CSCI.4430/6969 Programming Languages
Lecture Notes

January 27, 2005

1 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 $\mathbf{Z} \rightarrow \mathbf{Z}$, 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 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*.

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.

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:

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

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).

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 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)`:

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

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

1. 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.

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

3. 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.