CS611 Lecture 10 — Semantics and scope rules — 20 September 2004

Scribe: Bryan Silverthorn, Yogeshwer Sharma — Lecturer: Andrew Myers

1 Finishing the translation of uML

In the previous lecture we saw a new language, an extension of the λ-calculus called uML. We also saw its reduction rules and evaluation contexts, and began to define a translation from uML to the λ-calculus with call by value semantics (λ-CBV). Today, we will consider the translation of the tuple, let and letrec constructs of uML, relating them to the broader topics of recursion, error checking, and strong typing. Finally, we will introduce the topic of variable scope in the context of the lambda calculus, and define translations to λ-CBV for the two most common scope rules.

2 Translation from uML to λ-CBV

Let us consider the translation of tuples first. We represent tuples as lists which use the *CONS*, *LEFT*, and *RIGHT* functions as their "interface". Define

$$\begin{aligned} \mathit{CONS} &\triangleq \lambda x.\lambda y.\lambda f.(f\ x\ y) \\ \mathit{LEFT} &\triangleq \lambda p.(p\ \lambda x.\lambda y.x) \\ \mathit{RIGHT} &\triangleq \lambda p.(p\ \lambda x.\lambda y.y) \end{aligned}$$

CONS is a function for constructing lists by adding one element at the head of the list. The *LEFT* construct extracts the head of the list, while *RIGHT* chops off the head. Now, we can define the translation from tuples to λ-CBV as follows:

$$\begin{aligned} [\![()]\!] &\triangleq \mathit{NULL} \\ [\![(e_1, e_2, \ldots, e_n)]\!] &\triangleq \mathit{CONS}\ [\![e_1]\!]\ [\![(e_2, \ldots, e_n)]\!] \\ [\![\#1\ e]\!] &\triangleq \mathit{LEFT}\ [\![e]\!] \\ [\![\#n\ e]\!] &\triangleq [\![\#n-1\ (\mathit{RIGHT}\ [\![e]\!])]\!] \end{aligned}$$

For the case of the let expression:

$$[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] = (\lambda x.[\![e_2]\!])[\![e_1]\!]$$

Now comes the last of our uML constructs, letrec. This construct allows us to define multiple *mutually recursive* functions, each of which is able to make calls to itself and to the other functions defined in the same letrec block. We will consider the case when only one function is defined in the letrec. Let us define another construct for the naming of recursive functionss: (rec $f.e$) defines a function named f whose body e can refer to itself. If we take the rec construct for granted, we have the following translation.

$$[\![\mathsf{letrec}\ f_1 = \lambda x_1.e_1\ \mathsf{in}\ e_2]\!] = (\lambda f_1.[\![e_2]\!])(\mathsf{rec}\ f_1.\lambda x_1.[\![e_1]\!])$$

And the rec construct can be converted to λ-CBV using the Y-combinator, which we have seen previously. We make the term $\lambda f.e$ from the values in (rec $f.e$) and get $Y(\lambda f.e)$ as the definition of rec. So the full version of the translation above becomes:

$$[\![\mathsf{letrec}\ f_1 = \lambda x_1.e_1\ \mathsf{in}\ e_2]\!] = (\lambda f_1.[\![e_2]\!])(Y(\lambda f_1.\lambda x_1.e_1))$$

3 Soundness of the translation

We have defined the translation from uML to λ-CBV. Now a new question arises: is the translation *sound* and *complete*? The question of "soundness" asks "does any evaluation/reduction in the target language correspond to an equivalent evaluation/reduction in the source language?" What is the answer to this question with our translation?

Consider the expression $\textsf{if}\ 0\ \textsf{then}\ 1\ \textsf{else}\ 2$. This will not evaluate to anything in uML (because 0 is not a boolean value), but what happens to it in λ-CBV? 0 has the same representation as FALSE in the target language, so it will evaluate to 2 without any problems. This example suggests that there exists some expression e such that some evaluation of $[\![e]\!]$ in the target language does not correspond to an evaluation of e in the source language. Clearly the answer to our original question is "no": our translation is not sound.

How to approach this problem? There are several approaches. We can leave the load of writing the correct programs on the programmer and hence ignore the problem. But this does not seem to be a good idea: consider a program which does not make sense in the programming language, and which therefore does something unexpected after translation to the machine-language—where it did "make sense" (at the machine level). Prominent examples of languages which take this approach are C and C++.

Another approach is to augment our translation so that an "incorrect" expression cannot be translated to a "correct" expression (ie, one that evalutes successfully at runtime). In the augmented translation, we introduce another construct called error which signifies that there is no further evaluation possible.

Now we introduce the idea of types (although very primitive) which take care of some "incorrect" translations like above. We keep a tag with each type of value, specify 0 for booleans, 1 for integers, 2 for tuples and 3 for functions, and we check that we are getting the right kind of values where they are needed (for example, we check that we have a boolean conditional in an if-then-else construct).

