

***Programming Language Concepts and Semantics
Part II***

***Lecture XIII
Imperatives***

In this lecture we will look at some fundamental concepts common in imperative programming languages: commands and updatable variables.

The lecture follows Chapter 7 of “Fundamental Concepts and Formal Semantics of Programming Languages, Lecture Notes by Peter Mosses. University of Aarhus”

Commands: Fundamental Concepts

A **command** $C:Cmd$ is a construct executed for its *effects* rather than to compute a value (concerned with flow of control).

ML expressions of `unit` type are similar to commands (which can be seen as computing a fixed null value). Same for C `void` procs.

Sequencing of commands can be binary ($Cmd := seq(Cmd, Cmd)$) or n-ary ($Cmd ::= seq(Cmd+)$) (note that $n > 0$). The empty sequence is represented by $Cmd ::= skip$. A sequence such as $seq(C1, \dots, Cn)$ determines the execution order of its commands.

An expression can be turned into a command ($Cmd ::= effect(Exp)$) by evaluating it (for its effects) and then discarding its value.

Sequencing a command and an expression results in an expression ($Exp ::= seq(Cmd, Exp) \mid seq(Exp, Cmd)$) whose value is determined by the value of its subexpression. The difference is whether this happens before or after the command is executed.

A **while** loop with boolean condition: $Cmd ::= while(Exp, Cmd)$. ML's while-expression `while E1 do E2` corresponds to $seq(while(E1, effect(E2)), tuple())$. Other loops (such as 'for' loops) can be obtained by combining other commands, but specific constructs are preferable.

Commands: Formal Semantics

Let us now introduce the MSOS modules that define the dynamic semantics of the previous abstract constructs.

Module *Cons/Cmd*:

State ::= Cmd

Final ::= skip

Cmd ::= skip

Since commands can be seen as computing the null value, the final state will be skip.

The rules for binary sequencing are very simple:

Module *Cons/Cmd/seq*:

$$C1 \text{ --}\{\dots\}\text{--> } C1'$$

seq(C1,C2):Cmd --{...}-> seq(C1',C2)

seq(skip,C2):Cmd ---> C2

Commands: Formal Semantics (Cont.)

Those for n-ary sequencing only slightly more complicated:

Module *Cons/Cmd/seq-n*:

$$\begin{aligned} C+ &= (C1, C2^*) \\ C1 \text{ --}\{\dots\}\text{--}> C1' \\ (C1', C2^*) &= C'+ \end{aligned}$$

 $\text{seq}(C+): \text{Cmd} \text{ --}\{\dots\}\text{--}> \text{seq}(C'+)$

$$C+ = (\text{skip}, C2+)$$

 $\text{seq}(C+): \text{Cmd} \text{ ---}> \text{seq}(C2+)$

$$\text{seq}(\text{skip}): \text{Cmd} \text{ ---}> \text{skip}$$

Note the need for $C+$ in the second rule above, due to the lack of a command of the form $\text{seq}()$ (recall that skip is used for that)

The rules for the other sequencing commands are very simple:

Module *Cons/Cmd/effect*:

$$E \text{ --}\{\dots\}\text{--}> E'$$

 $\text{effect}(E): \text{Cmd} \text{ --}\{\dots\}\text{--}> \text{effect}(E')$

$$\text{effect}(V): \text{Cmd} \text{ ---}> \text{skip}$$

Commands: Formal Semantics (Cont.)

Module *Cons/Cmd/seq-Cmd-Exp*:

$$C \text{ --}\{\dots\}\text{--> } C'$$

$$\text{seq}(C, E) : \text{Exp} \text{ --}\{\dots\}\text{--> } \text{seq}(C', E)$$

$$\text{seq}(\text{skip}, E) : \text{Exp} \text{ ---> } E$$

Module *Cons/Cmd/seq-Exp-Cmd*:

$$E \text{ --}\{\dots\}\text{--> } E'$$

$$\text{seq}(E, C) : \text{Exp} \text{ --}\{\dots\}\text{--> } \text{seq}(E', C)$$

$$C \text{ --}\{\dots\}\text{--> } C'$$

$$\text{seq}(V, C) : \text{Exp} \text{ --}\{\dots\}\text{--> } \text{seq}(V, C')$$

$$\text{seq}(V, \text{skip}) : \text{Exp} \text{ ---> } V$$

We could have substituted the last two rules for

$\text{seq}(V, C) : \text{Exp} \text{ --> } \text{seq}(C, V)$, but we prefer to define each construct independently.

Commands: Formal Semantics (Cont.)

