1 Overview

Last time 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 scoping rules, *static* and *dynamic* scoping.

2 Translation from uML to λ-CBV (continued)

Let us consider the translation of tuples. We have already seen how to represent lists in the λ-calculus using the functions *LIST*, *HEAD*, *TAIL*, *NULL*, and *EMPTY* with the following properties:

$$\begin{aligned} \mathit{HEAD}\ (\mathit{LIST}\ e_1\ e_2) &= e_1 \\ \mathit{TAIL}\ (\mathit{LIST}\ e_1\ e_2) &= e_2 \\ \mathit{EMPTY}\ (\mathit{LIST}\ e_1\ e_2) &= \mathit{FALSE} \\ \mathit{EMPTY}\ \mathit{NULL} &= \mathit{TRUE}. \end{aligned}$$

Using these constructs, we can define the translation from tuples to λ-CBV as follows:

$$\begin{aligned} [\![()]\!] &\triangleq \mathit{NULL} \\ [\![(e_1, e_2, \ldots, e_n)]\!] &\triangleq \mathit{LIST}\ [\![e_1]\!]\ [\![(e_2, \ldots, e_n)]\!] \\ [\![\#n\ e]\!] &\triangleq \mathit{HEAD}\ (\mathit{TAIL}^n\ [\![e]\!]). \end{aligned}$$

As in the last lecture, this translation is not sound, because there are stuck uML expressions whose translations are not stuck; for example, #1 ( ).

For **let** expressions, we define

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

Now comes the last of our uML constructs, **letrec**:

$$\mathbf{letrec}\ f_1 = \lambda x_1.\, e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ f_n = \lambda x_n.\, e_n\ \mathbf{in}\ e.$$

This construct allows us to define mutually recursive functions, each of which is able to call itself and other functions defined in the same **letrec** block. We will consider only the case $n = 1$. Recall that, using the Y-combinator, we can produce a fixpoint $Y(\lambda f.\, \lambda x.\, e)$ of $\lambda f.\, \lambda x.\, e$. We can think of $Y(\lambda f.\, \lambda x.\, e)$ as a recursively-defined function f such that $f = \lambda x.\, e$, where the body e can refer to f. Then we define

$$[\![\mathbf{letrec}\ f = \lambda x.\, e_1\ \mathbf{in}\ e_2]\!] \triangleq (\lambda f.\, [\![e_2]\!])\ (Y(\lambda f.\, [\![\lambda x.\, e_1]\!])).$$

3 Soundness of the Translation

We have defined the translation from uML to λ-CBV. As observed, the translation is not sound, because there are some nonsensical expressions that get stuck in the uML semantics, yet their translations converge to a λ-CBV value.

There are several ways of dealing with this issue. One approach is to ignore it entirely. This places the full responsibility for ensuring correct programs on the back of the programmer. Experience has shown that

this is not necessarily a good idea! Prominent examples of languages that take this approach are C and C++.

Another approach is to augment our translation so that an erroneous expression cannot be translated to a correct expression—that is, one that evaluates successfully at runtime. In the augmented translation, we introduce another construct called *ERROR*, which signifies that there is no further valid evaluation possible.

3.1 Runtime Types

Now we introduce the idea of *runtime types* which take care of some incorrect translations. We keep a tag with each type of value, say 0 for Booleans, 1 for integers, 2 for tuples, and 3 for functions. We check that we are getting the right kind of values where they are expected. For example, we would check that we have a Boolean value for the test in a conditional if-then-else construct.

