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.