

1 Introduction

In the last two lectures, we studied the static and dynamic approaches to variable scoping. These scoping disciplines are mechanisms for binding names to values so that the values can later be retrieved by their assigned name.

Both static and dynamic naming strategies 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 a more liberal naming discipline that is independent of any hierarchy induced by the syntactic structure or call structure.

Such a non-hierarchical scoping structure is provided by *modules*. A *module* is like a software black box with its own local namespace. It can export resources as a set of names without revealing its internal composition. Thus the names that can be used at a certain point in a program need not "come down from above" but can also be names exported by modules.

Good programming practices encourage *modularity*, especially in the construction of large systems. Programs should be composed of discrete components that communicate with one another along small, well-defined interfaces and are reusable. Modules are consistent with this idea. Each module can treat the others as black boxes; that is, they know nothing about what is inside the other module except as revealed by the interface.

Early programming languages had one global namespace in which names of all functions in source files and libraries were visible to all parts of the program. This was the approach for example of FORTRAN and C. There are certain problems with this:

- The various parts of the program can become tightly coupled. In other words, the global namespace does not enforce the modularity of the program. Replacing any particular part of the program with an enhanced equivalent can require a lot of effort.
- Undesired name collisions occur frequently, since names inevitably tend to coincide.
- In very large programs, it is difficult to come up with unique names, thus names tend to become non-mnemonic and hard to remember.

A solution to this problem is for the language to provide a *module mechanism* that allows related functions, values, and types to be grouped together into a common context. This allows programmers to create a local namespace, thus minimizing naming conflicts. Examples of modules are classes in Java and C++, packages in Java, or structures in ML. Given a module, we can access the variables in it by qualifying the variable names with the name of the module, or we can *import* the whole namespace of the module into our code, so we can use the module's names as if they had been declared locally.

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 refer to the entity with name **x** in the module **m**.

The second method of accessing the contents of a module 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 the 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 A simple module mechanism

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. There must be some values that we can use as names with an equality test. The syntax of the new language is:

$e ::=$...	
	module $(x_1 = e_1, \dots, x_n = e_n)$	(module definition)
	$e_m \cdot e$	(selector expression)
	with $e_m e$	(bringing into scope)
	lookup-error	(error)

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 names to values. Here is a translation of the module definition:

$$\llbracket \mathbf{module} (x_1 = e_1, x_2 = e_2, \dots, x_n = e_n) \rrbracket \rho \triangleq$$

$$\lambda x. \mathbf{if} \ x = x_1 \ \mathbf{then} \ \llbracket e_1 \rrbracket \rho \ \mathbf{else}$$

$$\quad \mathbf{if} \ x = x_2 \ \mathbf{then} \ \llbracket e_2 \rrbracket \rho \ \mathbf{else}$$

$$\quad \dots$$

$$\quad \mathbf{if} \ x = x_n \ \mathbf{then} \ \llbracket e_n \rrbracket \rho \ \mathbf{else}$$

$$\quad \mathbf{lookup-error}$$

The above is one possible translation. Note that ρ is passed as the environment to the translation of e_1, \dots, 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 this function:

$$\lambda \rho'. \lambda x. \mathbf{if} \ x = x_1 \ \mathbf{then} \ \llbracket e_1 \rrbracket \rho' \ \mathbf{else}$$

$$\quad \mathbf{if} \ x = x_2 \ \mathbf{then} \ \llbracket e_2 \rrbracket \rho' \ \mathbf{else}$$

$$\quad \dots$$

$$\quad \mathbf{if} \ x = x_n \ \mathbf{then} \ \llbracket e_n \rrbracket \rho' \ \mathbf{else}$$

$$\quad \mathbf{lookup-error}$$

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

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

¹Actually 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.

Java and C++ avoid 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. In terms of our translation, these languages start by inductively building up an environment for variable binding, then use the environment as the starting point for computing the fixed point.

The remainder of the translation is as follows:

$$\begin{aligned} \llbracket e_m.e \rrbracket \rho &\triangleq (\llbracket e_m \rrbracket \rho) (\llbracket e \rrbracket \rho) \\ \llbracket \mathbf{with} \ e_m \ e \rrbracket \rho &\triangleq \llbracket e \rrbracket (\mathbf{MERGE} \ \rho \ (\llbracket e_m \rrbracket \rho)) \\ \mathbf{MERGE} \ \rho \ \rho' &\triangleq \lambda x. \mathbf{let} \ y = \rho' \ x \ \mathbf{in} \ \mathbf{if} \ y = \mathbf{lookup-error} \ \mathbf{then} \ \rho \ x \ \mathbf{else} \ y \end{aligned}$$

In the translation of $\llbracket e_m.e \rrbracket$, presumably $\llbracket e \rrbracket \rho$ would evaluate to a name.

Note that these translations are really functions of environments. That is, the translation above for $\llbracket e_m.e \rrbracket$ can be written:

$$\llbracket e_m.e \rrbracket = \lambda \rho. (\llbracket e_m \rrbracket \rho) (\llbracket e \rrbracket \rho).$$