CSCI.4430/6969 Programming Languages
Lecture Notes

September 8, 2005

2.3.2 Sequencing Combinator

The *normal order sequencing* combinator is

$$Seq = \lambda x.\lambda y.(\lambda z.y \quad x)$$

where z is chosen so that it does not appear free in y.

This combinator guarantees that x is evaluated before y, which is important in programs with side-effects. Assuming we had a "display" function sending output to the console, an example is

$$((Seq \quad (\text{display "hello"})) \quad (\text{display "world"}))$$

The combinator would not work in applicative order (call by value) evaluation because evaluating the `display` functions before getting them passed to the `Seq` function would defeat the purpose of the combinator: to sequence execution. In particular, if the arguments are evaluated *right to left*, execution would not be as expected.

The *applicative-order sequencing* combinator can be written as follows:

$$ASeq = \lambda x.\lambda y.(y \quad x)$$

where y is a lambda abstraction that *wraps* the original last expression to evaluate.

The same example above would be written as follows:

$$((ASeq \quad (\text{display "hello"})) \quad \lambda x.(\text{display "world"}))$$

with x fresh, that is, not appearing free in the second expression.

This strategy of *wrapping* a lambda-calculus expression to make it a value and delay its evaluation is very useful. It enables to simulate *call by name* parameter passing in languages using *call by value*. The process of wrapping is also called *freezing* an expression, and the resulting frozen expression is called a *thunk*. Evaluating a thunk to get back the original expression is also called *thawing*.

2.4 η-Conversion

Consider the expression

$$(\lambda x.(\lambda x.x^2 \quad x) \quad y).$$

Using β-reduction, we can take $E = (\lambda x.x^2 \quad x)$ and $M = y$. In the reduction we only replace the one x that is free in E to get

$$\xrightarrow{\beta} (\lambda x.x^2 \quad y).$$

We use the symbol $\xrightarrow{\beta}$ to show that we are performing β-reduction on the expression (As another example we may write $\lambda x.x^2 \xrightarrow{\alpha} \lambda y.y^2$ since α-renaming is taking place).

Another type of operation possible on lambda calculus expressions is called η-conversion ("eta"-reduction when applied from left to right). We perform η-reduction using the rule

$$\lambda x.(E \quad x) \xrightarrow{\eta} E.$$

η-reduction can only be applied if E is a lambda expression taking a single argument, and x does not appear free in E.

Starting with the expression, $\lambda x.(\lambda x.x^2 \quad x)$, we can perform η-reduction to obtain

$$\lambda x.(\lambda x.x^2 \quad x) \xrightarrow{\eta} \lambda x.x^2.$$

Starting with the same expression as before, $(\lambda x.(\lambda x.x^2 \quad x) \quad y)$, we can perform η-reduction to obtain

$$(\lambda x.(\lambda x.x^2 \quad x) \quad y) \xrightarrow{\eta} (\lambda x.x^2 \quad y),$$

which gives the same result as β-reduction.

Another example of η-reduction follows:

$$\lambda x.(y \quad x) \xrightarrow{\eta} y.$$

η-reduction can be considered a program optimization. For example, consider the following Oz definitions:

```
declare Increment = fun {$ X} X+1 end

declare Inc = fun {$ X} {Increment X} end
```

Using η-reduction, we could reduce `{Inc 6}` to `{Increment 6}` avoiding one extra-function call. This compiler optimization is also called *inlining*.

η-conversion can also affect termination of expressions in applicative order expression evaluation. For example, the Y reduction combinator has a terminating applicative order form that can be derived from the normal order combinator form by using η-conversion (see Section 2.5.)

2.5 Recursion Combinator

The *recursion* combinator allows recursion in lambda expressions. For example, suppose we want to implement a recursive version of the factorial operation,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}.$$

We could start by attempting to write a the recursive function f in the lambda calculus (assuming it has been extended with conditionals, and numbers):

$$\begin{array}{lll} f : \lambda g \lambda n(\text{if} & (= & n \ 0) \\ & 1 & \\ & (* & n \ (g \ (- \ n \ 1)))). \end{array}$$

This function does not work yet because it does not receive a single argument. Before we can input an integer to the function, we must input a function to satisfy g so that the returning function is the desired factorial. Let's call this function X. Looking within the function, we see that the function X must take an integer and return an integer, that is, its type is $\mathbf{Z} \to \mathbf{Z}$. The function f will return the proper recursive function with the type $\mathbf{Z} \to \mathbf{Z}$, but only when supplied with the correct function X. Knowing the input and output types of f, we can write the type of f as

$$f : (\mathbf{Z} \to \mathbf{Z}) \to (\mathbf{Z} \to \mathbf{Z}).$$

What we need is a function X that, when applied to f, returns the correct recursive function.

