C1] e s)
C[I:=E] = λe.λs.update (accessenv [I] e)
                       (E[E] e s) s
C[if E then C1 else C2] = λe.λs.E[E]s = 0 ->
                                  C[C1]e s □
                                  C[C2]e s
```

The meaning of a command is to change the store. The meaning of an if statement is to evaluate the expression E first and then appropriate command is selected and evaluated. The meaning of an assignment statement is :

- a location for an identifier is evaluated (accessenv [I] e) .
- an expression E is evaluated (E[E] e s) .
- a location is updated with value produced by the function E
  (update (accessenv [I] e) (E[E]e s) s) .

```
E: Expression -> Env -> Store -> Exp_value
E[E1+E2] = λe.λs.E[E1]e s + E[E2]e s
E[I] = λe.λs.access (accessenv [I] e) s
E[N] = N[N]
```

The meaning of an expression is to produce a new value. Semantic valuation function N is similar to function V , which is introduced in chapter 2. Now, we can apply syntax semantic valuation functions to syntax construct to get their meaning.

Example :

```
P[begin var i; i:=10; begin var i; i:=20 end;
   i:=i+1 end] =
B[begin var i; i:=10; begin var i;  i:=20  end;
  i:=i+1  end]  emptyenv emptystore =
C[i:=10; begin var i; i:=20 end; i:=i+1]
  (D[var i]emptyenv) emptystore =
C[i:=10; begin var i; i:=20 end; i:=i+1]
 (updateenv [i] reserve_loc emptyenv)
   emptystore =
C[i:=10; begin var i; i:=20 end; i:=i+1]
  [i ↦ 1]emptyenv emptystore =
C[begin var i; i:=20 end; i:=i+1]
  [i↦ 1]emptyenv (C[i:=10] [i ↦ 1]emptyenv
    emptystore) =
C[begin var i; i:=20 end; i:=i+1]
  [i↦ 1]emptyenv (update (accessenv [i]
    [i ↦ 1] emptyenv) (E[10] [i↦ 1]emptyenv
      emptystore) emptystore) =
C[begin var i; i:=20 end; i:=i+1]
[i↦ 1]emptyenv (update 1 10 emptystore) =
C[begin var i; i:=20 end; i:=i+1] [i↦ 1]emptyenv
  [1 ↦ 10]emptystore =
 let e1= [ i↦ 1]emptyenv
 let s1= [1 ↦ 10]emptystore
C[i:=i+1] e1 (C[begin var i; i:=20 end] e1 s1 ) =
C[i:=i+1] e1 (B[begin var i; i:=20 end] e1 s1 ) =
C[i:=i+1] e1 (C[i:=20] (D [var i] e1) s1) =
C[i:=i+1] e1 (C[i:=20]
  (updateenv [i] reserve_loc e1) s1 ) =
C[i:=i+1] e1 (C[i:=20] [1 ↦ 2]e1 s1 ) =
C[i:=i+1] e1 (update (accessenv [i] [1 ↦ 2]e1)
  (E[20] s1) s1 ) =
C[i:=i+1] e1 (update 2 20 s1 ) =
C[i:=i+1] e1 [2 ↦ 20]s1 =
  let e2= [ i↦ 2]emptyenv
  let s2= [1 ↦ 10, 2 ↦ 20]emptystore
update ( accessenv [i] e1 ) (E[i+1] e1 s2) s2 =
update 1 ((acess (accessenv [i] e1 )s2) + 1) s2 =
update 1 (access 1 s2)+1 s2 =
update 1 10+1 s2 =
update 1 11 s2 =
[1 ↦ 11, 2 ↦ 20]emptystore
```

The meaning of the above program is the store, which map location 1 to value 11 and location 2 to value 20.

Implementation of semantic valuation functions is shown in appendix . Now, we can analyze any semantic valuation function.

Examples :

```
(setq new_env
 (funcall (D '(var i & var j & var k)) emptyenv))

        =>  ( is evaluated to )

   ( (i 599) (j 41) (k 855) )
```

Function D changed the environment. The new environment maps identifier i to location 599, identifier j to location 41 and identifier k to location 855.

```
(funcall (funcall (C '( ( i := (1)) &
                        ( j := (2)) &
                        ( k := ((i) + (j)))
                       )
                  ) new_env ) emptystore)

        =>  ( is evaluated to )

   ( (599 1) (41 2) (855 3) )
```

Function C changed the store. The new store maps location 599 to value 1, location 41 to value 2 and location 855 to value 3.

