OPERATIONAL SEMANTICS

The style of operational semantics we shall study is that developed by Plotkin and Kahn (who called it natural semantics). The language is defined by a set of inference rule. A computation is then defined by a finite proof tree built from the inference rules. Each rule is developed and applied by examining the syntax of the program and using the appropriate one. Thus the *semantics* of the language are devloped from the *syntax* by choosing the correct sequence of inference rules.

BIG-STEP SEMANTICS

We are going to use inference rules for whole syntactic forms, such as expressions, statements etc. Each rule will be a big step, in contrast with a compiler's small steps involving instructions on a real machine. This style of semantics is also possible, but we will not use it here. In order to handle variables and their values, we will need a finite function called *store* that maps variable identifiers to values. We will meet this concept again in the denotational style of semantics. Any program has a finite number of identifiers, so the function is finite. We will write it as:

$[\mathrm{x} \mapsto v_1, \mathrm{y} \mapsto v_2 \ldots]$

where x and y are identifiers (i.e. part of the syntax of the program) and v_1 and v_2 are values taken from some set (possibly infinite) of values. This store of values is always present in any inference rule, since it is assumed to be true at all times. We could write:

$$[\mathrm{x} \mapsto 1, \mathrm{y} \mapsto 2] \vdash \mathrm{x{+}y} \to 3$$

where $\vdash$ means if the expression on the right-hand side is true, then the expression on the right-hand side follows as true also. The whole thing then reads "in the case that x maps to 1 and y maps to 2 in the store, then it follows that x+y evaluates to 3". We will miss out the "turnstile" and its left-hand side, assuming that the store function is always present.

INFERENCE RULES

The semantics of a language will be presented as a set of inferences rules on the values contained in ths store, and the particular language form in question. Each rules has the following form:

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_n}{C}\left(\text{side conditions}\right)$$

The *Ps* are *judgements* (premises or assumptions) and *C* is a single judgement (conclusion). If the premises are true (assumed or proved), then the conclusion is also true. If there are no premises, then the rule is an *axiom*:

$$\frac{\quad}{C}$$

The side conditions often express constraints that the language of premises and conlcusions (i.e. logic) cannot. They also must be true for the rule to be applicable.

DERIVATIONS OR PROOF TREES

A *derivation* or *proof tree* whose root is a conclusion and whose subtrees are derivations of its premises. Each subtree is handled similarly. Eventually the leaves of the tree are either empty (as they are in axioms) or assumptions for the whole tree. Ecah of these derivations must be finite. This will be especially important in the case of programs with loops. A typical proof tree might be:

$$\frac{\dfrac{P_1 \quad P_2}{C_3} \qquad P_4 \qquad \dfrac{P_5 \quad \dfrac{P_6 \quad P_7}{C_8} \quad P_9}{C_{10}}}{C_{11}}$$

C_{11} is the overall conclusion that depends on three subtrees. The left subtree is a conclusion C_3 that depends on premises P_1 and P_2, the middle subtree is just a premise P_4, and the right subtree has a conclusion C_{10} that, in turn, depends on P_5, P_9 and a conclusion C_8 that depends on P_6 and P_7. Such a proof tree will express the meaning of a program and also act to show how the program produces a final result, given a set of initial values of the variables in the store.

GETTING THE STORE INTO THE RULES

The store represents the program state, or the *context* in which expressions are evaluated (and commands executed). Evaluation (or execution) will be represented as a relation on the set $(\text{Syntax} \times \mathit{Store}) \times \mathit{Value}$, i.e the set of pairs which have the set of pairs consisting of a piece of syntax and the store function as its left-hand part, and a value as its right. The syntax comes from the abstract syntax of the language, and every alternative on the right hand side of every rule will needs an inference rule.

INFERENCE RULES FOR EXPRESSIONS

The simplest rule is an axiom for c**onstants**, which evaluate to the values they denote:

$$\frac{}{[\mathrm{N},\sigma] \to n} \quad (n = \text{the value of N})$$

Since there is nothing above the line, there are no conditions on the truth of the relational statement on the bottom. This is read as N evaluates to n in the store σ. We can omit the line in this case:

$$[\mathrm{N},\sigma] \to n\,(n = \text{the value of N})$$

Technically N is a numeral (i.e. syntax) whereas *n* is a value (i.e. semantics). The "side" condition in parentheses is the way we shall add extra meaning to the inference rule that cannot be expressed in the rule itself.

