A SIMPLE IMPERATIVE LANGUAGE

Eventually we will present the semantics of a full-blown language, with declarations, types and looping. However, there are many complications, so we will build up slowly. Our first version will be language like the example used in the syntax section, but without a while loop. It has only one type (natural numbers) and simple arithmetic. However, as a half-way house to the full language, we will add a primitive command whose sole job is to stand in for a non-terminating loop (whose semantics we will examine later). This will be called \texttt{diverge}. The full syntax is:

\[
\begin{align*}
P &\in \text{Program} \\
C &\in \text{Command} \\
E &\in \text{Expression} \\
B &\in \text{Boolean-expr} \\
I &\in \text{Identifier} \\
N &\in \text{Numeral} \\
P &= C \\
C &= C_1 ; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid I := E \mid \text{diverge} \\
E &= E_1 + E_2 \mid I \mid N \\
B &= E_1 = E_2 \mid \neg B
\end{align*}
\]

THE STORE FUNCTION

Unlike the calculator, this language needs an unbounded store, which can represent the bindings of identifiers to values (which are just natural numbers). Just as in operational semantics, we will call on the \textsl{store} function to do this job. The store will be a semantic domain based on the function that maps identifiers to numbers:

\[
\text{Store} = \text{Id} \rightarrow \text{Nat}
\]

where \textsl{Id} is the (primitive) domain of identifiers, and \textsl{Nat} is the usual domain of natural numbers. We will also need the Boolean domain, as before. The store will be defined more accurately then we did in operational semantics as a semantic algebra:

\[
\text{Domain } \text{Store} = \text{Id} \rightarrow \text{Nat} \\
\text{Operations} \\
\text{newstore: } \text{Store} \\
\text{newstore} = \lambda i. \text{zero} \\
\text{access: } \text{Id} \rightarrow \text{Store} \rightarrow \text{Nat} \\
\text{access} = \lambda i. \lambda s. s(i) \\
\text{update: } \text{Id} \rightarrow \text{Nat} \rightarrow \text{Store} \rightarrow \text{Store} \\
\text{update} = \lambda i. \lambda n. \lambda s[i \mapsto n] s
\]

The \texttt{newstore} operation is a constant function that always returns zero given any identifier as argument. \texttt{access} simply uses the function that is passed to return the value bound to the identifier argument. \texttt{update} returns a new function updated the new mapping of identifier to value. These are all presented as lambda functions so that the order of the arguments can be shown. That is why the functionality of each functions is shown in curried form.

NON-TERMINATING COMMANDS

The store will clearly be the main semantic argument in the evaluation of a command:

\[
C : \text{Command} \rightarrow \text{Store} \rightarrow \text{Store}
\]
Since every one of the command forms will bottom out in an assignment, the store will change when a command is executed. However, what happens if we execute diverge? We don’t want the program to continue after that point, because we want the semantics to be like that of an infinite loop. The solution is called “lifting” a domain. We will add a special value to the store domain called “bottom”, written as ⊥. It will represent a store that is locked, unchangeable and inaccessible. In general, a lifted domain is one which has bottom added to it. i.e. for any domain A, $A_\bot = A \cup \{\bot\}$. This changes the behavior of functions that map lifted domains:

For a function with functionality $A_\bot \rightarrow B_\bot$, we can define a strict function by:

$$\lambda x.e : A_\bot \rightarrow B_\bot$$

$$(\lambda x.e) \bot = \bot$$

$$(\lambda x.e) a = [x \leftarrow a] e, \text{for all } a \neq \bot$$

In other words, if a strict function receives bottom, it immediately returns bottom; in all other cases it simply does its normal job.

