Scribe: Yookyung Jo, Lars Backstrom Lecturer: Andrew Myers

1 Overview

This lecture talks about more on naming.

- scoping semantics
- non-hierarchical scoping

2 Scoping semantics as translation

Last time, we looked at how names are interpreted. Naming is a key issue in programming. How to name affects how easy or secure it is to program. We feel that Turing machines or other programming languages that do not have names seem missing something. Somehow, programs with naming map closer to how we solve problems.

We build semantics for name scoping by translation. Let's define our target language as below.

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.\, e \mid e_0 e_1 \\ & \mid & \mathit{error} \mid s \mid n \\ & \mid & \text{let } x = e_1 \text{ in } e_2 \mid \ldots \end{array}$$

where *error* represents programs with variables that are not bound, s is a string, n is a number. The language is much like uML. The allowed expressions are not exhaustively enumerated. (When we say *if* $e_1 = e_2$ *then* e_3 *else* e_4, $e_1 = e_2$ means string comparison.)

What is the meaning of the translated expression, $[\![e]\!]$?
We need to recognize the meaning of an expression by considering the context (environment). We can think of our translation as conversion from an expression e into a function that expects a mapping from names to values. An environment is such a function that maps variable names to values.

$$\text{environment } \rho : \text{variable names} \mapsto \text{values}$$

Then we could think of $[\![e]\!]$ as a function that maps an environment to a value.

$$[\![e]\!] : \text{environment} \mapsto \text{value}$$

We could get the target language expression by applying $[\![e]\!]$ to some environment.

$$[\![e]\!]\rho : \text{target language expression}$$

Here are some examples of environments.
The empty environment is,

$$\rho_0 = \lambda x.\, \mathit{error}$$

because there are no variables bound in the empty environment.
Another example of an environment definition is,

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

Let's take a look at the actual translation of expressions.

$$\begin{aligned} [\![n]\!] &= \lambda\rho.\, n \\ [\![x]\!] &= \lambda\rho.\, \rho\text{“x”} \end{aligned}$$

Or it is more convenient to think of target language expression, which is the translation applied to an environment.

$$\begin{aligned} [\![n]\!]\rho &= n \\ [\![x]\!]\rho &= \rho\text{“x”} \\ [\![\text{let } x = e_1 \text{ in } e_2]\!]\rho &= \text{let } y = [\![e_1]\!]\rho \text{ in } [\![e_2]\!]\,(EXTEND\ \ \rho\text{“x”}y) \end{aligned}$$

where EXTEND is a function that generates a new environment by overriding some of the mappings in the original environment. It can be defined as,

$$EXTEND = \lambda\rho x v.\, \lambda x'.\, \text{if } x' = x \text{ then } v \text{ else } \rho\ \text{“}x'\text{”}$$

2.1 static scoping

So far, we dealt with the translation of expressions that does not depend on whether we use static scoping or dynamic scoping. However, the translation of application and lambda abstraction depends on scoping. In static scoping, an identifier is bound to a value when it first occurs in the program. Thus, the binding of a variable in function definition is determined by the scope present in the the function definition.

$$\begin{aligned} [\![e_1\ e_2]\!]\rho &= \text{let } f = [\![e_1]\!]\rho \text{ in let } v = [\![e_2]\!]\rho \text{ in } fv \\ [\![\lambda x.\, e]\!]\rho &= \lambda y.\, [\![e]\!]\,(EXTEND\ \ \rho\text{“x”}y) \qquad y \notin FV(e) \end{aligned}$$

2.2 Dynamic scoping

For dynamic scoping, we need to adjust our notion of variable binding in function application and abstraction. In dynamic scoping, an identifier is bound to a value when the function is called, not when the function expression is evaluated. Thus, the scope present in function definition is ignored. And when we do the actual function evaluation, we need to pass the dynamic environment to the function we call.