For defining `while` we *must* however rely on other constructs:

Module *Cons/Cmd/while*:

see `Cmd/cond`, `Cmd/seq`

```
while(E,C):Cmd ---> cond(E,seq(C,while(E,C)),skip)
```

It is difficult (impossible?) to imagine a language with a `while` but no conditional construct. Also, sequentiality is mandatory to distinguish between one evaluation of `E` and the next.

Updatable Variables: Fundamental Concepts

A **variable VAR: Var** determines where a value is stored. Needs to be allocated: $\text{Var} ::= \text{alloc}(\text{Exp})$ where `alloc` obtains a new cell and initialises it to the value given by `Exp`. Note that it returns the cell, not the value itself.

A **cell CL:Cell** represents **simple** variables (which can only be assigned simple values). **Compound** variables can be assigned structured values and thus might have subvariables. They are used to represent data-structures.

The **value assigned** to a variable ($\text{Exp} ::= \text{assigned}(\text{Var})$) is stable between assignments.

For compound variables, computing `VAR` before evaluating the `Exp` might be important (if, for example, `VAR` contains expressions to compute the indices of array vars). For this

$\text{Exp} ::= \text{assign-seq}(\text{Var}, \text{Exp})$ first computes the cell for `Var`, then the storable value from `Exp`, and then adds the binding.

Identifiers are bound to cells, but are not cells themselves. Identifiers bound to the same cell are *aliased*. This can happen, for example, when the same variable is supplied twice as a parameter). Assigning a value to one, affects all other aliased ones.

We will assume cells are never recycled, and disregard whether they are stored in stack, a heap, etc.

Updatable Variables: Fundamental Concepts (Cont.)

Variables are represented by reference values in ML

A reference is itself a(n expression) value $\text{Exp} := \text{ref}(\text{Var})$. It needs to be explicitly dereferenced $\text{Var} := \text{deref}(\text{Exp})$ to obtain the variable component of the reference (thus allowing to inspect its stored value).

ML does not have syntactic constructs for allocating, assigning, and dereferencing of variables. It simply provides initial bindings for identifiers ref , $:=$ and $!$ which can then be using through the general application construct app .

Updatable Variables: Formal Semantics

Let us now introduce the MSOS modules that define the dynamic semantics of the previous abstract constructs.

Recall the reserved identifier $S:Store := (Cell, Storable)Map$. Whereas $Cell$ can be chosen freely, $Storable$ should include values which can be stored in simple variables. For ML this includes all expressible values. For other languages this might not be as simple.

Module *Cons/Var/alloc*:

Label = {store, store' : Store, ...}

$$E \text{ --}\{\dots\}\text{--> } E'$$

alloc(E) : Var --{...}--> alloc(E')

$$\text{new-cell}(S) = CL, \{CL \mid \text{-> } SV\} / S = S'$$

alloc(SV) : Var --{store=S, store'=S', ---}--> CL

where SV is the variable identifier for $Storable$, and the operation $\text{new-cell}(S)$ chooses any cell not in use in store S .

The last rule shows how to specify a label which is *almost* unobservable.

Updatable Variables: Formal Semantics (Cont.)

Module *Cons/Var/assign-seq*:

Label = {store,store':Store,...}

VAR --{...}-> VAR'

assign-seq(VAR,E):Exp --{...}-> assign-seq(VAR',E)

E --{...}-> E'

assign-seq(CL,E):Exp --{...}-> assign-seq(CL,E')

def lookup(CL,S), {CL|-> SV}/S = S'

assign-seq(CL,SV):Exp --{store=S,store'=S',---}-> SV

Module *Cons/Var/assigned*:

Label = {store,store':Store,...}

VAR --{...}-> VAR'

assigned(VAR):Exp --{...}-> assign-seq(VAR')

lookup(CL,S) = SV

assigned(CL):Exp --{store=S,store'=S,---}-> SV

Note how the last rule has an unobservable label.

Updatable Variables: Formal Semantics (Cont.)

Module *Cons/Exp/ref*:

Value ::= ref(Cell)

$$\text{VAR} \dashrightarrow \{\dots\} \rightarrow \text{VAR}'$$

ref(VAR) : Exp $\dashrightarrow \{\dots\} \rightarrow$ ref(VAR')

Module *Cons/Var/deref*:

Value ::= ref(Cell)

$$\text{E} \dashrightarrow \{\dots\} \rightarrow \text{E}'$$

deref(E) : Var $\dashrightarrow \{\dots\} \rightarrow$ deref(E')

deref(ref(CL)) : Var \dashrightarrow CL