Denotational Semantics

The main idea behind the denotational method is that the meaning of a program can be explained in terms of the application of functions to their arguments. Since the mathematics of functions is well understood, we can base the semantics of a programming language on primitive, well-behaved mathematical objects without using a real machine or even a virtual machine like operational semantics does. Moreover, we can also model all the major concepts like memory, variables, types etc. using sets and functions, and thus avoid the high level of abstraction in the axiomatic method. For these reasons, the denotational method ahs been the most studied method, and is generally considered the best, although this is clearly relative to the use to which the semantic method is put.

Fundamentals

A language is partly defined through its syntax. If we define an abstract syntax for a language, then the forms in the syntax give us a source for the semantic attachments; every syntactic form will have a corresponding semantic object attached to it. Every semantic object is either a set or a function (which is just a mapping between sets anyway). Just as in the operational and axiomatic methods, we start with the idea of program state. However, this is modeled using a function, called the *store*. The basic idea of a store is simple: it is a mapping between names of variables and the values they contain. We can write this as $store: I \rightarrow V$, where I is a *domain* (a set of semantic objects) of names, or identifiers, and V is a domain of values, including numbers, characters, strings and whatever else we consider to be primitive values. The store changes as assignments alter the mapping of individual variables, so we need a way to alter the function. This can be notated as, for instance $s[x \rightarrow 4]$or as update(s, x, 4) where s is a store and in that store, x is mapped to the value 4. This clearly gives us the basis of assignment.

Components

The first step is to define an abstract syntax. We will define a language similar to the one used in the axiomatic example:

P ::= S
S ::= I = E | S_1;S_2 | if B then S_1 else S_2 end | while B do S end
B ::= I == E | I > E | I < E | B_1 or B_2 | B_1 and B_2 | not B
E ::= N | I | E_1 + E_2 | E_1 – E_2

where P is a (syntactic) domain of programs, S is a domain of statements, I is a domain of names (identifiers), B is a domain of Boolean expressions, E is a domain of arithmetic expressions and N is the domain of integer numerals.

This is not a complete language but is adequate to illustrate the parts of the method. At least is has the three structured programming constructs: sequence, conditional and while loop.

The next step is to identify the semantic domains to be used. We will need the following domains: Z, the (semantic) domain of integers; Tr, the domain of truth values; Id, the domain of names, and the store model Id $\rightarrow$ Z, defined above.

The final step is to define a family of meaning functions that each take a piece of the syntax and map it to a semantic object. (Syntax will be surrounded by special square brackets. e.g. ⟦x⟧ means the identifier x.) We will call this meaning function M, although other writers use different function names for the different constructs. If we start with the meaning of numerals, we can write:

M⟦N⟧ = n, where n$\in$Z corresponding to N. Really this is:

M⟦0⟧ = 0, M⟦1⟧ = 1, M⟦2⟧ = 2 etc.

Next comes assignment. Clearly the meaning of a particular assignment depends on the at the point of execution. So we give the function M an extra argument for that store. The form for this follows standard ML-like syntax where the arguments are simply written consecutively, without parentheses.

M⟦I=E⟧s = s[I $\mapsto$ M⟦E⟧s]

which means the store s is updated with the identifier I mapped to the value of the expression E relative to the store s. Note we could write the same thing as a lambda function – the denotation:

M⟦I=E⟧ = λs.s[I $\mapsto$ M⟦E⟧s]

Expressions are handled individually:

M⟦I⟧s = s(I) or M⟦I⟧ = λs.s(I)
M⟦E_1 + E_2⟧s = M⟦E_1⟧s + M⟦E_2⟧s, or M⟦E_1 + E_2⟧ = λs.M⟦E_1⟧s + M⟦E_2⟧s
M⟦E_1 - E_2⟧s = M⟦E_1⟧s - M⟦E_2⟧s, or M⟦E_1 - E_2⟧ = λs.M⟦E_1⟧s - M⟦E_2⟧s

M⟦I==E⟧s = true if s(I) = M⟦E}s and false otherwise, or M⟦I==E⟧ = λs.s(I) *equals* M⟦E⟧s
M⟦I > E⟧s = true if s(i) > M⟦E⟧s and false otherwise, or M⟦I > E⟧ = λs.s(i) *gt* M⟦E⟧s
M⟦I < E⟧s = true if s(i) < M⟦E⟧s and false otherwise, or M⟦I < E⟧ = λs.s(i) *lt* M⟦E⟧s
M⟦B_1 or B_2⟧ s= true if one or both of M⟦B_1⟧s and M⟦B_2⟧s is true and false otherwise,
or M⟦B_1 or B_2⟧ = λs. M⟦B_1⟧s *or* M⟦B_2⟧s
M⟦B_1 and B_2⟧s = false if one of M⟦B_1⟧s and M⟦B_2⟧s is false and true otherwise.
or M⟦B_1 and B_2⟧ = λs. M⟦B_1⟧s *and* M⟦B_2⟧s
M⟦not B⟧s = true if M⟦B⟧s is false and true otherwise, or M⟦not B⟧ = λs.*not* M⟦B⟧s.

The semantic operations *equals, gt, lt, and , or, not* can be expressed in the lambda calculus, but that is not shown here.