Call the new translation $\mathcal{E}[\![e]\!]$. Define

$$\begin{aligned}
\mathcal{E}[\![\mathbf{true}]\!] &\triangleq PAIR\ \overline{0}\ TRUE \\
\mathcal{E}[\![\mathbf{false}]\!] &\triangleq PAIR\ \overline{0}\ FALSE \\
\mathcal{E}[\![n]\!] &\triangleq PAIR\ \overline{1}\ \overline{n} \\
\mathcal{E}[\![(e_1, e_2, \ldots, e_n)]\!] &\triangleq PAIR\ \overline{2}\ [\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!]] \\
\mathcal{E}[\![\lambda x.\, e]\!] &\triangleq PAIR\ \overline{3}\ \lambda x.\, \mathcal{E}[\![e]\!] \\
ERROR &\triangleq PAIR\ \overline{4}\ \overline{0}.
\end{aligned}$$

Thus each value is paired with its runtime type. The translation of general terms will be something like this.

$$\begin{aligned}
&\mathcal{E}[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!] \\
&\quad\triangleq\ (\lambda x.\, IF\ (EQ\ (FIRST\ x)\ \overline{0})\ (IF\ (SECOND\ x)\ (\lambda z.\, \mathcal{E}[\![e_1]\!])\ (\lambda z.\, \mathcal{E}[\![e_2]\!])\ ID)\ ERROR)\ \mathcal{E}[\![e_0]\!]
\end{aligned}$$

We might have defined it more simply as

$$IF\ (EQ\ (FIRST\ \mathcal{E}[\![e_0]\!])\ \overline{0})\ (IF\ (SECOND\ \mathcal{E}[\![e_0]\!])\ (\lambda z.\, \mathcal{E}[\![e_1]\!])\ (\lambda z.\, \mathcal{E}[\![e_2]\!])\ ID)\ ERROR$$

which is functionally equivalent; but by doing it the way we did, $\mathcal{E}[\![e_0]\!]$ is only evaluated once.

We have to modify our definition of values in λ-CBV to accommodate the tags. Under the new definition, *ERROR* will not be a value, but a non-value stuck expression.

The motivation behind 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*, so that it also will be stuck. Thus neither program will generate a value.

3.2 Strong Typing

Total adherence to source language semantics in the program translation, verified at compile time, runtime, or both, is called *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 Static vs. Dynamic Scope

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 scope* (also known as *lexical scope*).

In practice, 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:

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

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, 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, 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.

4.1 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 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 $\mathcal{SS}[\![\cdot]\!]$ and $\mathcal{DS}[\![\cdot]\!]$ for the two scoping rules. Recall that an *environment* is a function from variables x to values. The translations will take a λ-term e and an environment ρ and produce a meta-expression involving values and environments that can be evaluated under the usual rules of the λ-calculus to produce a value.

The translation for static scoping is as follows:

$$\begin{array}{rcl} \mathcal{SS}[\![x]\!]\ \rho & \triangleq & \rho(x) \\ \mathcal{SS}[\![e_1\ e_2]\!]\ \rho & \triangleq & (\mathcal{SS}[\![e_1]\!]\ \rho)\ (\mathcal{SS}[\![e_2]\!]\ \rho) \\ \mathcal{SS}[\![\lambda x.\, e]\!]\ \rho & \triangleq & \lambda v.\, \mathcal{SS}[\![e]\!]\ \rho[v/x], \end{array}$$

where $\rho[v/x]$ refers to the environment ρ with the value of x replaced by v:

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

The translation for dynamic scoping is:

$$\begin{array}{rcl} \mathcal{DS}[\![x]\!]\ \rho & \triangleq & \rho(x) \\ \mathcal{DS}[\![\lambda x.\, e]\!]\ \rho & \triangleq & \lambda v.\, \lambda \tau.\, \mathcal{DS}[\![e]\!]\ \tau[v/x] \quad \text{(throw out lexical environment)} \\ \mathcal{DS}[\![e_1\ e_2]\!]\ \rho & \triangleq & (\mathcal{DS}[\![e_1]\!]\ \rho)\ (\mathcal{DS}[\![e_2]\!]\ \rho)\ \rho. \end{array}$$

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, $\mathcal{SS}[\![e]\!]\,\rho$ is (β,η)-equivalent to $e\{\rho(y)/y,\ y \in \mathbf{Var}\}$.

Proof. By structural induction on e.

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

□

The pairing of a function $\lambda x.\, e$ with an environment ρ is called a *closure*. The theorem above says that $\mathcal{SS}[\![\cdot]\!]$ 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.