Functional Programming Languages

BENJAMIN GOLDBERG

New York University ⟨goldberg@cs.nyu.edu⟩

Functional programming languages are a class of languages designed to reflect the way people think mathematically, rather than reflecting the underlying machine. Functional languages are based on the lambda calculus, a simple model of computation, and have a solid theoretical foundation that allows one to reason formally about the programs written in them. The most commonly used functional languages are Standard ML, Haskell, and "pure" Scheme (a dialect of LISP), which, although they differ in many ways, share most of the properties described here. For a complete description of these languages and of functional programming in general, see Bird and Wadler [1988], Paulson [1991], Sussman and Abelson [1985], Hudak et al. [1992], Milner et al. [1990], Artificial Intelligence Laboratory [1992], and Hudak [1989].

In contrast to the usual imperative languages (e.g., C, Fortran, and Ada), in which variables represent cells in memory that can be modified using the assignment operator = (or :=), functional languages view the use of the = operator as an expression of an equation. For example, if a functional program contains the declaration

```
let x = f(y)
```

then this would introduce the name `x` and assert that the equation `x = f(y)` is true. There is no notion of a memory cell, and certainly no notion that somehow later in the program `x` might change (so that the equation would become false).

If one were to write

```
let x = x + 1
```

in a functional program, this would represent an equation with no finite solution (and would either be rejected by a compiler or result in a nonterminating computation), whereas in C this would increment the contents of the memory cell denoted by `x`.

Function names are introduced in a similar way. The declaration

```
let f(x, y) = x + y
```

introduces the function `f` and states that `f(x, y)` and `x + y` are equal, for any `x` and `y`. The expression on the right-hand side, the body of `f`, cannot be a sequence of statements modifying the values of `x` and `y` (and perhaps other variables). As in mathematics, a function is a single-valued relation such that, given the same argument(s), it will return the same result. This is certainly not the case in the imperative programs.

Because variables in a functional program cannot be modified, repetition must be expressed in a functional program via recursion rather than through the use of loops. Consider the factorial function, defined formally as:

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

In a functional language the executable-definition of factorial is generally written as

```
let factorial(n)=
    if n == 0 then 1
    else n*factorial(n − 1)
```

and follows directly from the formal definition (where == is the equality comparison operator).

This is in contrast to the C version

```
int fac(int n)
{int prod = 1;
 for (int i = 0;
       i <= n; i++)
     prod = prod * i ;
}
```

which can be understood only by tracing the sequence of modifications to the variables prod and i during the iteration.

Functional languages exhibit a property called *referential transparency*, which essentially means that "like can be replaced by like." For example, the expression

```
f(y) + f(y)
```

is equivalent to

```
let x = f(y)
 in x + x
```

in which the two original occurrences of `f(y)` are replaced by `x`. This follows directly from the fact that the declaration `x= f(y)` denotes an equation in a functional language, but certainly would not be the case in an imperative language. In an imperative language, the first call to `f` might change the value of a variable used in the second call to `f`. This kind of behavior, known as a *side-effect*, cannot occur in a functional language.

HIGHER-ORDER FUNCTIONS

Most functional languages support functions that operate over functions—that is, functions that take other functions as parameters and/or return functions as values. A simple example is the `compose` function, defined as

```
let compose(f,g)=let h(x)=f(g(x))
                 in h
```

In this case, the parameters to `compose`, `f`, and `g`, must both be functions. In addition, the value returned by `compose`, namely `h`, is also a function, defined by the equation `h(x) = f(g(x))`.

As another example of the use of higher-order functions, consider again the factorial function. It might be argued that the formal definition of factorial given above was tailored to suit the recursive nature of the definition of factorial in the functional language, and that a more reasonable and common definition of factorial is

$$n! = \prod_{i=1}^{n} i$$

The product operator, $\prod$, is a very useful operator that has the general form:

$$\prod_{i=m}^{n} f(i)$$

for some initial value `m`, some final value `n`, and some function `f`. In a functional language, $\prod$ can easily be written as a higher-order function of three parameters, `m`, `n`, and `f`, defined by

```
prod(m, n, f)
    = if m == n then f(m)
      else f(m) * prod(m, n, f)
```

where == is the equality comparison operator. Thus, factorial can be defined as:

```
let fac(n) = let f(i) = i
             in prod(1,n,f)
```

and the power function, computing x^n, can be defined as

```
let power(x,n) = let f(i) = x
                 in prod(1,n,f)
```

NON-STRICT FUNCTIONAL LANGUAGES

In most programming languages, a function call of the form

$$\mathtt{f}(e_1, \ldots, e_n)$$

causes the argument expressions, $e_1 \ldots e_n$, to be evaluated before the body of the function `f` is executed. This is also the case in ML and Scheme. However, in a class of functional languages called *non-strict* functional languages, of which Haskell is the most popular, no function evaluates its arguments unless they are needed. For example, the function `f` defined by

```
let f(x,y,z) = if x == 0
               then y + 1 else z
```

always evaluates its first, parameter, `x`, but only one of `y` or `z` will be evaluated. Thus, in the call

```
f(4, g(3), h(2)}
```

the expression `g(3)` will not be evaluated.

Non-strictness is attractive for two reasons. First, it frees the programmer from worrying about various issues of control, such as choosing the correct order of evaluation among various expressions. In a non-strict language, an expression program won't be evaluated unless it is needed. For example, in a producer-consumer problem, the producer is guaranteed to produce only what the consumer needs.

Another feature of non-strictness is that it allows the construction of infinite data structures. To see this, consider the recursive definition

```
let ones = 1 :: ones
```

where the :: operator constructs a list whose first element is the left operand, in this case 1, and whose subsequent elements come from the right operand, in this `ones`. That is, `ones` is a list that is recursively defined to be 1 followed by all the element of `ones`. Clearly, the only solution to this equation is if `ones` is the infinite list of 1's. In a strict language, where :: requires the value of its arguments, the evaluation of the right-hand side of the equation would never terminate. However, in a non-strict language, the :: does not evaluate its operands until they are actually needed. Ultimately, only those elements of *ones* that are required by other parts of the programs will be computed. The rest of ones (which is infinite) would be left uncomputed.

RESEARCH ISSUES IN FUNCTIONAL LANGUAGES

The functional language research community is very active in a number of areas. Of particular, interest is improving the speed of functional language implementations. There are two primary approaches: through compiler-based program analysis and optimization techniques, and through the execution of functional programs on parallel computers. Another area of research attempts to increase the expressiveness of functional languages for applications in which the notion of state and the change of state (through assignment) is seen as necessary in conventional programs. New constructs have been proposed that, although they appear to be side-effect operators such as array updates, actually preserve the referential transparency property.

REFERENCES

ARTIFICIAL INTELLIGENCE LABORATORY 1992. Report on the algorithmic language scheme. Tech. Rep., Cambridge, MA, (Nov.), revised.

BIRD, R. AND WADLER, P. 1988. *Introduction to Functional Programming*. Prentice Hall, Englewood Cliffs, NJ.

HUDAK, P., ET AL. 1992. Report on the programming language Haskell. *SIGPLAN Not. 27*, 5, Section R.

HUDAK, P. 1989. The conception, evolution, and application of functional programming languages. *ACM Comput. Surv. 21*, 3, 359–411.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, MA.

PAULSON, L. C. 1991. *ML for the Working Programmer*. Cambridge University Press, Cambridge, UK.

SUSSMAN, G. AND ABELSON, H. 1985. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.