CS611 Lecture 11 Modules and State 22 September 2004
Scribe: Krishnaprasad Vikram, David Crandall Lecturer: Andrew Myers

1 Introduction

The last lecture introduced the static and dynamic approaches to variable scoping. Both of these approaches are hierarchical in the sense that variables enter and leave scope according to the program's abstract syntax tree (in the case of static scoping) or the tree of function calls (in dynamic scoping). This dependence on hierarchy might prove restrictive or inflexible for writing certain kinds of programs. In such cases we might want more fine-grained access to variables that are defined in various parts of the program.

This can be done by introducing *non-hierarchical* scoping structures. The idea is that a large program is composed of a collection of software black-boxes, each of which exports resources as a set of names. In such a scheme, the names that can be used at a certain point in the program need not necessarily come *down* from *above* but can also come from various such black-boxes.

One issue with such a scoping scheme is the communication of names from one part of a program to another. One way in which this has traditionally been done is by having a global namespace as in FORTRAN or C. With a global namespace, the names of functions in source files and libraries are visible to all other parts of the program. There are certain problems with this:

- The various parts of the program can become tightly coupled. In other words, the global namespace approach does not enforce the modularity of the program. Replacing any particular part of the program with an enhanced equivalent could require a lot of effort.
- Name collisions could occur since names inevitably tend to coincide.

A solution to this problem is for the language to provide a *module mechanism* that allows similar functions, values, and types to be grouped together into a common context, as we shall see next.

2 Modules

A *module* is a collection of named things (such as values, functions, types etc.) that are somehow related to one another. The programmer can choose which names are *public* (exported to other parts of the program) and which are *private* (inaccessible outside the module).

There are usually two ways to access the contents of a module. The first is with the use of a *selector expression*, where the name of the module is prefixed to the variable in a certain way. For instance, we write m.x in Java and m::x in C++ to indicate a reference to name x within module m.

The second method of accessing module contents is to use an expression that brings names from a module into scope for a section of code. For example, we can write something like with m do e, which means that x can be used in block of code e without prefixing it with m. In ML, for instance, the command "Open List" brings names from the module List into scope. In C++ we write "using namespace module_name;" and in Java we write "import module_name;" for the similar purposes.

Another issue is whether to have modules as *first class* or *second class* objects. First class objects are entities that can be passed to a function as an argument, bound to a variable or returned from a function. In ML, modules are not first class objects whereas in Java, modules can be treated as first class objects using the *reflection mechanism*. While first-class treatment of modules increases the flexibility of a language, it usually requires some extra overhead at run-time.

3 Module syntax and translation to uML

We now extend uML, our simple ML-like language, to support modules. We call the new language $uML+M$ to denote that it supports modules. The syntax of the new language is:

$$
\begin{array}{lll}
e ::= & \ldots & \\
| & \mathsf{module}\ (x_1 = e_1, \ldots, x_n = e_n) & \textit{(module definition)} \\
| & e_m.e & \textit{(selector expression)} \\
| & \mathsf{with}\ e_m\ x & \textit{(bringing into scope)} \\
| & \mathsf{lookup\text{-}error} & \textit{(error)}
\end{array}
$$

We now want to define a translation from $uML + M$ to uML.[1] To do this, we notice that a module is really just an environment, since it is a mapping from variables names to values. Here is a translation of the module definition:

$$
\begin{array}{l}
[\![\mathsf{module}\ (x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n)]\!]\ \rho\ = \\
\qquad \lambda x.\ \textbf{if}\ x\ =\ x_1\ \textbf{then}\ [\![e_1]\!]\ \rho\ \textbf{else} \\
\qquad\quad \textbf{if}\ x\ =\ x_2\ \textbf{then}\ [\![e_2]\!]\ \rho\ \textbf{else} \\
\qquad\quad \ldots \\
\qquad\quad \textbf{if}\ x\ =\ x_n\ \textbf{then}\ [\![e_n]\!]\ \rho\ \textbf{else} \\
\qquad\quad \mathsf{lookup\text{-}error}
\end{array}
$$

The above is one possible translation. Note that ρ is passed as the environment to the translation of $e_1 \ldots e_n$. This has an important consequence: variables defined within the module are *not* visible within the initialization expression of other variables in the module. For instance in the above translation, we cannot refer to any of the x_i's within any of the e_i's. Nor does it seem possible to use the resulting environment within itself for the purpose of accessing the module variables since this leads to circularity problems.