A non-strict function does not check for bottom, but simply carries on:

$$\lambda x.e : A_\bot \rightarrow B_\bot$$

$$(\lambda x.e) a = [x \leftarrow a] e, \text{for all } a$$

As a simple example, let’s apply first a strict function, and then a non-strict function to an argument that is itself an application.

$$(\lambda x.\text{zero})((\lambda y.\text{one}) \bot)$$

$$=(\lambda x.\text{zero}) \bot$$

$$=\bot$$

using the rule for strictness twice. On the other hand,

$$(\lambda x.\text{zero})((\lambda y.\text{one}) \bot)$$

$$= \text{zero}$$

We do not need to reduce the argument, because the function being applied is a constant function that always returns zero regardless of the argument. We thus have a method of applying a strict function that is best done by reducing the argument before applying the function. This is the applicative method of expression reduction. We can make this even plainer by using the ‘let’ form:

(let $x = e_1$ in $e_2$) is an abbreviation for $(\lambda x.e_2)e_1$

So we will use the let form to put strict function applications into a more readable form.

Revisiting the valuation function for commands, it will actually operate on a lifted store so that we can use strict functions to effectively prevent any further execution of commands.

$$C : \text{Command} \rightarrow \text{Store}_\bot \rightarrow \text{Store}_\bot$$

**SEMANTIC ALGEBRAS**

We will not repeat the semantic algebras for the Booleans and the natural numbers. The Store domain has been defined above. The only new domain is one for identifiers. We will distinguish between the semantics domain $Id$ and the set Identifier, but they are basically identical, and there are no operations, it is just a set of constant objects with names $x, y, z$ etc.

**VALUATION FUNCTIONS**

We will proceed bottom-up again to leave the most complex for last. Most of these are the same as we have seen already in the calculator.
There is not much to say here. The set $\mathbb{N}$ is primitive.

$$B : \text{Boolean-expr} \rightarrow \text{Store} \rightarrow \text{Tr}$$

$$B[E_1 + E_2] = \lambda s. E[E_1]s$$

$$B[-B] = \lambda s. \text{not}(B[B][s])$$

Again, this is straightforward, except that we are expressing the right-hand side as a lambda function with one argument, as $\text{Store}$.

$$E : \text{Expression} \rightarrow \text{Store} \rightarrow \text{Tr}$$

$$E[E_1 + E_2] = \lambda s. E[E_1][s]$$

$$E[I] = \lambda s. \text{access}[I][s]$$

$$E[N] = \lambda s. N[N][s]$$

The first argument to $\text{access}$ is the identifier that corresponds to the syntax $I$. We signify this by writing it in the funny square brackets.

The valuation functions for commands are:

$$C: \text{Command} \rightarrow \text{Store}_\perp \rightarrow \text{Store}_\perp$$

$$C[C_1; C_2] = \lambda s. C[C_2][C[C_1][s]]$$

$$C[\text{if } B \text{ then } C] = \lambda s. B[B][s] \rightarrow C[C][s]$$

$$C[\text{if } B \text{ then } C_1 \text{ else } C_2] = \lambda s. B[B][s] \rightarrow C[C_1][s] \land C[C_2][s]$$

$$C[I := E] = \lambda s. \text{update}[I][E[E][s]]$$

$$C[\text{diverge}] = \lambda s. \perp$$

The first four are reminiscent of the operational rules, but instead of being indirect rules of inference (connected by an implication) these functions very directly given the denotation for each form. The functions are all strict because if at any point any of them gets a ‘improper’ store, i.e. bottom, then it immediately returns bottom and refuses to look at anything else. However, if, an if-then is being executed and is handed a good store (not bottom), then even when the then command returns bottom, if the test is false the store returned is the original good one. The assignment functions reduces to a call to update to actually return a changed store.

The interesting addition is the function for $\text{diverge}$. This guarantees to return an improper store (bottom) whatever it is given. It need not be strict, but is so for consistency purposes.

