Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

1 Static vs. dynamic scoping

The *scope* of a variable is where that variable can be mentioned and used. Until now we could look at a program as written and immediately determine where any variable was bound. This was possible because the λ-calculus uses *static scoping* (also known as *lexical scoping*). The places where a variable can be used are determined by the lexical structure of the program. An alternative to static scoping is *dynamic scoping*, in which a variable is bound to the most recent (in time) value assigned to that variable.

The difference becomes apparent when a function is applied. In static scoping, any free variables in the function body are evaluated in the context of the defining occurrence of the function; whereas in dynamic scoping, any free variables in the function body are evaluated in the context of the function call. The difference is illustrated by the following program:

$$\begin{array}{l}\textbf{let } d = 2 \textbf{ in} \\ \quad \textbf{let } f = \lambda x.\, x + d \textbf{ in} \\ \quad\quad \textbf{let } d = 1 \textbf{ in} \\ \quad\quad\quad f\ 2\end{array}$$

In ML, which uses lexical scoping, the block above evaluates to 4:

1. The outer d is bound to 2.
2. f is bound to $\lambda x.\, x + d$. Since d is statically bound, this is will always be equivalent to $\lambda x.\, x + 2$ (the value of d cannot change, since there is no variable assignment in this language).
3. The inner d is bound to 1.
4. $f\ 2$ is evaluated using the environment in which f was defined; that is, f is evaluated with d bound to 2. We get $2 + 2 = 4$.

If the block is evaluated using dynamic scoping, it evaluates to 3:

1. The outer d is bound to 2.
2. f is bound to $\lambda x.\, x + d$. The occurrence of d in the body of f is not locked to the outer declaration of d.
3. The inner d is bound to 1.
4. $f\ 2$ is evaluated using the environment of the call, in which d is 1. We get $2 + 1 = 3$.

Dynamically scoped languages are quite common, and include many interpreted scripting languages. Examples of languages with dynamic scoping are (in roughly chronological order): early versions of LISP, APL, PostScript, TeX, and Perl. Early versions of Python also had dynamic scoping before they realized their error.

Dynamic scoping does have some advantages:

- Certain language features are easier to implement.
- It becomes possible to extend almost any piece of code by overriding the values of variables that are used internally by that piece.

These advantages, however, come with a price:

- Since it is impossible to determine statically what variables are accessible at a particular point in a program, the compiler cannot determine where to find the correct value of a variable, necessitating a more expensive variable lookup mechanism. With static scoping, variable accesses can be implemented more efficiently, as array accesses.
- Implicit extensibility makes it very difficult to keep code modular: the true interface of any block of code becomes the entire set of variables used by that block.

1.1 Scope and the interpretation of free variables

Scoping rules are all about how to evaluate free variables in a program fragment. With static scope, free variables of a function $\lambda x.\, e$ are interpreted according to the lexical (syntactic) context in which the term $\lambda x.\, e$ occurs. With dynamic scope, free variables of $\lambda x.\, e$ are interpreted according to the environment in effect when $\lambda x.\, e$ is applied. These are not the same in general.

We can demonstrate the difference by defining two translations $\mathcal{S}[\![\cdot]\!]$ and $\mathcal{D}[\![\cdot]\!]$ for the two scoping rules, static and dynamic. These translations will convert λ_{CBV} (with the corresponding scoping rule) into uML. (Because uML already has static scoping, the static scoping translation should have no effect on the meaning of the program.

For both translations, we use an *environment* to capture the interpretation of names. Here, an environment is simply a function from variables x to values.

$$\textbf{Environment} \quad \rho : \textbf{Var} \rightharpoonup \textbf{Value}$$

For example, the empty environment is: $\rho_0 = \lambda x.\, \textbf{error}$ because there are no variables bound in the empty environment. If we wanted an environment that bound only the variable y to 2, we could represent it as a uML term as follows:

$$\{\text{“}y\text{”} \mapsto 2\} = \lambda x.\, \textbf{if } x = \text{“}y\text{”} \textbf{ then } 2 \textbf{ else error}$$

The meaning of a language expression e is relative to the environment in which e occurs. Therefore, its meaning $[\![e]\!]$ is a function from an environment to the computation of a value.

$$[\![e]\!] : \textbf{Environment} \rightarrow \textbf{Expr}$$

We obtain a target language expression (in **Expr**) by applying $[\![e]\!]$ to some environment:

$$[\![e]\!]\rho : \textbf{Expr}$$

The translations take a term e and an environment ρ and produce a target-language expression involving values and environments that can be evaluated under the usual uML rules to produce a value.

