Chapter 2

λ Calculus

The λ *(lambda) calculus* [7] created by Church and Kleene in the 1930's is at the heart of functional programming languages. We will use it as a foundation for sequential computation. The λ calculus is Turing-complete, that is, any computable function can be expressed and evaluated using the calculus. The λ calculus is useful to study programming language concepts because of its high level of abstraction.

In the following sections, we will motivate the λ calculus and introduce its syntax and semantics (Section 2.1,) present the notion of scope of variables (Section 2.2,) the importance of the order of expression evaluation (Section 2.3,) the notion of combinators (Section 2.4,) currying (Section 2.5,) η-conversion ("*eta*-"conversion) (Section 2.6,) the sequencing and recursion combinators (Sections 2.7 and 2.8,) and higher-order programming in the λ calculus including an encoding of natural numbers and booleans (Section 2.9.)

2.1 Syntax and Semantics

We will briefly motivate the calculus and then introduce its syntax and semantics. The mathematical notation for defining a *function* is with a statement such as:

$$f(x) = x^2, \quad f : \mathbf{Z} \to \mathbf{Z},$$

where $\mathbf{Z}$ is the set of all integers. The first $\mathbf{Z}$ represents the *domain* of the function, or the set of values x can take. The second $\mathbf{Z}$ represents the *range* of the function, or the set containing all possible values of $f(x)$.

Suppose $f(x) = x^2$ and $g(x) = x + 1$. Traditional function *composition* is defined as:

$$f \circ g = f\left(g(x)\right).$$

With our functions f and g,

$$f \circ g = f\left(g(x)\right) = f(x+1) = x^2 + 2x + 1.$$

Similarly,

$$g \circ f = g\left(f(x)\right) = g(x^2) = x^2 + 1.$$

Therefore, function composition is not commutative.

In the λ calculus, we can use a different notation to represent the same concepts. To define a function $f(x) = x^2$, we instead may write[1]:

$$\lambda x.x^2.$$

Similarly for $g(x) = x + 1$ we write:

$$\lambda x.x + 1.$$

To describe a function *application* such as $f(2) = 4$, we write

$$(\lambda x.x^2 \quad 2) \Rightarrow 2^2 \Rightarrow 4.$$

The *syntax* for λ calculus expressions is

$$\begin{array}{rll} e ::= & v & \text{– variable} \\ | & \lambda v.e & \text{– functional abstraction} \\ | & (e \quad e) & \text{– function application} \end{array}$$

The *semantics* of the λ calculus, or the way of evaluating or simplifying expressions, is defined by the rule:

$$(\lambda x.E \quad M) \Rightarrow E\{M/x\}.$$

The new expression $E\{M/x\}$ can be read as "replace 'free' x's in E with M". Informally, a "free" x is an x that is not nested inside another lambda expression. We will cover free and bound variable occurrences in detail in Section 2.2.

For example, in the expression:

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

$E = x^2$ and $M = 2$. To evaluate the expression, we replace x's in E with M, to obtain:

$$(\lambda x.x^2 \quad 2) \Rightarrow 2^2 \Rightarrow 4.$$

In the λ calculus, all functions may only have one variable. Functions with more than one variable may be expressed as a function of one variable through *currying*. Suppose we have a function of two variables expressed in the standard mathematical way:

$$h(x, y) = x + y, \quad h : (\mathbf{Z} \times \mathbf{Z}) \rightarrow \mathbf{Z}.$$

With currying, we can input one variable at a time into separate functions. The first function will take the first argument, x, and return a function that will take

[1]Being precise, the λ calculus does not directly support number constants (such as '1') or primitive operations (such as '+' or x^2) but these can be encoded as we shall see in Section 2.9. We use this notation here for pedagogical purposes only.

the second variable, y, and will in turn provide the desired output. To create the same function with currying, let:

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

and:

$$g_x : \mathbf{Z} \to \mathbf{Z}.$$

That is, f maps every integer x to a function g_x, which maps integers to integers. The function $f(x)$ returns a function g_x that provides the appropriate result when supplied with y. For example,

$$f(2) = g_2, \quad \text{where} \quad g_2(y) = 2 + y.$$

So:

$$f(2)(3) = g_2(3) = 2 + 3 = 5.$$

In the λ calculus, this function would be described with currying by:

$$\lambda x.\lambda y.x + y.$$

To evaluate the function, we nest two function application expressions:

$$((\lambda x.\lambda y.x + y \quad 2) \quad 3).$$

We may then simplify this expression using the semantic rule, also called β-*reduction* (*"beta"-reduction*,) as follows:

$$((\lambda x.\lambda y.x + y \quad 2) \quad 3) \quad \Rightarrow \quad (\lambda y.2 + y \quad 3) \quad \Rightarrow \quad 2 + 3 \quad \Rightarrow \quad 5.$$