Next are **simple variables**:

$$\frac{}{[\mathrm{I},\sigma] \to \sigma(\mathrm{I})}$$

Again, since there is nothing above the line, there are no conditions on the truth of the bottom line. It is always true that x evaluates to *store*(I).

Arithmetic expressions follow naturally:

$$\frac{[\mathrm{E}_1,\sigma]\rightarrow u \quad [\mathrm{E}_2,\sigma]\rightarrow v}{[\mathrm{E}_1 \text{ plus } \mathrm{E}_2,\sigma]\rightarrow u+v}$$

Note the distinction between syntax (E_1 plus E_2) and semantics ($u+v$). We are explaining the meaning of the plus operator in terms of the addition operation on values, denoted as $+$ in the rule. We can do the same thing for the other arithmetic operators.

Relational expressions are similar, but evaluate to true or false. We need two rules for one. E.g. for equality:

$$\frac{[\mathrm{E}_1,\sigma]\rightarrow u \quad [\mathrm{E}_2,\sigma]\rightarrow u}{[\mathrm{E}_1 \text{ equal } \mathrm{E}_2,\sigma]\rightarrow \mathit{true}}(u=v) \text{ and } \frac{[\mathrm{E}_1,\sigma]\rightarrow u \quad [\mathrm{E}_2,\sigma]\rightarrow v}{[\mathrm{E}_1 \text{ equal } \mathrm{E}_2,\sigma]\rightarrow \mathit{false}}(u\neq v)$$

The rules for greater than and less than are very similar:

$$\frac{[\mathrm{E}_1,\sigma]\rightarrow u \quad [\mathrm{E}_2,\sigma]\rightarrow v}{[\mathrm{E}_1 \text{ greater } \mathrm{E}_2,\sigma]\rightarrow \mathit{true}}(u>v) \text{ and } \frac{[\mathrm{E}_1,\sigma]\rightarrow u \quad [\mathrm{E}_2,\sigma]\rightarrow v}{[\mathrm{E}_1 \text{ greater } \mathrm{E}_2,\sigma]\rightarrow \mathit{false}}(u\leq v)$$

and similarly for less than.

INFERENCE RULES FOR COMMANDS

Commands (or statements) are not evaluated like expressions, but are executed. What does this mean in the context of operational semantics? The answer will be used again in denotational semantics: what a command does is two really two things. First, it has a side a effect; it changes program state. In Operational terms, it changes the store function. So the basic form of the evaluation realtion used for expressions is modified to include the store as the "value" of the command:

$$[\mathrm{C},\sigma]\rightarrow\sigma'$$

In other words, when a command is executed, it alters the store. The second thing that commands do is to take part in control, sequence, conditional and the loop. We can see this in the abstract syntax:

```
C ::= nop | I := E | C1;C2 | if B then C1 else C2 end | while B do C end
```

The second alternative is the state-altering form, assignment. The third is sequencing of commands (the recursion gives us any number of commands in a sequence). The fourth is the conditional and the fifth is the loop. All of these potentially alter the state. The first alternative, the no-operation, does not, of course. Let us start there. The inference rule for nop is:

$$[\mathrm{nop},\sigma]\rightarrow\sigma$$

It clearly does nothing.

Assignment needs the function update form:

$$\frac{[\mathrm{E},\sigma]\to u}{[\mathrm{I{:=}E},\sigma]\to[\mathrm{I}\mapsto u]\sigma}$$

The result of carrying out an assignment of I to the value of E, in the store σ, is a new function in which I maps to *u*, theresult of evaluating E in the same store. If I already maps to some value in σ, then the new value overrides that, as demonstrated in the section on updating functions. This is precisely what we need for the semantics of assignment. We have used the power of mathematical abstractions to model the operation of assignment.

The sequence has the rule:

$$\frac{[\mathrm{C}_1,\sigma]\to\sigma' \quad [\mathrm{C}_2,\sigma']\to\sigma''}{[\mathrm{C}_1;\mathrm{C}_2,\sigma]\to\sigma''}$$

This rule shows how the result of executing one statement serves as the store for executing the next in the sequence. The overall result is that the sequence C_1;C_2 acts just like a single command, albeit a compound.