However, we could translate module using techniques from the translation of letrec. The environment that results after the translation should be the same that is used within the module. This can be found by taking the fixed point of the function

$$
\begin{array}{l}
\lambda\rho'.\ \lambda x.\ \textbf{if}\ x\ =\ x_1\ \textbf{then}\ [\![e_1]\!]\ \rho'\ \textbf{else} \\
\qquad\quad \textbf{if}\ x\ =\ x_2\ \textbf{then}\ [\![e_2]\!]\ \rho'\ \textbf{else} \\
\qquad\quad \ldots \\
\qquad\quad \textbf{if}\ x\ =\ x_n\ \textbf{then}\ [\![e_n]\!]\ \rho'\ \textbf{else} \\
\qquad\quad \mathsf{lookup\text{-}error}
\end{array}
$$

This works fine if the e_i's are functions. However, it is not clear how to handle variables whose initialization expressions reference one another. For instance, consider

$$\mathsf{module}\ (x_1\ =\ x_2,\ x_2\ =\ x_1)$$

It is not quite clear what to do with such expressions. C++ avoids this problem by allowing functions within a class to call any other function within the class, but initialization expressions of variables are only allowed to refer to variables declared earlier in the class. For our purposes, we stay with the translation above.

The remainder of the translation is as follows:

$$
\begin{array}{rcl}
[\![e_m.x]\!]\rho & = & ([\![e_m]\!]\rho)\ \text{“}x\text{”} \\
[\![\mathsf{with}\ e_m\ e]\!]\rho & = & [\![e]\!](\mathsf{MERGE}\ \rho\ ([\![e_m]\!]\rho)) \\
\mathsf{MERGE}\ \rho\ \rho' & = & \lambda\ x.\mathsf{let}\ y\ =\ \rho'\ x\ \mathsf{in}\ \textbf{if}\ y\ =\ \mathsf{lookup\text{-}error}\ \textbf{then}\ \rho\ x\ \textbf{else}\ y
\end{array}
$$

[1] Actually our translation is to the target language $uML + S +$ lookup-error, a simple extension to uML that contains equality operators for strings and an extra token called lookup-error which is returned when a variable name is not found in a module.

Note that these translations are really functions of environments. That is, the translation above for $[\![e_m.x]\!]$ can be written:

$$[\![e_m.x]\!] \quad = \quad \lambda\ \rho.\ ([\![e_m]\!]\ \rho)\ \text{“}x\text{”}$$

4 State

Program state refers to the ability to change the values of program variables. The languages we have studied so far, such as λ-calculus and uML, have not had state, in the sense that once a value was bound to a variable, it was impossible to change that value. Although state is not a necessary feature of a programming language (e.g. λ-calculus is Turing complete but does not have a notion of state), it is a common feature of most languages. The reason why we still use the notion of the state in our languages is quite philosophical and has to do with our perception of the world. We tend to associate an identity with each object and differentiate between various objects much in the same way as we differentiate between different variables. However, it can be argued that even from the point of view of physics, identity is a myth: two electrons in the same state are indistinguishable in all respects and there is no way to assign an identity to them. Furthermore, we can argue that the concept of state in physics is an illusion and time is just another dimension of the universe. So it seems that our universe, just like λ-calculus, is purely functional!

4.1 Syntax

Despite this philosophy, programmers are accustomed to the notion of program state. So, we extend uML to include the ability to change the values of variables, and we call the new language uML!. We use a construct called a *reference cell* that allows its value to be changed over time. The syntax of uML! is as follows:

e	::=	...	
	|	ref e	*(create reference cell)*
	|	!e	*(evaluation or dereference)*
	|	e_1 := e_2	*(assignment or mutation)*
	|	unit	*(unit value)*

Informally, the semantics of these new expressions are as follows. ref e creates a reference cell having the initial value e. The !e expression evaluates to the current value of the reference cell that e evaluates to. $e_1 := e_2$ assigns the value that e_2 evaluates to to the reference cell that e_1 evaluates to, and in our semantics returns the value unit. (Note that other design choices could be made; for example, the assignment operator could return the value of e_2 so that expressions like $e_0 := e_1 := e_2$ could be written to assign the same value to multiple variables.)

Note that cell references are first class objects. This is more powerful than simple mutable variables as in C. As an example, consider the following expression:

```
let x = ref 1 in
  let y = x in
    let z = ( x := 2 ) in
      !y
```

Since y is an alias for x, dereferencing y gives us the value of x which in this case has changed from 1 to 2. The expression therefore evaluates to 2.

4.2 Operational semantics

We now write the operational semantics for the cell reference expressions in $uML!$. To do this, we define a configuration as a pair e, σ where e is an expression and σ is a function mapping locations (variable names) to values.

$$\frac{e \longrightarrow_{uML} e'}{e, \sigma \longrightarrow_{uML!} e', \sigma}$$

$$\frac{}{\mathsf{ref}\ v, \sigma \longrightarrow l, \sigma'} \qquad \text{where } \sigma' = \sigma[l \mapsto v] \wedge l \notin \text{domain}(\sigma)$$

$$\frac{}{!l, \sigma \longrightarrow \sigma(l), \sigma}$$

$$\frac{}{l := v, \sigma \longrightarrow \mathsf{unit}, \sigma[l \mapsto v]}$$

where $\sigma[l \mapsto v]$ is the same function as σ except that the value mapped to location l is changed to v.

Note that these are eager evaluation rules, evaluating values from left to right. So the evaluation contexts look like this:

$$E := \mathsf{ref}[\cdot] \mid ![\cdot] \mid [\cdot] := e \mid l := [\cdot]$$