The control structures are also handled specially:

$M[\![S_1;S_2]\!]s = M[\![S_2]\!](M[\![S_1]\!]s)$, or $M[\![S_1;S_2]\!] = \lambda s.M[\![S_2]\!](M[\![S_1]\!]s)$

i.e. the result of executing a sequence of two statements with a particular store is the same as executing the second one in the store that results from executing the first one.

$M[\![$if B then S_1 else S_2 end$]\!]s = M[\![S_1]\!]s$ if $M[\![B]\!]s$ = true and $M[\![S_2]\!]s$ if $M[\![B]\!]s$ is false,
or $M[\![$if B then S_1 else S_2 end$]\!] = \lambda s.M[\![B]\!]s \rightarrow M[\![S_1]\!]s \square M[\![S_2]\!]s$, where the form $\_\rightarrow\_\square\_$ is a conditional form that can also be expressed in lambda calculus (not shown here).
i.e. the result of executing a conditional statement depends on the value of the expression B.

$M[\![$while B do S end$]\!]s = M[\![$S; while B do S end$]\!]s$ if $M[\![B]\!]s$ is true and s if $M[\![B]\!]s$ is false,
or $M[\![$while B do S end$]\!] = \lambda s.\ M[\![B]\!]s \rightarrow M[\![$S; while B do S end$]\!]s \square s$

Of course this must be recursive to allow for an indefinite number of iterations of the loop. Note that the body is executed every time B evaluates to true and that the store is returned unchanged if B evaluates to false.

Example

If we take the example as before:

```
p = 0
n = x
while n > 0 do
    p = p + y
    n = n – 1
end
```

We can, with some effort, write out a derivation for $M[\![P]\!]$ with an initial store $s_0 = s[x \mapsto 3][y \mapsto 2]$, where s is an empty store. We simply match each syntactic form with the relevant function and evaluate the expressions relative to the store passed to the various functions. We will not show the whole derivation here, but there are important parts as follows:

The first step is to peel off the first assignment:

$M[\![p = 0; n = x;$ while $n > 0$ do $p = p + y; n = n – 1$ end$]\!]s_0$
$\Rightarrow M[\![n = x;$ while $n > 0$ do $p = p + y; n = n – 1$ end$]\!](M[\![p = 0]\!]s_0)$

Now $M[\![p = 0]\!]s_0 = s_0[p \mapsto M[\![0]\!]s_0]$, and $M[\![0]\!] = 0$ (the store is irrelevant here), so

$M[\![p = 0]\!]s_0 = s_0[p \mapsto 0]$ which is $s[x \mapsto 3][y \mapsto 2][p \mapsto 0]$, or s_1. We have added the binding for p to the initial store. The same goes for n = x:
$M[\![n = x]\!]s_1 \Rightarrow s_1[n \mapsto M[\![x]\!]\ s_1] \Rightarrow s_1[n \mapsto s_1(x)] \Rightarrow s_1[n \mapsto 3]$, or s_2

The loop is handled as:

$M[\![$while n > 0 do p = p + y; n = n – 1 end$]\!]s_2$
$\Rightarrow M[\![p = p + y;n = n - 1;$ while n > 0 do p = p + y; n = n – 1 end$]\!]s_2$, since $M[\![n > 0]\!]s_2$ is true: $M[\![n > 0]\!]s_2 \Rightarrow M[\![n]\!]s_2$ *gt* $M[\![0]\!]s_2 \Rightarrow 3$ *gt* $0 \Rightarrow$ true. Thus the body of the loop will yield a store that will be used for the next iteration. Eventually the store will contain n = 0, and the loop will stop.

An assignment with an expression is handled as:

$M[\![n = n – 1]\!]s_3$
$\Rightarrow s_3[n \mapsto M[\![n - 1]\!]s_3]$
and $M[\![n - 1]\!]s_3$
$\Rightarrow M[\![n]\!]s_3 - M[\![1]\!]s_3$
$\Rightarrow s_3(n) - 1$
$\Rightarrow$ (for instance) 3 - 1, if $s_3 = s[x \mapsto 3][y \mapsto 2][p \mapsto 2][n \mapsto 3]$
$\Rightarrow 2$

If we applied these techniques to derive a meaning for the whole program we would end up with a state $s_f = s[x \mapsto 3][y \mapsto 2][p \mapsto 6][n \mapsto 0]$
Note that we have not proved that the program multiplies 2 by 3 as in the axiomatic method, nor do we have a sequence of low level instructions running on virtual machine as in the operational method, but what we do have is a *denotation* for the program as a program state in which p is 6 and n is 0. Moreover, we have arrived at that point via a sequence of states that represents the execution sequence of the program's statements. If we wished we could replace the initial values of x and y with placeholders for any value (e.g. a and b). We could still do the derivation, but we would not be able to simplify the store expressions involving arithmetic, which would be left with expressions involving a and b. For example, take the assignments p = p + y; n = n - 1 in the store $s[x \mapsto a][y \mapsto b][p \mapsto 0][n \mapsto a]$. This gives a new store: $s[x \mapsto a][y \mapsto b][p \mapsto b][n \mapsto a - 1]$. The next iteration gives $s[x \mapsto a][y \mapsto b][p \mapsto b + b][n \mapsto a - 2]$. Clearly, without knowing the value of a we cannot terminate the program, but with some advanced techniques we can actually show that the final store is $s[x \mapsto a][y \mapsto b][p \mapsto b * a][n \mapsto 0]$. We can therefore prove that the program terminates with p = a * b, part of the original specification in the axiomatic method.