Operational semantics

We will show the operational semantics for a simple language which is defined by the following abstract syntax:

P ::= S
S ::= I = E | S_1;S_2 | if B then S_1 else S_2 end | while B do S end
B ::= true | false | 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 whole program, S is a statement, E is an arithmetic expression, B is a Boolean expression, I is an identifier, and N is the category for integers. This is not a complete language, but it does possess some arithmetic for calculations and the structured programming constructs of sequence, conditional and iteration (while). So it can actually do some simple jobs. For instance, since it does not possess multiplication, we need to add this by writing an algorithm for repeated addition. This program is:

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

This program multiplies x by y and places the result in p. At the end, n is zero.

The interpreter and Virtual Machine

Now we need to define an interpreter for this language and the corresponding virtual machine. The most important part of this is to model the memory that contains the variables. Let us call this memory M, and give it two operations lookup and update. Lookup takes an identifier as one argument and returns the value bound to the variable (we will assume the variable is already present.) Update takes an identifier and a value and binds the identifier to the value in the memory. For instance update(M, x, 0) will put x = 0 in M, and then lookup(M, x) returns 0. How we implement these operations, and the data structure to support them is left out here. Essentially we treat memory as an abstract data type. A real interpreter would have to make real choices as to the implementation of the ADT.

The interpreter is an overloaded function "interpret" that takes the syntactic forms and the memory M as arguments, and calls the appropriate routines to get the job done. It returns the changed memory when it is done. Along the way we will also define two expression evaluators: "arith" and "bool" for the two kinds of expression. They are also overloaded for the various types of expression, and call built-in functions for arithmetic (add and sub, that operate on integers) and for the Boolean operations (equals, gt, lt, or, and, not, that operate on the Boolean values T and F). Note that these could be assumed primitive on the virtual machine, or could be translated into a lower level set of instructions on a more primitive virtual machine. The interpreter functions are:

interpret(I = E, M) = update(M, I, arith(E, M))
interpret(S_1;S_2, M) = interpret(S_2, interpret(S_1, M))
interpret(if B then S_1 else S_2 end, M) = if bool(B, M) is T then interpret(S_1, M)
else interpret(S_2, M)
interpret(while B do S end, M) = if bool(B, M) is T then interpret(S;while B do S end, M)
else M

arith(I, M) = lookup(I, M)
arith(N, M) = N
arith(E_1 + E_2, M) = add(arith(E_1, M), arith(E_2, M))
arith(E_1 - E_2, M) = sub(arith(E_1, M), arith(E_2, M))

bool(true, M) = T
bool(false, M) = F
bool(I == E, M) = equals(lookup(I, M) , arith(E, M))
bool(I > E, M) = gt(lookup(I, M) , arith(E, M))
bool(I < E, M) = lt(lookup(I, M) , arith(E, M))
bool(B_1 or B_2, M) = or(bool(B_1, M), bool(B_2, M))
bool(B_1 and B_2, M) = and(bool(B_1, M), bool(B_2, M))
bool(not B, M) = not(bool(B, M))

The operational semantics for the program above can thus be expressed as a series of calls to the interpreter. We will need an initial memory M_0 which contains values for x and y. The portion to be expanded is underlined.

interpret(p=0;n=x;while n>0 do p=p+y;n=n-1 end, M_0)
⇒ interpret(n=x;while n>0 do p=p+y;n=n-1 end, $\underline{\text{interpret(p=0, }M_0\text{)}}$)
⇒ interpret(n=x;while n>0 do p=p+y;n=n-1 end, $\underline{\text{update(}M_0\text{, p, arith(0, }M_0\text{))}}$)
⇒ interpret(n=x;while n>0 do p=p+y;n=n-1 end, M_1), where M_1 contains p = 0
⇒ interpret(while n>0 do p=p+y;n=n-1 end, $\underline{\text{interpret(n=x, }M_1\text{)}}$)
⇒ interpret(while n>0 do p=p+y;n=n-1 end, update(M_1, n, $\underline{\text{arith(x, }M_1\text{)}}$))
⇒ interpret(while n>0 do p=p+y;n=n-1 end, update(M_1, n, lookup(x, M_1)))
⇒ interpret($\underline{\text{while n>0 do p=p+y;n=n-1 end}}$, M_2), where M_2 contains n = x
⇒ if bool(n>0, M_2) is T then interpret(p=p+y;n+n-1;while N>0 do p=p+y;n+n-1 end, M_2)
else M_2

and so on. At each stage we “run” the appropriate VM code for the syntactic form. If our interpreter is defined correctly, we will end up with a final memory that contains the value of x * y in p and n will contain 0. It is even feasible to prove that the memory will contain x * y in p, but this is much better done in the axiomatic and denotational methods.

Summary

Our virtual machine contains an ADT for memory, called M, with two operations update and lookup. Primitive values are integers and Booleans. Built-in operations on this VM are add and sub for arithmetic, and equals, gt, lt, or, and, not for Booleans. Control structures are function call (including recursion), and if-then-else. Note that we don't need assignment (although we could have it if we wanted) since we are using functions calling other functions as the basic idea.