In this lecture we introduce the topic of *scope* in the context of the λ-calculus and define translations from λ-CBV to FL for the two most common scoping rules, *static* and *dynamic* scoping.

1 Overview

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 *scope* of a variable is where that variable can be mentioned and used. In static 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:

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

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

1. The outer d is bound to 2.
2. The 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. When evaluating the expression $f\,2$, free variables in the body of f are evaluated using the environment in which f was defined. In that environment, d was 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. The 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. When evaluating the expression $f\,2$, free variables in the body of f are 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, but it was later changed to static scoping.

Dynamic scoping does have a few 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 will be 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.

2 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 term $\lambda x.\, e$ are interpreted according to the 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 static and dynamic scoping, respectively. These translations will convert λ-CBV with the corresponding scoping rule into FL.

For both translations, we use environments to capture the interpretation of names. An *environment* is simply a partial function with finite domain from variables x to values.

$$\rho : \mathit{Var} \rightharpoonup \mathit{Val}$$

Let $\rho[v/x]$ denote the environment that is identical to ρ except that its value at x is v:

$$\rho[v/x](y) \quad \triangleq \quad \begin{cases} \rho(y), & \text{if } y \neq x, \\ v, & \text{if } y = x. \end{cases}$$

The set of all environments is denoted Env. The meta-operator $[v/x]$ is called a *rebinding operator*.

We will need a mechanism to represent environments in the target language. For this purpose we need to code variables and environments as values of the target language. We write $\ulcorner x \urcorner$ to represent the code of the variable x. The exact nature of the coding is unimportant, as long as it is possible to look up the value of a variable given its code and update an environment with a new binding. Thus the only requirement is that there be definable methods lookup and update in the target language such that for any variable x and value v, if R is a representation of the environment ρ, then

$$\mathsf{lookup}\ R\ \ulcorner x \urcorner \quad = \quad \begin{cases} \rho(x), & \text{if } x \in \mathsf{dom}\,\rho, \\ \mathsf{error}, & \text{if } x \notin \mathsf{dom}\,\rho \end{cases}$$

and $\mathsf{update}\ R\ v\ \ulcorner x \urcorner$ is a representation of $\rho[v/x]$. As is customary, we will use ρ to represent both the environment and its representation in the target language, writing $\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner$ and $\mathsf{lookup}\ \rho\ \ulcorner x \urcorner$.

For example, we might represent variables as integers and environments as a functions that take an integer input n and return the value associated with the variable whose code is n, or error if there is no such binding. With this encoding, the empty environment would be $\lambda n.\,\mathsf{error}$, and the environment that binds only the variable y to 2 could be represented as

$$\lambda n.\,\mathsf{if}\ n = \ulcorner y \urcorner\ \mathsf{then}\ 2\ \mathsf{else}\ \mathsf{error}.$$

With this encoding we could define

$$\mathsf{lookup} \triangleq \lambda \rho n.\,\rho(n) \qquad\qquad \mathsf{update} \triangleq \lambda \rho v n.\,\lambda m.\,\mathsf{if}\ m = n\ \mathsf{then}\ v\ \mathsf{else}\ \rho(m).$$

Given such an encoding, let Env' denote the set of all representations of environments in the target language. The meaning of a language expression e is relative to the environment in which e occurs. Therefore, the meaning $[\![e]\!]$ is a function from environments to expressions in the target language,

$$[\![e]\!] : Env' \to \mathsf{FL},$$

where FL represents the set of all target language expressions. Thus $[\![e]\!]\rho$ is an expression in the target language FL involving values and environments that can be evaluated under the usual FL rules to produce a value.

3 Static Scoping

The translation for static scoping is:

$$\begin{aligned}
\mathcal{S}[\![x]\!]\,\rho &\triangleq \mathsf{lookup}\ \rho\ \ulcorner x \urcorner \\
\mathcal{S}[\![e_1 e_2]\!]\,\rho &\triangleq (\mathcal{S}[\![e_1]\!]\,\rho)\,(\mathcal{S}[\![e_2]\!]\,\rho) \\
\mathcal{S}[\![\lambda x.\,e]\!]\,\rho &\triangleq \lambda v.\,\mathcal{S}[\![e]\!]\,(\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner), \quad v \text{ fresh.}
\end{aligned}$$

There are a couple of things to notice about this 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 λ-terms are closed, so there is no longer any role for the scoping mechanism of the target language to decide what to do with free variables.