We could try applying f to itself, i.e.

$$(f \quad f).$$

This does not work, because f expects something of type $\mathbf{Z} \to \mathbf{Z}$, but it is taking another f, which has the more complex type $(\mathbf{Z} \to \mathbf{Z}) \to (\mathbf{Z} \to \mathbf{Z})$. A function that has the correct input type is the identity combinator, $\lambda x.x$. Applying the identity function, we get

$$\begin{array}{rrl} (f \quad I) \Rightarrow & \lambda n.(\text{if} & (= \ n \ 0) \\ & & 1 \\ & & (* \ n \ (I \ (- \ n \ 1)))) \\ \Rightarrow & \lambda n.(\text{if} & (= \ n \ 0) \\ & & 1 \\ & & (* \ n \ (- \ n \ 1))), \end{array}$$

which is equivalent to

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) & \text{if } n > 0 \end{cases}.$$

We need to find the correct expression X such that when X is applied to f, we get the recursive factorial function. It turns out that the X that works is

$$\begin{array}{rl} X : (& \lambda x.(\lambda g.\lambda n.(\text{if } (= \ n \ 0) \ 1 \ (n \ (g \ (- \ n \ 1)))) \ \lambda y.((x \ x) \ y)) \\ & \lambda x.(\lambda g.\lambda n.(\text{if } (= \ n \ 0) \ 1 \ (n \ (g \ (- \ n \ 1)))) \ \lambda y.((x \ x) \ y))). \end{array}$$

Note that this has a structure similar to the non-terminating expression

$$(\lambda x.(x \quad x) \quad \lambda x.(x \quad x)),$$

and explains why the recursive function can keep going.

X can be defined as $(Y \quad f)$ where Y is the recursion combinator,

$$(f \quad X) \Rightarrow (f \quad (Y \quad f)) \Rightarrow (Y \quad f).$$

The recursion combinator that works for an applicative evaluation order is defined as

$$\begin{array}{rl} Y = \lambda f.(& \lambda x.(f \ \lambda y.((x \ x) \ y)) \\ & \lambda x.(f \ \lambda y.((x \ x) \ y))). \end{array}$$

The same combinator that is valid for normal order is

$$\begin{array}{rl} Y = \lambda f.(& \lambda x.(f \ (x \ x)) \\ & \lambda x.(f \ (x \ x))). \end{array}$$

How do we get from normal ordering to applicative order? Use η-expansion (that is, η-conversion in reverse). This is an example where η-conversion can have an impact on the termination of an expression.

3 Higher-Order Programming

Most imperative programming languages, e.g., Java and C++, do not allow us to treat functions or procedures as first-class entities, for example, we cannot create and return a new function that did not exist before. A function that can only deal with data values is called a *first-order* function. For example, `Increment`, whose type is $\mathbf{Z} \to \mathbf{Z}$, can only take integer values and return integer values. Programming only with first-order functions, is called *first-order* programming.

If a function can take another function as an argument, or if it returns a function, it is called a *higher-order* function.

For example, the `Curry` combinator, whose type is:

$$\texttt{Curry} : (\mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}) \rightarrow (\mathbf{Z} \rightarrow (\mathbf{Z} \rightarrow \mathbf{Z})).$$

is a higher-order (third order) function. It takes a function of type $\mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$ and returns a function of type $\mathbf{Z} \rightarrow (\mathbf{Z} \rightarrow \mathbf{Z})$. That is, `Curry` takes a first-order function and returns a second-order function. The ability to view *functions* as data is called *higher-order programming*.

3.1 Currying as a higher-order function

Higher-order programming is a very powerful technique, as shown in the following Oz example. Consider an exponential function, `Exp`, as follows:

```
declare Exp =
fun {$ B N}
   if N==0 then
      1
   else
      B * {Exp B N-1}
   end
end.
```

And recall the Curry combinator in Oz:

```
declare Curry =
fun {$ F}
   fun {$ X}
      fun {$ Y}
         {F X Y}
      end
   end
end.
```

We can create a function to compute the powers of 2, `TwoE`, by just using:

```
declare TwoE= {{Curry Exp} 2}.
```

If we want to create a `Square` function, using `Exp`, we can create a *reverse curry* combinator, `RCurry`, as:

```
declare RCurry =
fun {$ F}
   fun {$ X}
      fun {$ Y}
         {F Y X}
      end
   end
end,
```

where the arguments to the function are simply reversed.