The composition operation $\circ$ can itself be considered a function, also called a *higher-order* function, that takes two other functions as its input and returns a function as its output; that is if the first function is of type $\mathbf{Z} \to \mathbf{Z}$ and the second function is also of type $\mathbf{Z} \to \mathbf{Z}$, then:

$$\circ : (\mathbf{Z} \to \mathbf{Z}) \times (\mathbf{Z} \to \mathbf{Z}) \to (\mathbf{Z} \to \mathbf{Z}).$$

We can also define function composition in the λ calculus. Suppose we want to compose the square function and the increment function, defined as:

$$\lambda x.x^2 \quad \text{and} \quad \lambda x.x + 1.$$

We can define function composition as a function itself with currying as follows:

$$\lambda f.\lambda g.\lambda x.(f \quad (g \quad x)).$$

Applying two variables to the composition function with currying works the same way as before, except now our variables are functions:

$$\begin{array}{ll} & ((\lambda f.\lambda g.\lambda x.(f \quad (g \quad x)) \quad \lambda x.x^2) \quad \lambda x.x + 1) \\ \Rightarrow & (\lambda g.\lambda x.(\lambda x.x^2 \quad (g \quad x)) \quad \lambda x.x + 1) \\ \Rightarrow & \lambda x.(\lambda x.x^2 \quad (\lambda x.x + 1 \quad x)). \end{array}$$

The resulting function gives the same results as $f(g(x)) = (x+1)^2$.

In the Scheme programming language we can use λ calculus expressions. They are defined using a similar syntax. To define a function we use the code:

```
(lambda([x y z ...])  expr)
```

where variables `x`, `y`, `z`, etc. are optional. Scheme syntax allows you to have functions of zero variables, one variable, or more than one variable. The code:

```
(lambda(x) (* x x))
```

describes the square function. Note that even common operations are considered functions and are always used in a prefix format. You may define variables (which may themselves be functions) with:

```
(define a b).
```

For example,

```
(define f (lambda(x) (* x x)))
```

defines a function $f(x) = x^2$. To perform a procedure call, use the code:

```
(f [x y z ...])
```

where `x`, `y`, `z`, etc. are additional parameters that `f` may require. The code:

```
(f 2)
```

evaluates $f(2) = 4$.

2.2 Free and Bound Variables in the λ Calculus

The process of simplifying (or β-reducing) in the λ calculus requires further clarification. The general rule is to find an expression of the form

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

called a *redex* (for *red*ucible *ex*pression,) and replace the "free" x's in E with M's. A *free* variable is one that is not *bound* by a lambda expression representing a functional abstraction. The functional abstraction syntax, $\lambda v.e$, defines the *scope* of the variable v to be e, and effectively *binds* occurrences of v in e. For example, in the expression

$$(\lambda x.x^2 \quad x+1)$$

the second x is bound by the λx, because it is part of the expression defining that function, *i.e.*, the function $f(x) = x^2$. The final x, however, is not bound by any function definition, so it is said to be free. Do not be confused by the fact that the variables have the same name. The two occurrences of the variable x are in different scopes, and therefore they are totally independent of each other.

An equivalent C program could look like this:

```
int f(int x) {
  return x*x;
}

int main() {
  int x;
  ...
  x = x + 1;
  return f(x);
}
```

In this example, we could substitute `y` (or any other variable name) for all `x` occurrences in function `f` without changing the output of the program. In the same way, the lambda expression

$$(\lambda x.x^2 \quad x+1)$$

is equivalent to the expression

$$(\lambda y.y^2 \quad x+1).$$

We cannot replace the final x, since it is unbound, or free. To simplify the expression

$$(\lambda x.(\lambda x.x^2 \quad x+1) \quad 2)$$

You could let $E = (\lambda x.x^2 \quad x+1)$ and $M = 2$. The only free x in E is the final occurrence so the correct reduction is

$$(\lambda x.x^2 \quad 2+1).$$

The x in x^2 is bound, so it is not replaced.