The conditional, as expected, needs two rules:

$$\frac{[\mathrm{B},\sigma]\to \mathit{true} \quad [\mathrm{C}_1,\sigma]\to\sigma'}{[\text{if B then C}_1\text{ else C}_2\text{ end}]\to\sigma'} \text{ and } \frac{[\mathrm{B},\sigma]\to \mathit{false} \quad [\mathrm{C}_2,\sigma]\to\sigma'}{[\text{if B then C}_1\text{ else C}_2\text{ end}]\to\sigma'}$$

These rules show that only one of the two commands C_1 and C_2 is executed, depending on the evaluation of the Boolean B. The resultant store is for the whole conditional form whichever store comes from the execution of the command.

We have left the loop rule until last because it is the most interesting, and the most problematic. None of the languages we have looked at for representation (logic, sets, functions etc.) have the notion of looping in them. We are obliged, as we will be in denotational semantics, to use recursion to handle looping. There is one form of recursion, however, that we do have and that is in the abstract syntax. We can use this to good effect here. Essentially we will say that executing a while loop when the exit condition is true is just like execting the body of the loop and then doing the whole thing again; i.e. while B do C end is (if B is true) C;while B do C end. When the condition becomes false, the body is not executed, so there is no change. The rule follows:

$$\frac{[\mathrm{B},\sigma]\to \mathit{true} \quad [\text{C;while B do C end},\sigma]\to\sigma'}{[\text{while B do C end},\sigma]\to\sigma'}$$

Actually it is easier to write it out with three premises since the sequence will have to be expanded anyway.

$$\frac{[\mathrm{B},\sigma]\to \mathit{true} \quad [\mathrm{C},\sigma]\to\sigma'' \quad [\text{while B do C end},\sigma'']\to\sigma'}{[\text{while B do C end},\sigma]\to\sigma'}$$

The rule when B is false is:

$$\frac{[\mathrm{B},\sigma]\to \mathit{false}}{[\text{while B do C end},\sigma]\to\sigma}$$

showing that the store is unchanged.

EXAMPLE DERIVATION FOR A WHOLE PROGRAM

We will show the inference rules at work for the following simple program that multiplies two numbers by reapeated addition. We will meet this little program again, which, although it is unnecessary in a language that already has multiplication, it is simple enough to show all the parts of the method at work. The program is:

```
X := 2;
Y := 3;
M := 0;
while X greater 0 do
   M := M plus Y;
   X := X minus 1
end
```

We would like to show that at the end of execution, M has the value 6 and X has the value 0, because the loop has terminated.

Since the three initial assignments are simple, we have the following:

$$\dfrac{[2,\sigma_0]\to 2}{[\text{X:=2},\sigma_0]\to[\text{X}\mapsto 2]\sigma_0},\ \dfrac{[3,\sigma_1]\to 3}{[\text{Y:=3},\sigma_1]\to[\text{Y}\mapsto 3]\sigma_1},\ \dfrac{[0,\sigma_2]\to 0}{[\text{M:=0},\sigma_2]\to[\text{M}\mapsto 0]\sigma_2}$$

We need to use the sequence rule to put these together:

$$\dfrac{\dfrac{\dfrac{[2,\sigma_0]\to 2}{[\text{X:=2},\sigma_0]\to[\text{X}\mapsto 2]\sigma_0}\quad\dfrac{[3,\sigma_1]\to 3}{[\text{Y:=3},\sigma_1]\to[\text{Y}\mapsto 3]\sigma_1}}{[\text{X:=2;Y:=3},\sigma_0]\to[\text{Y}\mapsto 3][\text{X}\mapsto 2]\sigma_0}\quad\dfrac{[0,\sigma_2]\to 0}{[\text{M:=0},\sigma_2]\to[\text{M}\mapsto 0]\sigma_2}}{[\text{X:=2;Y:=3;M:=0},\sigma_0]\to[\text{M}\mapsto 0][\text{Y}\mapsto 3][\text{X}\mapsto 2]\sigma_0}$$

since we must have $\sigma_1=[\text{X}\mapsto 2]\sigma_0$ and $\sigma_2=[\text{Y}\mapsto 3][\text{X}\mapsto 2]\sigma_0$ for the rules to work. σ_0 is the initial store which is empty.