Call the new translation $\mathcal{E}[\![e]\!]$. The translation becomes,

$$\begin{aligned}
\mathcal{E}[\![\textsf{true}]\!] &= [\![(0, \textsf{true})]\!] \\
\mathcal{E}[\![\textsf{false}]\!] &= [\![(0, \textsf{false})]\!] \\
\mathcal{E}[\![n]\!] &= [\![(1, n)]\!] \\
\mathcal{E}[\![(e_1, e_2, \ldots, e_n)]\!] &= [\![(2, (e_1, e_2, \ldots, e_n))]\!] \\
\mathcal{E}[\![\lambda x.e]\!] &= [\![(3, \lambda x.e)]\!]
\end{aligned}$$

In such case, the translation of general terms will be something like this.

$$\begin{aligned}
&\mathcal{E}[\![\textsf{if}\ e_0\ \textsf{then}\ e_1\ \textsf{else}\ e_2]\!] \\
&= [\![\textsf{if}\ \mathit{FIRST}(e_0) \neq 0\ \textsf{then error else}\ (\mathit{SECOND}(e_0)\ \lambda z.e_1\ \lambda z.e_2)]\!]
\end{aligned}$$

The motivation behind introducing the new construct error is that if the source program should become stuck somewhere during its execution, then the translation of that program will evaluate to error. More concretely,

$$e \rightarrow^* e' \ \wedge \ \nexists\, e''.e' \rightarrow e'' \quad \Rightarrow \quad [\![e]\!] \rightarrow^* \textsf{error}$$

Total adherence to source language semantics in the program translation, verified at compile time, runtime, or both, is *strong typing*. Both strongly typed and weakly typed languages are used modern programming. For example, ML and Scheme are strongly typed languages while C, C++ and Pascal are not.

The translation from e to $[\![e]\!]$ is *compilation* and the process of reducing $[\![e]\!]$ and obtaining a value is *execution*. Compilation, which happens once, is considered a *static* process, whereas execution is *dynamic*.

4 Variable scoping rules

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 *block-structured static scope* (also called *lexical scope*).

Recall that the scope of a variable is where that variable can be mentioned and used. In block-structured static scope, the places where a variable can be used are determined by the lexical structure of the program, and they are determined by the nesting (block) structure of the program itself. However, in many languages, such as ML, *holes in scope* occur when a variable is defined whose name is already in scope, effectively shadowing that variable inside the "hole" created by the scope of the new variable. Because holes in scope can lead to programmer errors, some languages, such as Java and CLU, prevent some of them.

The usual alternative to static (lexical) scope is *dynamic scope*, in which a variable is bound to the most recent dynamic binding of that variable. Scoping rules can change the meaning of a program, such as the following:

```
let delta = 2 in
  let bump = λx.x+delta in
     let delta = 1 in
        bump(2)
```

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

$\quad$ `delta` is bound to 2.
$\rightarrow$ `bump` is set to $\lambda x.x$+`delta`.
Since `delta` is statically bound, this is immediately equivalent to $\lambda x.x + 2$ (it is a *closure*—it is paired with the values of its variables).
$\rightarrow$ `delta` is bound to 1. Nested referrals to `delta` will use the new value.
$\rightarrow$ `bump(2)` is evaluated using the environment in which `bump` was defined.
That is, `bump` is evaluated with `delta` equal to 2. We get $2 + 2 = 4$.

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

$\quad$ `delta` is bound to 2.
$\rightarrow$ `bump` is set to $\lambda x.x$+`delta`.
The value of `delta` in `bump` may change—it is not a closure.
$\rightarrow$ `delta` is bound to 1.
$\rightarrow$ `bump(2)` is evaluated using the current environment, where `delta` is 1.
That is, `bump` is evaluated using the global environment. 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, Perl, and Python. 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, each 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.

Why might dynamic scoping be easier to implement? We can demonstrate the difference by defining translations for each of the two scoping rules. Our translation will convert an expression e into a function that expects an environment ρ, a mapping from identifiers (x) to values (ρ("x")). Note that we bypass, for the evaluation of the expression e, the lexically-scoped variable lookup of the λ-calculus. The translation for static scoping is as follows:

$$\begin{aligned}
[\![x]\!] &= \lambda\rho.\rho(\text{``}x\text{''}) \\
[\![n]\!] &= \lambda\rho.n \\
[\![e_1\ e_2]\!]\rho &= ([\![e_1]\!]\rho)([\![e_2]\!]\rho) \\
[\![\lambda x.e]\!]\rho &= \lambda y.[\![e]\!](\mathit{EXTEND}\ \rho\ \text{``}x\text{''}\ y)
\end{aligned}$$

Where $\mathit{EXTEND} \triangleq \lambda p.\lambda x.\lambda v.\lambda x'.\textbf{if } x' = x \textbf{ then } v \textbf{ else } \rho(x')$. And the translation for dynamic scoping is:

$$\begin{array}{rcl} [\![\lambda x.e]\!]\rho_{\text{lex}} & = & \lambda y.\lambda\rho_{\text{dyn}}.[\![e]\!](\mathit{EXTEND}\ \rho_{\text{dyn}}\ \text{“}x\text{”}\ y) \quad \text{(throw out lexical environment)} \\ [\![e_1\ e_2]\!]\rho & = & ([\![e_1]\!]\rho)([\![e_2]\!]\rho)\rho \end{array}$$

Using the same definition of *EXTEND*, above. Notice that the translation of a function for static scoping contains two components: the body of the function $[\![e]\!]$ and the lexical environment ρ. This pair of components together form a *closure*. By contrast, in dynamic scoping the translated function does not record the lexical environment: closures are not needed.