COMP 317: Semantics of Programming Languages

Denotational Semantics

Denotational semantics describes the meaning of a program as some mathematical object: this is the program's *denotation*. Typically, the denotation of a program is a function that maps states to states. Given a program *P*, its denotation, [[*P*]], is the function that takes a state *s* as argument and maps it to the state that results from running *P* in the state *s*, provided that the program *P* terminates in the state *s*.

The syntax of our language is described by the following BNF definitions:

<Var> ::= 'a | 'b | ... | 'z | 'A | 'B | ... | 'aa | ...

<Num> ::= *see the first hand-out for how to define numerals*

<Expression> ::= <Var> | <Num> | - <Expression> |
<Expression> + <Expression> |
<Expression> * <Expression>

<BooleanExpression> ::= <Expression> < <Expression> |
<Expression> == <Expression> |
true | false | ! <BooleanExpression> |
<BooleanExpression> && <BooleanExpression> |
<BooleanExpression> || <BooleanExpression>

<Program> ::= skip | <Var> := <Expression>; |
<Program> <Program> |
if (<BooleanExpression>) {<Program>} else {<Program>} |
while (<BooleanExpression>) {<Program>}

We first give a denotational semantics for arithmetic expressions, then for Boolean expressions, and finally for programs. Before all that, we first define what we mean by a **state**.

State

In any imperative language, assignment is the most basic form of program: the other linguistic constructs (conditionals and loops) are simply ways of organising assignments into series that are executed in a particular order. For example, think of how (according to our intuitions) the following program is evaluated:

```
'x := 0;
'f := 1;
while ('x <= 2) {
    'x := 'x + 1;
```

```
        'f := 'f * 'x;
    }
```

On each pass through the loop, the value of `'x` is increased by 1, and the body of the loop is gone through three times (while `'x` has values `0`, `1`, then `2`). Thus, we could "unfold" the program above to the following series of assignments:

```
'x := 0;
'f := 1;
'x := 'x + 1;
'f := 'f * 'x;
'x := 'x + 1;
'f := 'f * 'x;
'x := 'x + 1;
'f := 'f * 'x;
```

(one of the benefits of a formal semantics is that we could give a rigorous argument that these two programs are indeed equivalent).

Assignments, therefore, lie at the heart of imperative programming languages. They also lie at the heart of the semantics of imperative languages. In particular, they suggest that any semantics should be based on the notion of *state* (or *storage*), by which we mean the particular values that are associated with a program's variables. When a program is being executed, we can think of the computer that is running the program as being in a certain state. This state is determined by the values stored by the program's variables, and these values are updated by assignments to the variables. For example, after running the program above, the computer will be in a state where the variable `'x` has the value `3`, and the variable `'f` has the value `6`. We can think of a state as being a table that tells us the value associated with any given variable (in our language, variables aren't declared, and have no scope, so the state should tell us the value associated with *any* variable. This would give us an infinitely long table, and it is more convenient to think of a state as a function that takes a variable as argument and returns the value stored in that variable. More formally, a state is a function

```
Var  ->  Int
```

where `Var` is the set of variable names, and `Int` is the set of numbers (in our language, all variables store numbers). For example, after running a program

```
'x := 8;
'y := 'x + 1;
'z := 'y + 2;
```

we obtain a state *s*, with

- *s(*`'x`*) = 8*,
- *s(*`'y`*) = 9*, and
- *s(*`'z`*) = 11*.

(We don't know what values *s* gives to variables other than `'x`, `'y` or `'z`, but presumably those values aren't changed by the program.)

The semantics we will give will describe how assignments (and other programs) update states. Before we do this, note that an assignment like

```
'y := 'x + 1;
```

depends on the state of the computer before the assignment is executed: the value assigned to `'y` depends on the value of `'x` in this "initial" state (e.g., if the value of `'x` in the initial state is *8*, then `'y` is assigned the value *9*, and if `'x` has the value *23*, then `'y` is assigned the value *24*).

The Denotational Semantics of Arithmetic Expressions

Given an arithmetic expression such as

```
2 * ('x + 'y)
```

we want to describe its denotation as a number, but clearly the value of such an expression depends on the values stored in the variables `'x` and `'y`. In other words, the value of an arithmetic expression depends on the state of the computer when the expression is evaluated. We therefore describe the denotation of an arithmetic expression *e* as a function [[*e*]] : *State -> Int*. This function is defined inductively as follows:

1. For a number *n*, clearly the value should just be *n*, independent of the state:
   $$[\![n]\!](s) = n.$$
   (more precisely, we should distinguish between numerals and numbers: see the first hand-out).