Let us set $\sigma_{in}=[\text{M}\mapsto 0][\text{Y}\mapsto 3][\text{X}\mapsto 2]\sigma_0$, the store for the loop. The body can be handled first in a top-down fashion, but since we are going to have to execute it several times, we will make it generic:

$$\dfrac{\dfrac{\dfrac{[\text{M},\sigma_{in}]\to\sigma_{in}(\text{M})\quad[\text{Y},\sigma_{in}]\to\sigma_{in}(\text{Y})}{[\text{M plus Y},\sigma_{in}]\to\sigma_{in}(\text{M})+\sigma_{in}(\text{Y})}}{[\text{M:=M plus Y},\sigma_{in}]\to[\text{M}\mapsto\sigma_{in}(\text{M})+\sigma_{in}(\text{Y})]\sigma_{in}}\quad\dfrac{\dfrac{[\text{X},\sigma'_{in}]\to\sigma'_{in}(\text{X})\quad[1,\sigma'_{in}]\to 1}{[\text{X minus }1,\sigma'_{in}]\to\sigma'_{in}(\text{X})-1}}{[\text{X:=X minus }1,\sigma'_{in}]\to[\text{X}\mapsto\sigma'_{in}(\text{X})-1]\sigma'_{in}}}{[\text{M:=M plus Y;X:=X minus }1,\sigma_{in}]\to[\text{X}\mapsto\sigma'_{in}(\text{X})-1]\sigma'_{in}}$$

where $\sigma'_{in}=[\text{M}\mapsto\sigma_{in}(\text{M})+\sigma_{in}(\text{Y})]\sigma_{in}$.

The final store is then $\left[\mathrm{X} \mapsto \left(\left[\mathrm{M} \mapsto \sigma_{in}(\mathrm{M}) + \sigma_{in}(\mathrm{Y})\right]\sigma_{in}\right)(\mathrm{X}) - 1\right]\left[\mathrm{M} \mapsto \sigma_{in}(\mathrm{M}) + \sigma_{in}(\mathrm{Y})\right]\sigma_{in}$ in terms of the store going into the loop.

Unfortunately, the loop itself conccot be handled top-down, since, in general, we don't know how many times it will iterate. So we will build it bottom up, using stores that we will either calculate asa we go, or that we will have to leave until the end of the loop. The general scheme is:

$$\frac{[B,\sigma_{in}] \to \mathit{true} \quad [C,\sigma_{in}] \to \sigma'_{in} \quad \dfrac{[B,\sigma'_{in}] \to \mathit{true} \quad [C,\sigma'_{in}] \to \sigma''_{in} \quad \dfrac{\overset{[B,\sigma^{n}_{in}] \to \mathit{false}}{\vdots}}{[\text{while B do C end},\sigma''_{in}] \to \sigma_{out}}}{[\text{while B do C end},\sigma'_{in}] \to \sigma_{out}}}{[\text{while B do C end},\sigma_{in}] \to \sigma_{out}}$$

Eventually, as we move up the page, there will be no need for a premise with the body, since the Boolean expression will evaluate to false. Starting with the final conclusion, we know σ_{in}, so we can write the premises

$$\frac{[\mathrm{X},\sigma_{in}] \to 2 \quad [0,\sigma_{in}] \to 0}{[\text{X greater } 0,\sigma_{in}] \to \mathit{true}}(2 > 0)) \text{ and } [\text{M:=M+Y;X:=X-1},\sigma_{in}] \to \sigma_3$$

where $\sigma'_{in} = [\mathrm{X} \mapsto 1][\mathrm{M} \mapsto 3]\sigma_{in}$ using the formula we derived for executing the body.

Now we can do the whole loop again to calualate σ''_{in}, since $[\text{X greater } 0,\sigma'_{\text{in}}] \to \mathit{true}(1>0)$. Executing the body gives us $\sigma''_{in} = [\mathrm{X} \mapsto 0][\mathrm{M} \mapsto 6]\sigma'_{in}$. Now we have $[\text{X greater } 0,\sigma''_{in}] \to \mathit{false}$, so we can use the exit form of the while rule, and the derivation , or proof, is finished.

SUMMARY

We have achieved two things here. One is that we can use this technique to derive the results of a computation purely by applying logical rules through matching of syntactc forms. We do not need a real machine, and we do not need to translate our language into another one, or build a virtual machine in any language. The other is that we have *proved* that the program produces the result that it does since our methods are based on well-understood and reliable techniques from mathematics. A nice side result is that we have also proved that the program terminates, at least in this case. Showing that the program terminates for all positive X and Y is harder. We will see how to do that when we study axiomatic semantics.