We finish off with the function for programs. Here we also add some interest by having the ability to pass a single number to a program, and to return a single number. This will serve as primitive input-output. Since there are no read or write commands, we must assign the input value to a variable in the semantics, and retrieve the final result from a variable. We will choose $A$ for the input and $Z$ for the output. These are arbitrary choices, not present in the syntax, but present only in the semantics.

$$P : \text{Program} \rightarrow \text{Nat} \rightarrow \text{Nat}_\perp$$

$$P[C] = \lambda n. \text{let } s = \left( \text{update}[A][n \text{ newstore}] \right) \text{ in }$$

$$\text{let } s' = C[C][s] \text{ in }$$

$$\text{access}[Z][s']$$

Note that the return domain is lifted since the result of executing the program may be bottom, representing a failed execution at some stage (i.e. a $\text{diverge}$ command). The use of the let form assures us that bottom will be handled correctly.
AN EXAMPLE DERIVATION

First, a derivation of a program with an input value will be examined. The program is:

\[
Z := 1;
\]

\[
\text{if } A = 0 \text{ then diverge;}
\]

\[
Z := 3
\]

The derivation starts as:

\[
P[Z := 1; \text{if } A = 0 \text{ then diverge; } Z := 3](\text{two})
\]

\[
= \text{let } s = \text{update}[A]\text{two newstore in}
\]

\[
\text{let } s' = C[Z := 1; \text{if } A = 0 \text{ then diverge; } Z := 3]s\text{ in}
\]

\[
\text{access}[Z]s'
\]

Let \(\text{update}[A]\text{two newstore} = \left[\left[\lambda i. \text{zero}\right] \mapsto \text{two}\right] = s_1\). So in the above expression,

Working on \(C[Z := 1]s_1\), we have

\[
C[Z := 1]s_1
\]

\[
= (\lambda s.\text{update}[Z](E[1]s)s)s_1
\]

\[
= \text{update}[Z](N[1])s_1
\]

\[
= \left[\left[Z \mapsto \text{one}\right]\right]s_1
\]

\[
= s_2
\]

This is the store for the rest of the program. So

\[
C[\text{if } A = 0 \text{ then diverge; } Z := 3]s_2
\]

\[
= (\lambda s. C[Z := 3](C[\text{if } A = 0 \text{ then diverge}]s))s_2
\]

\[
= C[Z := 3](C[\text{if } A = 0 \text{ then diverge}]s_2)
\]

\[
= C[Z := 3](\lambda s. B[A = 0]s \rightarrow C[\text{diverge}]s \sqcup s)s_2
\]

\[
= C[Z := 3](B[A = 0]s_2 \rightarrow C[\text{diverge}]s_2 \sqcup s_2)
\]

\[
C[\text{diverge}]s_2 = (\lambda s. \bot)s_2 \Rightarrow \bot, \text{ so we will get non-termination if the test is true. However}
\]

\[
B[A = 0]s_2 = (\lambda s. E[A]s \text{ equals } E[0]s)s_2
\]

\[
= E[A]s_2 \text{ equals } E[0]s_2
\]

\[
= \text{access } [A]s_2 \text{ equals zero}
\]

\[
= \left(\left[\left[Z \mapsto \text{one}\right][[A] \mapsto \text{two}]\text{newstore}\right]s_2\right)[A] \text{ equals zero}
\]

\[
= \text{two equals zero}
\]

\[
= \text{false}
\]

So, \text{diverge} is not executed. The last command is thus executed in the (unchanged) store \(s_2\):

\[
C[Z := 3]s_2
\]

\[
= \left[\left[Z \mapsto \text{three}\right]\right]s_2
\]

\[
= s_3
\]
The denotation for the whole program is then

\[
\text{access}[[Z]]s_3
\]

\[
= (([[Z]] \mapsto \text{three}) s_2) [[Z]]
\]

\[
= \text{three}
\]

which is a simple number, as we expected. However, if the input value is \text{zero}, we get

\[
s_4 = [[[A]] \mapsto \text{zero}] \text{newstore}
\]

and the conditional is

\[
\begin{align*}
\text{B} [[[A = 0]] s_4 & \rightarrow \text{C} [[[\text{diverge}]] s_4 \cdot s_4] \\
& = \text{true} \rightarrow \text{C} [[[\text{diverge}]] s_4 \cdot s_4] \\
& = (\lambda s. \bot) s_4 \\
& = \bot
\end{align*}
\]