$$\begin{aligned} [\![e_1\ e_2]\!]\rho_{dyn} &= \text{let } f = [\![e_1]\!]\rho_{dyn} \text{ in let } v = [\![e_2]\!]\rho_{dyn} \text{ in } fv\rho_{dyn} \\ [\![\lambda x.\, e]\!]\rho_{lex} &= \lambda y \lambda \rho_{dyn}.\, [\![e]\!]\,(EXTEND\ \ \rho_{dyn}\text{“x”}y)\,. \qquad \text{(throw out lexical environment)} \end{aligned}$$

2.3 Static scope vs. dynamic scope

Dynamic scoping translates function code into just another code. While in static scope translation, both function code and the naming environment $([\![e]\!], \rho_{lex})$ is included.
Dynamic scoping has its advantages. Translation of certain features in the program is easier with dynamic scope. For example, we do not need to look up the lexical environment for variable binding for function code, while in static scope, we have to carry the function closure which is the function code and its lexical environment.
Also, sometimes, it becomes handy to be able to customize a function by providing new values for its free variables.
However, its advantages also imply its disadvantages. In dynamic scoping, program optimization is hard, because the compiler does not know statically how to determine the value of a variable. Simple optimization techniques with string matches used in static scoping cannot be used in dynamic scoping.
Dynamic scoping hurts program modularity because of the free variables in function that can change their values according the execution environment. As more and more of programming activities are to glue existing big modules into something, rather than writing a new piece of code, modularity is a serious concern. This is why people gradually moved to designing programming languages with lexical(static) scoping.

3 Modules

We'd like to be able to share resources easily across large blocks of code. There are a few different ways to do this. In some languages (like C), this is done with a global namespace. Every part of the program has access to the variables in this global namespace. Unfortunately, this can cause problems, as different blocks of code cannot use the same name from the global namespace to refer to two different variables. Additionally, it causes a large amount of clutter in the global namespace for large amounts of code.

An alternative to this is to use modules. These are similar to classes in Java and C++. The modules contain a number of different variable names, and given a particular module, we can access the variables in it. Now, we would like to extend our definition of uML to include the concept of modules as follows:

$$\begin{aligned} e ::= \quad & \mathsf{module}(x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n) \\ & \mid \; e_m.x \;\mid\; \mathsf{with}\ e_m\ e \;\mid\; \ldots \end{aligned}$$

Simply put, we can now have modules in uML that bindings for a number of names to variables. Then, given a module, we can extract a single variable from it using $e_m.x$ or we can evaluate an expression e with all of the variables in the module e_m by doing $\mathsf{with}\ e_m\ e$, which brings all variables in e_m into scope and then evaluates e.

Naturally, we would now like to extend our translation from uML to lambda calculus to include our new modules. The basic idea is that a module is very much like our naming environment, ρ. With this intuition, we define the following translations:

$$\begin{aligned} [\![module(x_1 = e_1, x_2 = e_2, ..., x_n = e_n)]\!]\rho &= \lambda x.\, if \quad x{=}x_1 \text{ then } [\![e_1]\!]\rho \text{ else if} \\ &\qquad\quad x{=}x_2 \text{ then } [\![e_2]\!]\rho \text{ else if} \\ &\qquad\quad \ldots \\ &\qquad\quad x{=}x_n \text{ then } [\![e_n]\!]\rho \text{ else error} \\ [\![e_m.x]\!]\rho &= [\![e_m]\!]\rho \text{ ``}x'' \\ [\![\text{with } e_m\ e]\!]\rho &= [\![e]\!] \text{ (MERGE } \rho \text{ } ([\![e_m]\!] \rho)) \end{aligned}$$

Notice that we've made a number of important decisions in our translation. In the module definition, we evaluate all of the expressions in the same naming environment. Thus, the functions and variables within the environment cannot refer to each other. In some senses this makes sense. We probably don't want to be able to do:

x1 = x2
x2 = x1

However, we might want to have functions in the module that refer to values in the same module. It is possible to modify the definition to account for this.