2. For a variable *x*, the value should simply be the value in the given state:
   $$[\![x]\!](s) = s(x).$$

3. When *e* has the form *e1* + *e2*, the value should be the sum of the values of *e1* and *e2*:
   $$[\![e1 + e2]\!](s) = [\![e1]\!](s) + [\![e2]\!](s).$$

4. The operations for "unary minus" and multiplication are treated similarly.

The Denotational Semantics of Boolean Expressions

This is left as an exercise!

The Denotational Semantics of Programs

The denotational semantics for programs is given by defining, for a program *P*, its denotation [[*P*]]. From our intuitive understanding of programs (strengthened by having seen the operational semantics), we know that executiong a program has the effect of updating the state in which execution of the program begins. Thus, the denotation [[*P*]] will be a function that takes a state *s* as argument and returns the state that results from executing *P* in the state *s*. Of course, there is the possibility that *P* fails to terminate when it is executed in *s*, in which case [[*P*]]*(s)* shouldn't return any state: in other words, [[*P*]]*(s)* is undefined. This means that, in general, [[*P*]] is a *partial* function from states to states. (The denotation functions [[*e*]] and [[*t*]] for expressions and tests are both total functions from states to numbers and truth-values, respectively, as evaluating arithmetic and Boolean

expressions always terminates.)

The function [[*P*]] is defined inductively as follows:

1. [[`skip`]]*(s)* = *s*

2. [[*x* `:=` *e*;]]*(s)* = *s*[*x* <- *n*],
   where *n* = [[*e*]]*(s)*.

3. [[*P1 P2*]]*(s)* = [[*P2*]]*(* [[*P1*]]*(s))*.

4. [[`if` (*T*) {*P1*} `else` {*P2*}]]*(s)* = [[*P1*]]*(s)*
   if [[*T*]]*(s)* = *true*.

5. [[`if` (*T*) {*P1*} `else` {*P2*}]]*(s)* = [[*P2*]]*(s)*
   if [[*T*]]*(s)* = *false*.

6. [[`while` (*T*) {*P*}]]*(s)* = *s*,
   if [[*T*]]*(s)* = *false*.

7. [[`while` (*T*) {*P*}]]*(s)* = [[`while` (*T*) {*P*}]]*(* [[*P*]]*(s))*,
   if [[*T*]]*(s)* = *true*.

Exercises

1. Give a denotational semantics for Boolean expressions by defining inductively the function [[*T*]] that maps states to truth values (define this function by induction on the structure of the Boolean expression *T*).

2. Most languages have an `if-then` construct (without an `else`). Extend the BNF definition with an `if-then` construct, and give a semantics for it (i.e., extend the inductive definition of the [[_]] denotation function). Show that any program of the form `if` (*T*) {*P*} is semantically equivalent to `if` (*T*) {*P*} `else` {`skip`}. I.e., show that for all states *S*, [[`if` (*T*) {*P*}]](*S*) = [[`if` (*T*) {*P*} `else` {`skip`}]](*S*).

3. Some languages (such as the OO language Eiffel) have "assertions": these are commands that assert that some condition holds. If the condition does hold, then the assertion has no effect, and computation proceeds to the next following command; if the condition does not hold, then the program crashes (and for the sake of this exercise, this can be considered to have the same effect as a non-terminating loop). The conditions that are asserted are just Boolean expressions (<BooleanExpression>s), and an assertion could be written as: `assert` *T* for some Boolean expression *T*. Add assertions to our language by specifying their syntax in BNF, and by giving them a denotational semantics.

4. One way of modelling side-effects in expressions is to introduce a `do-return`-construct for expressions (it could also be used for describing the semantics of subroutines or methods with non-void return types). `do-return`-expressions are of the form `do` *P* `return` *E*, where *P* is a program, and *E* is an expression (and the whole thing is an expression, not a program). Such an expression is evaluated as follows: the program *P* is executed, giving a new state; the expression *E* is then evaluated in that state, and that gives the result of the whole `return`-expression. Any further computation proceeds in the new state. For example, after executing the

program

```
'z := do 'x := 1; 'y := 2 * 'x; return 'y + 'x;
'a := 'y;
```

the variable `'a` has the value 2, and the variable `'z` has the value 3. Specify the syntax and semantics of our programming language extended with `do-return` expressions.
Hint: the denotation of an expression is a partial function that takes a store and (if the program terminates) returns a pair consisting of an updated store and a number. When we actually write down the definitions, we need some way of "splitting up" pairs; "where" is useful, e.g.:

$$[\![E1 + E2]\!](S) = (S'', n1 + n2) ,$$

where $(S', n1) = [\![E1]\!](S)$
and $(S'', n2) = [\![E2]\!](S')$.

In other words, first evaluate *E1* to get its updated store S' and its value *n1*, and then evaluate *E2* in the updated store S', to get an updated store S'' and value *n2*; the end result is the pair consisting of the twice-updated store S'' and the number *n1* + *n2*.

Grant Malcolm