4 Dynamic Scoping

The translation for dynamic scoping is:

$$\begin{aligned}
\mathcal{D}[\![x]\!]\,\rho &\triangleq \mathsf{lookup}\ \rho\ \ulcorner x \urcorner \\
\mathcal{D}[\![e_1 e_2]\!]\,\rho &\triangleq (\mathcal{D}[\![e_1]\!]\,\rho)\,(\mathcal{D}[\![e_2]\!]\,\rho)\,\rho && (1) \\
\mathcal{D}[\![\lambda x.\,e]\!]\,\rho &\triangleq \lambda v\tau.\,\mathcal{D}[\![e]\!]\,(\mathsf{update}\ \tau\ v\ \ulcorner x \urcorner), \quad v, \tau \text{ fresh.} && (2)
\end{aligned}$$

In (2), we have thrown out the lexical environment ρ and replaced it with a parameter τ. Thus the translation of a function no longer expects just a single argument v; it also expects to be provided with an environment τ describing the variable bindings at the call site. This environment is passed in to the function when it is called, as shown in (1).

Because a function can be applied in different and unpredictable locations that can vary at runtime, it is difficult in general to come up with an efficient representation of dynamic environments.

5 Correctness of the Static Scoping Translation

That static scoping is the scoping discipline of λ-CBV is captured in the following theorem.

Theorem 12.1. *For any λ-CBV expression e and $\rho \in Env$ such that $FV(e) \subseteq \mathsf{dom}\,\rho$, let $\rho' \in Env'$ be a representation of ρ. Then $\mathcal{S}[\![e]\!]\,\rho'$ is $\beta\eta$-equivalent to $e\{\rho(y)/y \mid y \in Var\}$.*

Proof. By structural induction on e. For variables and applications,

$$\begin{array}{rcl} \mathcal{S}[\![x]\!]\,\rho' & = & \mathsf{lookup}\ \rho'\ \ulcorner x \urcorner \ =\ \rho(x)\ =\ x\{\rho(y)/y \mid y \in Var\} \\ \mathcal{S}[\![e_1 e_2]\!]\,\rho' & = & (\mathcal{S}[\![e_1]\!]\,\rho')\,(\mathcal{S}[\![e_2]\!]\,\rho') \\ & = & (e_1\{\rho(y)/y \mid y \in Var\})\,(e_2\{\rho(y)/y \mid y \in Var\}) \\ & = & (e_1 e_2)\{\rho(y)/y \mid y \in Var\}. \end{array}$$

For a λ-abstraction $\lambda x.\, e$, for any value v, since $\mathsf{update}\ \rho'\ v\ \ulcorner x \urcorner$ is a representation for $\rho[v/x]$, by the induction hypothesis

$$\begin{array}{rcl} \mathcal{S}[\![e]\!]\,(\mathsf{update}\ \rho'\ v\ \ulcorner x \urcorner) & =_{\beta\eta} & e\{\rho[v/x](y)/y \mid y \in Var\} \\ & = & e\{\rho(y)/y \mid y \in Var - \{x\}\}\{v/x\} \\ & =_{\beta} & (\lambda x.\, e\{\rho(y)/y \mid y \in Var - \{x\}\})\,v \\ & = & ((\lambda x.\, e)\{\rho(y)/y \mid y \in Var\})\,v. \end{array} \tag{3}$$

Then

$$\begin{array}{rcll} \mathcal{S}[\![\lambda x.\, e]\!]\,\rho' & = & \lambda v.\, \mathcal{S}[\![e]\!]\,(\mathsf{update}\ \rho'\ v\ \ulcorner x \urcorner) & \\ & = & \lambda v.\, ((\lambda x.\, e)\{\rho(y)/y \mid y \in Var\})\,v & \text{by (3)} \\ & =_{\eta} & (\lambda x.\, e)\{\rho(y)/y \mid y \in Var\}. & \end{array}$$

□

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.