The translation of the key expressions (the ones from lambda calculus) for static scoping follows. We use one new piece of syntactic sugar in the target language. We write “x” to mean a representation of the identifier x as an integer. Of course, there are many possible ways to encode an identifier as an integer, for example by thinking of the identifier as a stream of bits that are the binary representation of the integer.

$$\begin{aligned}
\mathcal{S}[\![x]\!]\ \rho &= \rho(\text{“}x\text{”}) \\
\mathcal{S}[\![e_1\ e_2]\!]\ \rho &= (\mathcal{S}[\![e_1]\!]\ \rho)\ (\mathcal{S}[\![e_2]\!]\ \rho) \\
\mathcal{S}[\![\lambda x.\, e]\!]\ \rho &= \lambda y.\, \mathcal{S}[\![e]\!]\ (\mathit{EXTEND}\ \rho\ x\ y),
\end{aligned}$$

where $(\mathit{EXTEND}\ \rho\ x\ v)$ adds a new binding to the environment ρ with the value of x replaced by v:

$$\mathit{EXTEND} \triangleq \lambda \rho x v.\, (\lambda y.\, \textbf{if } x = y \textbf{ then } v \textbf{ else } \rho\ y)$$

There are a couple of things to notice about the translation. It eliminates all of the variable names from the source program, and replaces them with new names that are bound immediately at the same level. All the lambda terms are closed, and there is no longer any role for the scoping mechanism of the target language to decide what to do with free variables.

1.2 Dynamic scoping

We can construct a translation that captures dynamic scoping through a few small changes:

$$\begin{array}{rcll} \mathcal{D}[\![x]\!]\ \rho & = & \rho\ (\text{“}x\text{”}) & \\ \mathcal{D}[\![\lambda x.\, e]\!]\ \rho & = & \lambda y \rho'.\, \mathcal{D}[\![e]\!]\ (\mathit{EXTEND}\ \rho'\ x\ y) & \text{(throw out lexical environment!)} \\ \mathcal{D}[\![e_1\ e_2]\!]\ \rho & = & (\mathcal{D}[\![e_1]\!]\ \rho)\ (\mathcal{D}[\![e_2]\!]\ \rho)\ \rho & \end{array}$$

The key is that the translation of a function is no longer just a function expecting the formal parameter; the function also expects to be provided with an environment ρ' describing the variable bindings at the call site. Unlike with static scoping, the translation of a λ abstraction, on the other hand, discards the lexical environment ρ existing at the point where the abstraction is evaluated. This makes it easier to represent functions, but creates the need to pass the dynamic environment explicitly to the function when it is called, as shown in the translation of application.

Because a function can be applied in different and unpredictable locations, it is difficult in general to come up with an efficient representation of the dynamic environment.

1.3 Correctness of the static scoping translation

That static scoping is the scoping discipline of λ_{CBV} is captured in the following theorem.

Theorem For any λ_{CBV} expression e and environment ρ, $\mathcal{S}[\![e]\!]\ \rho$ is (β, η)-equivalent to $e\{\rho(y)/y,\ y \in \mathbf{Var}\}$.

Proof. By structural induction on e. We write $\rho[v \mapsto x]$ for $(\mathit{EXTEND}\ \rho\ v\ x)$.

$$\begin{array}{rcl} \mathcal{S}[\![x]\!]\ \rho & = & \rho(x) \quad = \quad x\{\rho(y)/y,\ y \in \mathbf{Var}\}, \\ \mathcal{S}[\![e_1\ e_2]\!]\ \rho & = & (\mathcal{S}[\![e_1]\!]\ \rho)\ (\mathcal{S}[\![e_2]\!]\ \rho) \\ & = & (e_1\{\rho(y)/y,\ y \in \mathbf{Var}\})\ (e_2\{\rho(y)/y,\ y \in \mathbf{Var}\}) \\ & = & (e_1\ e_2)\{\rho(y)/y,\ y \in \mathbf{Var}\}, \\ \mathcal{S}[\![\lambda x.\, e]\!]\ \rho & = & \lambda v.\, \mathcal{S}[\![e]\!]\ \rho[x \mapsto v] \\ & = & \lambda v.\, e\{\rho[x \mapsto v](y)/y,\ y \in \mathbf{Var}\} \\ & = & \lambda v.\, e\{\rho[x \mapsto v](y)/y,\ y \in \mathbf{Var} - \{x\}\}\{\rho[x \mapsto v](x)/x\} \\ & = & \lambda v.\, e\{\rho(y)/y,\ y \in \mathbf{Var} - \{x\}\}\{v/x\} \\ & =_\beta & \lambda v.\, (\lambda x.\, e\{\rho(y)/y,\ y \in \mathbf{Var} - \{x\}\})\ v \\ & =_\eta & \lambda x.\, e\{\rho(y)/y,\ y \in \mathbf{Var} - \{x\}\} \\ & = & (\lambda x.\, e)\{\rho(y)/y,\ y \in \mathbf{Var}\}. \end{array}$$

□

The pairing of a function $\lambda x.\, e$ with an environment ρ is called a *closure*. The theorem above says that $\mathcal{S}[\![\cdot]\!]$ can be implemented by forming a closure consisting of the term e and an environment ρ that determines how to interpret the free variables of e. By contrast, in dynamic scoping, the translated function does not record the lexical environment, so closures are not needed.

Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

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:

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

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

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

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 this function:

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

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

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

[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.

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:

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

In the translation of $[\![e_m.e]\!]$, presumably $[\![e]\!]\ \rho$ would evaluate to a name.

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

$$
[\![e_m.e]\!] \quad = \quad \lambda\rho.\,([\![e_m]\!]\ \rho)\ ([\![e]\!]\ \rho).
$$