However, things get more complicated. It is possible when performing β-reduction to inadvertently change a free variable into a bound variable, which changes the meaning of the expression. In the statement

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

the second y is bound to λy whereas the final y is free. Taking $E = \lambda y.(x \quad y)$ and $M = (y \quad w)$, we could mistakenly arrive at the simplified expression

$$\lambda y.((y \quad w) \quad y).$$

But now both the second and third occurrences of y are bound, because they are both a part of the functional abstraction starting by λy. This is wrong because one of the y occurrences should remain free as it was in the original expression. To solve this problem, we can change the λy expression to a λz expression:

$$(\lambda x.\lambda z.(x \quad z) \quad (y \quad w)),$$

which again does not change the meaning of the expression. This process is called α-*renaming* (*"alpha"-renaming.*) Now when we perform the β-reduction, the original two y variable occurrences are not confused. The result is:

$$\lambda z.((y \quad w) \quad z)$$

where the free y remains free.

2.3 Order of Evaluation

There are different ways to evaluate λ calculus expressions. The first method is to always fully evaluate the arguments of a function before evaluating the function itself. This order is called *applicative order*. In the expression:

$$(\lambda x.x^2 \quad (\lambda x.x + 1 \quad 2)),$$

the argument $(\lambda x.x + 1 \quad 2)$ should be simplified first. The result is:

$$\Rightarrow (\lambda x.x^2 \quad 2 + 1) \Rightarrow (\lambda x.x^2 \quad 3) \Rightarrow 3^2 \Rightarrow 9.$$

Another method is to evaluate the left-most redex first. Recall that a redex is an expression of the form $(\lambda x.E \quad M)$, on which β-reduction can be performed. This order is called *normal order*. The same expression would be reduced from the outside in, with $E = x^2$ and $M = (\lambda x.x + 1 \quad 2)$. In this case the result is:

$$\Rightarrow (\lambda x.x + 1 \quad 2)^2 \Rightarrow (2 + 1)^2 \Rightarrow 9.$$

As you can see, both orders produced the same result. But is this always the case? It turns out that the answer is a qualified *yes*: only if both orders of expression evaluation **terminate.** Otherwise, the answer is *no* for expressions whose evaluation does not terminate. Consider the expression:

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

It is easy to see that reducing this expression gives the same expression back, creating an infinite loop. If we consider the expanded expression:

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

we find that the two evaluation orders are not equivalent. Using applicative order, the $(\lambda x.(x \quad x) \quad \lambda x.(x \quad x))$ expression must be evaluated first, but this process never terminates. If we use normal order, however, we evaluate the entire expression first, with $E = y$ and $M = (\lambda x.(x \quad x) \quad \lambda x.(x \quad x))$. Since there are no x's in E to replace, the result is simply y. It turns out that it is only in these particular non-terminating cases that the two orders may give different results. The *Church-Rosser theorem* (also called the *confluence property* or the *diamond property*) states that if a λ calculus expression can be evaluated in two different ways and both ways terminate, both ways will yield the same result.

Also, if there is a way for an expression to terminate, using normal order will cause the termination. In other words, normal order is the best if you want to avoid infinite loops. Take as another example the C program:

```
int loop() {
  return loop();
}

int f(int x, int y) {
  return x;
}

int main() {
  return f(3, loop());
}
```

In this case, using applicative order will cause the program to hang, because the second argument `loop()` will be evaluated. Using normal order will terminate because the unneeded `y` variable will never be evaluated.

Though normal order is better in this respect, applicative order is the one used by most programming languages. Why? Consider the function $f(x) = x+x$. To find $f(4/2)$ using normal order, we hold off on evaluating the argument until after placing the argument in the function, so it yields

$$f(4/2) = 4/2 + 4/2 = 2 + 2 = 4,$$

and the division needs to be done twice. If we use applicative order, we get

$$f(4/2) = f(2) = 2 + 2 = 4,$$

which only requires one division.

Since applicative order avoids repetitive computations, it is the preferred method of evaluation in most programming languages, where short execution time is critical. Some functional programming languages, such as Haskell, use *call-by-need* evaluation, which will avoid performing unneeded computations (such as `loop()` above) yet will memoize the values of needed arguments (such as `4/2` above) so that repetitive computations are avoided. This *lazy evaluation* mechanism is typically implemented with *thunks*, or zero-argument functions that *freeze* the evaluation of an argument until it is actually used, and *futures* or references to these thunks that trigger the *thawing* of the expression when evaluated, and keep its value for further immediate access.

