Functional Languages

The Lambda Calculus

Functional languages started as an implementation of the ideas in the lambda calculus. In the lambda calculus there are three basic types of expression: a variable, such as x or y, a function such as λx.x, where the body of the function is any expression, and an application which is just two expression one after the other, as in a b, or c d. With the addition of parentheses, we have a complete language based on simple substitution. It is usual to also allow arithmetic expression with infix operators, such as x + y. The application of a function to its arguments can be 'reduced' or made simpler by following the substitution rule. E.g. the expression:

((λx.λy.x + y) 2) 3

Reduces to 5 by first binding 2 to x and substituting to give λy.2 + y, and then binding 3 to y an substituting to give 2 + 3, or 5.

Functions as first class arguments

The calculus does not care whether arguments to functions are values, like numbers, variables, like x, or functions. So the expression

((λx.λf.f x) 2) (λy.y + 3)

reduces to

(λf.f 2) (λy.y + 3)

by binding 2 to x and substituting. If we now bind f to the function λy.y + 3, an substitute, we get

(λy.y + 3) 2

which is 2 + 3, or 5.

Functions may also return functions as values, as in the first reduction above.

(λx.λf.f x) 2

Reduces to λf.f x. i.e. the function with parameter x returns the function with parameter f as a value.

These properties of the lambda calculus, i.e. treating functions as first class values, is taken over by all functional languages.

Eager evolution using pass-by-value

Another property of the lambda calculus is that reductions can proceed by first reducing the argument. E.g. in

(λx.x + 2) ((λy.y + 1) 4)

we first reduce the arguments to 4 +1, or 5, then apply λx.x + 2 to 5 to get 5 + 2, or 7. This style of reduction is carried over to functional languages as evaluation of arguments before binding their values to parameters. The mechanism used is pass-by-value. This is called eager evaluation. It is also possible to reduce the above expression by binding x directly to the argument. This gives

((λy.y + 1) 4) + 2

after substitution, which reduces to 7, as before. There is a theorem that says the two styles of reduction must give the same result, so there is a definite choice. The second style of reduction is lazy evaluation, which is a feature of some functional languages, and the mechanism used is pass-by-name.

LISP, the first functional language

John McCarthy and his students turned the lambda calculus into a language called LISP. They wrote an interpreter for the language which essentially carried out reduction using the 'universal evaluator' called eval. eval is a function that, when passed any expression in the language, returns its value. In LISP, all operations are represented in prefix form, so the last example would be written

```
((lambda (x)  (+ x 2)) ((lambda (y) (+ y 1)) 4))
```

Notice the greek letter λ is replaced by the word lambda, and the function's parameter is written in parentheses. Notice also that all applications must be surrounded by parentheses, e.g. (X Y) where X must be a function (or evaluate to a function) and Y is its argument. A sketch of the function eval is:

```
eval(exp) =
    if exp is a variable (such as x), return its binding
    if exp is a function (such as (lambda (x) x)), return it
    if exp is an application (such as (X Y),
        find the value of X by calling eval(X) (this should be a function),
        find the value of Y by calling eval(Y),
        bind the parameter of eval(X) to eval(Y),
        evaluate the body of function eval(X) with the new binding
```

LISP's model of computation is thus simply the valuation of expressions. There are no variables as in the imperative languages and there is no assignment. In fact, once a parameter has been bound to the value of its argument, it can never change. This means that the value of any expression is given by its form, since there are no hidden side effects caused by assignment. This property of functional languages is called 'referential transparency'.

In the full version of LISP, it is possible also to define functions by giving them a name (rather than just anonymous lambda expressions). This allows recursion since a function can then refer to itself by name. These functions can have any number of parameters, rather than just one as in the lambda calculus. The addition of a conditional form and some testing operations makes the language complete. Here is the factorial function, in LISP:

```
(defun factorial (n)
    (cond ((eq n 0) 1)
          (t (* n (factorial (- n 1))))))
```

The conditional form is a pseudo-function (or 'special' function) that is handled specially by eval. It can have any number of 'clauses', each one consisting of an expression which acts as a test, and an expression to act as a return value. The first test in factorial is

whether n, the parameter, equals 0. If these returns true, then the value of factorial is 1. If this test fails, the second clause is tried. Here the test is just the expression t, which, by definition, always evaluates to true, so it will always succeed. The value of factorial is then n times the value of factorial with n – 1 as the argument. A derivation of a particular expression can be given as a series of equalities:

```
(factorial 3)
= (* 3 (factorial (- 3 1)))
= (* 3 (factorial 2))
= (* 3 (* 2 (factorial (- 2 1))))
= (* 3 (* 2 (factorial 1)))
= (* 3 (* 2 (* 1 (factorial (- 1 1)))))
= (* 3 (* 2 (* 1 (factorial 0))))
= (* 3 (* 2 (* 1 1)))
= 6
```

Notice that this derivation gives the value of 3! At each step, just in a different form. This is referential transparency at work.

In fact, we can even express the recursion in factorial without using recursion directly. LISP has a form that allows the temporary binding of a variable to a value called let. The syntax is:

```
(let ((x e1)) e2)
```

We can then express factorial as:

```
(defun factorial (n)
   (let ((fact1 '(lambda (f n) (cond ((eq n 0) 1) (t (* n (f f (- n 1))))))))
      (fact1 fact1 n)))
```

In this function fact1 is bound to the lambda expression that computes factorial. However, the 'recursive' call in its body is in fact just calling the function passed through the parameter, and passing it also in the call. So the next call to f gets itself as an argument and also n reduced by 1. A sample derivation is:

```
(factorial 3)
= (fact1 fact1 3)
= (cond ((eq 3 0) 1) (t (* 3 (fact1 fact1 2))))
= (* 3 (fact1 fact1 2))
= (*3 (cond ((eq 2 0) 1) (t (* 2 (fact1 fact1 1)))))
= (* 3 (*2 (fact1 fact1 1)))
= (* 3 (* 2 (cond ((eq 1 0) 1) (t (* 1 (fact1 fact1 0))))))
= (* 3 (* 2 (*1 (fact1 fact1 0))))
= (* 3 (* 2 (* 1 (cond ((eq 0 0) 1) (t (* 0 (fact1 fact1 -1)))))))
= (* 3 (* 2 (* 1 1)))
= 6
```

This derivation is the same as the recursive version, except for the extra parameter in the call. In lambda calculus form it is:

$$\lambda n.((\lambda g.g\ g\ n)\ (\lambda f.\lambda n.n=0 \rightarrow 1 \square n*(f\ f\ n))$$

using a special form of the conditional for the lambda calculus.