```
(setq prog '(begin (var i & var j & var k) &
             ( (i := (1))  &
               (j := ((i) + (1))) &
               ( begin (var i & var l & var m) &
                       ( (i := (20)) &
                         (m := ((i) + (1))) &
                         (l := ((j) + (1)))
                       )
                   end ) &
               (k := ((i) + (2)))
             )
            end ))


 ( P prog )

    =>  ( is evaluated to )

( (599 1) (41 2) (784 20) (503 21) (981 3)
 (855 3) )
```

We can understand the store only with the environment. The environment for that example is :

((i 599)(j 41)(k 855)(i 784)(l 981)(m 503))

4. CONCLUSION

The implementation of semantic valuation functions has many advantages [1,2,5,7] :

- each semantic valuation function can be separately tested and analyzed,
- a rapid prototyped interpreter for the language is produced
- all advantages of prototyped systems such as :
  - greater user participation,
  - early detection of errors,
  - better user - developer communications,
  - early delivered executable systems to the users,
  - help by suppressing uncertainties in user requirements (I'll know what I want, when I see it !).

5. REFERENCES

1. Schmidt D.A., "Denotational semantics", Allyn and Bacon, Newton, 1986.

2. Allison L., "A practical introduction to denotational semantics", Cambridge university press, Melbourne, 1986.

3. Tennent R.D., "Principles of Programming Languages", Prentice Hall International, London, 1981.

4. Horowitz E., "Fundamentals of Programming Languages", Computer science press, Rockville, 1983.

5. Agresti W.W.,"New paradigms for software development",IEEE Computer society press, New York, 1986.

6. Mernik M., Kokol P., Žumer V.,"Formalne metode za opis semantike programskega jezika", XIV. Simpozijum o informacionim tehnologijama, Sarajevo - Jahorina 1990, pp. 181-1 - 181-10.

7. Mernik M., Kokol P., Žumer V.,"Π-BOSS: A new rapid prototyping paradigm for language design", Proceedings of International Conference on Computing and Information, Niagara Falls, pp. 504 - 507, 1990.

8. Marcotty M., Ledgard H.F., Bochmann G.V., "A Sampler of Formal Definitions", Computing Surveys, Vol. 8., No. 2., 1976.

APPENDIX : SEMANTIC VALUATION FUNCTIONS IN LISP

```
;; Env = Id -> D_value
;; Environment is modeled with association list
;;  ( (id1 loc1) ... (idn locn) )

;; accessenv: Id -> Env -> D_values
 (defun accessenv ()
  #'(lambda (i)  #'(lambda (e) (cadr (assoc i e))))

;; updateenv: Id -> D_value -> Env -> Env
 (defun updateenv ()
  #'(lambda (i)
   #'(lambda(d)
    #'(lambda (e) (putassoc e i d)))))

;; Store = Loc -> S_value
;; Store is modeled with association list
;;    ( (loc1 value1) ... (locn valuen) )

;; access : Loc -> Store -> S_value
 (defun access ()
  #'(lambda (l)
   #'(lambda (s) (assocl l s))))

;; update : Loc -> S_value -> Store -> Store
 ( defun update ()
  #'(lambda (l)
   #'(lambda (v)
    #'(lambda (s) (putassoc s l v)))))
```

```
;; reserve_loc is modeled with random numbers
 (defun reserve_loc ()  ( random 1000 ))

;; environment initialization
 (defun emptyenv ()   nil )

;; store initialization
 (defun emptystore ()  nil )

;; P: Program -> Store
 ( defun P (Program)
   (funcall (funcall (B  Program) (emptyenv))
     (emptystore)))

;; B: Block -> Env -> Store -> Store
 ( defun B (Block)
  #'( lambda ( e )
   #'(lambda ( s ) (funcall (funcall (C (cadddr
    Block)) (funcall (D (cadr Block)) e)) s))))

;; D: Dec -> Env -> Env
 (defun D (Dec)
  #'(lambda (e)
     (cond ((equal (caddr Dec) '&)
        ;; composed declaration
       (funcall (D (cdddr Dec)) (funcall
        (D (list (car Dec)  (cadr Dec))) e)))
      ;; identifier declaration
      (t (funcall (funcall (funcall
  (updateenv) (cadr Dec)) (reserve_loc)) e )))))
;; C: Comm -> Env -> Store -> Store
 (defun C (Comm)
  #'(lambda (e)
   #'(lambda (s)
     (cond ((not (equal (cdr Comm) nil))
       ;; composed commands
       (funcall (funcall (C (cddr Comm)) e)
        (funcall (funcall (C (list (car Comm)))
          e) s)))
      ;; if statements
      ((equal (caar Comm) 'IF)
         (cond ((equal (funcall (funcall
                (E (cadar Comm)) e) s) 0)
           (funcall (funcall (C (list
             (cadddr (car Comm)))) e) s))
          (t (funcall (funcall (C (list
              (caddddr (car Comm)))) e) s))))
       ;; block statements
       ((equal (caar Comm) 'BEGIN)
          (funcall (funcall (B (car Comm)) e) s))
        ;; assignment statement
        (t (funcall (funcall (funcall
          (update) (funcall (funcall
          (accessenv) (caar Comm)) e))
          (funcall (funcall (E
            (caddar Comm)) e ) s)) s))))))

;; E: Exp -> Env -> Store -> Exp_value
 (defun E (Exp)
  #'(lambda (e)
   #'(lambda (s)
     ;; number
     (cond ((numberp (car Exp)) (car Exp))
     ;; Exp + Exp
     ((equal (cadr Exp)  +)
        (+ (funcall (funcall (E (car Exp)) e) s)
          (funcall (funcall (E (caddr Exp))
           e) s)))
      ;; identifier
      (t (funcall (funcall (access)
              (funcall (funcall (accessenv )
               (car Exp)) e)) s))))))
```