2.4 Combinators

Any λ calculus expression with no free variables is called a *combinator*. Because the meaning of a lambda expression is dependent only on the bindings of its free variables, combinators always have the same meaning independently of the context in which they are used.

There are certain combinators that are very useful in the λ calculus:

The *identity* combinator is defined as:

$$I = \lambda x.x.$$

It simply returns whatever is given to it. For example:

$$(I \quad 5) \Rightarrow (\lambda x.x \quad 5) \Rightarrow 5.$$

The identity combinator in Oz[2] can be written:

```
declare I = fun {$ X} X end
```

Contrast it to, for example, a `Circumference` function:

```
declare Circumference = fun {$ Radius} 2*PI*Radius end
```

The semantics of the `Circumference` function depends on the definitions of `PI` and `*`. The `Circumference` function is, therefore, *not* a combinator.

The *application* combinator is:

$$App = \lambda f.\lambda x.(f \quad x),$$

and allows you to evaluate a function with an argument. For example

$$\begin{aligned} & ((App \quad \lambda x.x^2) \quad 3) \\ \Rightarrow \quad & ((\lambda f.\lambda x.(f \quad x) \quad \lambda x.x^2) \quad 3) \\ \Rightarrow \quad & (\lambda x.(\lambda x.x^2 \quad x) \quad 3) \\ \Rightarrow \quad & (\lambda x.x^2 \quad 3) \\ \Rightarrow \quad & 9. \end{aligned}$$

We will see more combinators in the following sections.

2.5 Currying

The *currying* higher-order function takes a function and returns a curried version of the function. For example, it would take as input the `Plus` function, which has the type

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

The type of a function defines what kinds of values the function can receive and what kinds of values it produces as output. In this case `Plus` takes two integers ($\mathbf{Z} \times \mathbf{Z}$,) and returns an integer ($\mathbf{Z}$.)

The definition of `Plus` in Oz is

```
declare Plus =
fun {$ X Y}
   X+Y
end
```

[2] We use examples in different programming languages (Scheme, C, Oz) to illustrate that the concepts in the λ calculus are ubiquituous and apply to many different sequential programming languages.

The currying combinator would then return the curried version of `Plus`, called `PlusC`, which has the type

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

Here, `PlusC` takes one integer as input and returns a function from the integers to the integers ($\mathbf{Z} \rightarrow \mathbf{Z}$.) The definition of `PlusC` in Oz is:

```
declare PlusC =
fun {$ X}
   fun {$ Y}
      X+Y
   end
end
```

The Oz version of the currying combinator, which we will call `Curry`, would work as follows:

`{Curry Plus}` $\Rightarrow$ `PlusC`.

Using the input and output types above, the type of the `Curry` function is

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

So the `Curry` function should take as input an uncurried function and return a curried function. In Oz, we can write `Curry` as follows:

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

2.6 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

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

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

Another type of operation possible on λ 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 x does not appear free in E.

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

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

We can also apply η-reduction to sub-expressions, *i.e.*, 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 statically reduce `{Inc 6}` to `{Increment 6}` avoiding one extra function call (or β reduction step) at run-time. 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 2.8.)

2.7 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 λ 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*.

2.8 Recursion Combinator

The *recursion* combinator allows defining recursive computations in the λ calculus. For example, suppose we want to implement a recursive version of the factorial function:

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

We could start by attempting to write the recursive function f in the λ calculus (assuming it has been extended with conditionals, and numbers) as follows[3]:

$$\begin{array}{l} f = \lambda n.(\text{if} \quad (= \quad n \ \ 0) \\ \qquad\qquad\quad 1 \\ \qquad\qquad\quad (* \ \ n \ \ (f \ \ (- \ \ n \ \ 1)))). \end{array}$$

The problem is that this function definition uses a free variable f, which is the very factorial function that we are trying to define. To avoid this circular definition, we can extend the definition with another functional abstraction (lambda expression) to take the factorial function as follows:

$$\begin{array}{l} f = \lambda g.\lambda n.(\text{if} \quad (= \quad n \ \ 0) \\ \qquad\qquad\quad\ \ 1 \\ \qquad\qquad\quad\ \ (* \ \ n \ \ (g \ \ (- \ \ n \ \ 1)))). \end{array}$$

[3]We will use prefix notation for mathematical expressions to be more consistent with function application syntax in the λ calculus as introduced in Section 2.1

Before we can input an integer to the function, we must input a function to satisfy g so that the returned function computes the desired factorial value. Let us call this function X. Looking within the function, we see that the function X must take an integer and return an integer, that is, X's 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 we apply f to it, it returns the correct recursive factorial function, that is, $(fX) = \lambda n.(\text{if}(= n0)1(*n(X(-n1)))) = X$, and so we need to solve the fixed point X of the function f, *i.e.*, the solution to the equation $(fX) = X$.