We can then define `Square` as:

```
declare Square = {{RCurry Exp} 2}.
```

Lisp is a programming language that has demonstrated the use of higher-order programming to an extreme: a full Lisp interpreter written in Lisp, treats Lisp programs simply as data. This allows the interpreter program to be an input to itself.

3.2 Numbers in the λ-Calculus

The λ-calculus is a Turing-complete language, that is, any computable function can be expressed in the pure lambda calculus. In many of the examples, however, we have said: "assume that the calculus has been extended with numbers and conditionals".

Let us see one possible representation of numbers in the pure lambda calculus:

$$\begin{aligned} |0| &= \lambda x.x \\ |1| &= \lambda x.\lambda x.x \\ &... \\ |n+1| &= \lambda x.|n| \end{aligned}$$

That is, zero is represented as the identity combinator. Each succesive number $(n+1)$ is represented as a lambda abstraction (or procedure) that takes any value and returns the representation of its predecessor (n). You can think of zero as a first-order function, one as a second-order function, and so on.

In Oz, this would be written:

```
declare Zero = I

declare Succ =
fun {$ N}
   fun {$ X}
      N
   end
end
```

Using this representation, the number 2, for example, would be the lambda-calculus expression: $\lambda x.\lambda x.\lambda x.x$, or equivalently in Oz:

```
{Succ {Succ Zero}}
```

3.3 Booleans in the λ-Calculus

Now, let us see one possible representation of booleans in the pure lambda calculus:

$$\begin{aligned} |true| &= \lambda x.\lambda y.x \\ |false| &= \lambda x.\lambda y.y \\ |if| &= \lambda b.\lambda t.\lambda e.((b \quad t) \quad e) \end{aligned}$$

That is, `true` is represented as a function that takes two arguments and returns the first, while `false` is represented as a function that takes two arguments and returns the second. `if` is a function that takes:

- a function `b` representing a boolean value (either `true` or `false`),
- an argument `t` representing the *then* branch, and
- an argument `e` representing the *else* branch,

and returns either `t` if `b` represents `true`, or `e` if `b` represents `false`.

Let us see an example evaluation sequence for `(((if true) 4) 5)`:

$$
\begin{array}{ll}
 & (((\lambda b.\lambda t.\lambda e.((b \quad t) \quad e) \quad \lambda x.\lambda y.x) \quad 4) \quad 5) \\
\xrightarrow{\beta} & ((\lambda t.\lambda e.((\lambda x.\lambda y.x \quad t) \quad e) \; 4) \quad 5) \\
\xrightarrow{\beta} & (\lambda e.((\lambda x.\lambda y.x \quad 4) \quad e) \quad 5) \\
\xrightarrow{\beta} & ((\lambda x.\lambda y.x \quad 4) \quad 5) \\
\xrightarrow{\beta} & (\lambda y.4 \quad 5) \\
\xrightarrow{\beta} & 4
\end{array}
$$

Note that this definition of booleans works properly in normal evaluation order, but has problems in applicative evaluation order. The reason is that applicative order evaluates *both* the *then* and the *else* branches, which is a problem if used in recursive computations (where the evaluation may not terminate) or if used to guard improper operations (such as division by zero.)

In Oz, the following (uncurried) definitions can be used to test this representation:

```
declare LambdaTrue =
fun {$ X Y}
   X
end

declare LambdaFalse =
fun {$ X Y}
   Y
end

declare LambdaIf =
fun {$ B T E}
   {B T E}
end
```

Exercises

5. η-reduce the following λ-calculus expressions, if possible:
   (a) $\lambda x.(\lambda y.x\,x)$
   (b) $\lambda x.(\lambda y.y\,x)$
6. Use η-reduction to get from the applicative order Y combinator to the normal order Y combinator.
7. For a given function f, prove that $(f \quad (Y \quad f)) \Rightarrow (Y \quad f)$.
8. What would be the effect of applying the reverse currying combinator, `Rcurry`, to the function composition combinator?

```
declare Compose = fun {$ F G}
                     fun {$ X}
                        {F {G X}}
                     end
                  end
```

Give an example of using the `{RCurry Compose}` function.

9. Give an alternative representation of numbers in the lambda calculus (Hint: Find out about *Church numerals*). Test your representation using Oz.

10. Give an alternative representation of booleans in the lambda calculus (Hint: One possibility is to use $\lambda x.x$ for `true` and $\lambda x.\lambda x.x$ for `false`. You need to figure out how to define `if`.) Test your representation using Oz.