So the store for \(Z := 3\) is improper, and then

\[
\text{C} [[[Z := 3]]] \bot
\]

\[
= (\lambda s. \text{update}[[Z]]([[E][[3]]] s)) \bot
\]

\(\bot\), because of strictness

So the update is never carried out. The final denotation is

\[
\text{let } s' = \bot \text{ in} \\
\text{access}[[Z]] s'
\]

\(\bot\), directly from the definition of \text{let}

**COMPILED DENOTATIONS**

We can do more by passing the program a general input \(n\), instead of an actual number. Of course, now we cannot reduce the denotation to a number; it must remains as a conditional. The denotation is:

\[
\lambda n. \text{let } s = \text{update}[[A]] n \text{ newstore in} \\
\left(\begin{array}{c}
\text{let } s'_1 = \text{update}[[Z]] \text{one } s \text{ in} \\
\text{let } s' = \left(\begin{array}{c}
\text{let } s'_2 = \left(\begin{array}{c}
\text{access}[[A]] s'_1 \text{ equals zero} \rightarrow (\lambda s. \bot) s'_1 \cdot s'_2 \text{ in} \\
\text{update}[[Z]] \text{three } s'_2
\end{array}\right) \text{ in}
\end{array}\right)
\end{array}\right) \text{ in}
\]

\[
\text{access}[[Z]] s'
\]

In this expression, all the syntax is gone (except for identifiers), but we are left with a function, which, if applied to two, will produce the denotation \text{three}, as before, and bottom if applied to \text{zero}. This resembles compiling, and hints that these denotations can be used to study how languages may be compiled for a particular virtual machine (or a real machine). Here the virtual machine is one that can evaluate lambda expressions by substitution.

One final step is to attempt to simplify this rather large expression. If we apply functions to proper stores, when we know them, and also use a transformation which is easy to prove by extensionality of functions:

\[
\text{let } s = (e_1 \rightarrow \bot \cdot e_2) \text{ in } e_3 \text{ is the same as } e_1 \rightarrow \bot \cdot [e_2 \setminus s] e_3
\]

then we can reduce the expression to

\[
\lambda n. n \text{ equals zero} \rightarrow \bot \cdot \text{three}
\]
which is intuitively correct. Here we have optimized all identifiers away, leaving only numbers.

PROVING PROGRAM EQUIVALENCE

Since the method produces static denotations, we can compare programs by comparing their denotations. So, the two programs \([X := 0; Y := X + 1]\), and \([Y := 1; X := 0]\) should be equivalent in that they end up with the same values in the store. We can prove this by deriving the denotation in each case, and then using the principle of extensionality of functions. The derivation of the first program yields a denotation of \(s_1 = [Y \mapsto one][X \mapsto \text{zero}]s\) for any store \(s\), and the second gives a denotation of \(s_2 = [X \mapsto \text{zero}][Y \mapsto \text{one}]s\) for the same store. There is no way to alter the first store into the second, or vice versa by any of the rules we have. We must prove that these two stores are the same function somehow. We do it by using the fact that if \(x\) is any argument and \(f x = g x\), then \(f\) and \(g\) are the same function. So if we can show that our two stores produce the same result for all arguments, then we have shown they are the same store. Clearly, there are two special cases: \([X]\), and \([Y]\).

\[
\begin{align*}
& s_1 ([X]) = \text{zero}, \text{ and } s_2 ([X]) = \text{zero} \\
& s_1 ([Y]) = \text{one}, \text{ and } s_2 ([Y]) = \text{one}
\end{align*}
\]

The last case is the “anything else” case:

\[
\begin{align*}
& s_1 ([I]) = s ([I]), \text{ and } s_2 ([I]) = s ([I]) \text{ for } I \text{ which is not } X \text{ nor } Y. \text{ Hence the equivalence is proved.}
\end{align*}
\]