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:

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

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 f is applied to X, we get X, 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 λ calculus expression 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,

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

The recursion combinator that works for 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 normal order evaluation version of the recursion combinator is:

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

How do we get from the normal order evaluation recursion combinator to the applicative order evaluation recursion combinator? We use η-expansion (that is, η-conversion from right to left.) This is an example where η-conversion can have an impact on the termination of an expression.

2.9 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 primitive types (*i.e.*, not other functions) 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 (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*.[4]

2.9.1 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:

[4]The ability in some imperative programming languages to pass pointers to functions, as in a generic sort routine that can receive different element ordering functions, is only half of the equation. Truly higher-order programming requires the ability to create arbitrary *new* functions as done in the currying example in the text.

```
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}.
```

To illustrate the execution of this expression, consider the following Oz computation steps (equivalent to two β-reduction steps in the λ calculus):

```
TwoE = {{Curry Exp} 2}
     = {{fun {$ F}
            fun {$ X}
               fun {$ Y}
                  {F X Y}
               end
            end
         end Exp} 2}
     = {fun {$ X}
           fun {$ Y}
              {Exp X Y}
           end
       end 2}
     = fun {$ Y}
         {Exp 2 Y}
       end
```

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}.
```

Higher-order programming enables us to view functions as data. Lisp is a functional programming language that uses the same syntax for programs and for data (lists.) This enables *meta-circular* interpretation: a full Lisp interpreter, written in Lisp, can be an input to itself.

2.9.2 Numbers in the λ Calculus

The λ calculus is a Turing-complete language, that is, any computable function can be expressed in the pure λ calculus. In many of the previous examples, however, we have used numbers and conditionals.

Let us see one possible representation of numbers in the pure λ 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 functional (or procedural) abstraction 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 λ calculus expression: $\lambda x.\lambda x.\lambda x.x$, or equivalently in Oz:

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

2.9.3 Booleans in the λ Calculus

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

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

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):

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

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.) The applicative order evaluation versions of `if`, `true`, and `false` can *wrap* the *then* and *else* expressions inside functional abstractions, so that they are values and do not get prematurely evaluated, similarly to how the applicative order evaluation sequencing operator wrapped the expression to be evaluated last in the sequence (see Section 2.7.)

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
```

2.10 Exercises

1. α-convert the outer-most x to y in the following λ calculus expressions, if possible:

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

   (b) $\lambda x.(\lambda x.x\, y)$

2. β-reduce the following λ calculus expressions, if possible:

   (a) $(\lambda x.\lambda y.(x\, y)\, (y\, w))$

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

3. Simulate the execution of `Square` in Oz, using the definition of `RCurry` given in Section 2.9.1.

4. Using the number representation in Section 2.9.2, define functions `Plus`, `PlusC` (its curried version) in Oz, and test them using Mozart (Oz's run-time system.)

5. Write a function composition combinator in the λ calculus.

6. Define a curried version of `Compose` in Oz, `ComposeC`, without using the `Curry` combinator. (Hint: It should look very similar to the λ calculus expression from Exercise 5.)

7. η-reduce the following λ calculus expressions, if possible:

   (a) $\lambda x.(\lambda y.x\, x)$

   (b) $\lambda x.(\lambda y.y\, x)$

8. Use η-reduction to get from the applicative order Y combinator to the normal order Y combinator.

9. 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.

10. Define a + operation for the representation of numbers given in Section 2.9.2. Test your addition operation in Oz.

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

12. Give an alternative representation of booleans in the λ 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.

13. Create an alternative representation of booleans in the λ calculus so that conditional execution works as expected in applicative evaluation order (Hint: Use the strategy of wrapping the *then* and *else* branches to turn them into values and prevent their premature evaluation.) Test your representation using Oz.

14. For a given function f, prove that $(f \quad (Y \quad f)) \Rightarrow (Y \quad f)$.