Chapter 2

Functional languages

Programming languages are said to be *functional* when the basic way of structuring programs is the notion of *function* and their essential control structure is *function application*. For example, the Lisp language [22], and more precisely its modern successor Scheme [31, 1], has been called functional because it possesses these two properties.

However, we want the programming notion of function to be as close as possible to the usual mathematical notion of function. In mathematics, functions are "first-class" objects: they can be arbitrarily manipulated. For example, they can be composed, and the composition function is itself a function.

In mathematics, one would present the *successor* function in the following way:

$$\begin{array}{rll} \mathit{successor}: & \mathbf{N} & \longrightarrow \mathbf{N} \\ & n & \longmapsto n+1 \end{array}$$

The functional composition could be presented as:

$$\begin{array}{rll} \circ: & (A \Rightarrow B) \times (C \Rightarrow A) & \longrightarrow (C \Rightarrow B) \\ & (f, g) & \longmapsto (x \longmapsto f\ (g\ x)) \end{array}$$

where $(A \Rightarrow B)$ denotes the space of functions from A to B.

We remark here the importance of:

1. the notion of *type*; a mathematical function always possesses a *domain* and a *codomain*. They will correspond to the programming notion of type.

2. lexical binding: when we wrote the mathematical definition of *successor*, we have assumed that the addition function + had been previously defined, mapping a pair of natural numbers to a natural number; the meaning of the *successor* function is defined using the *meaning* of the addition: whatever + denotes in the future, this *successor* function will remain the same.

3. the notion of *functional abstraction*, allowing to express the behavior of $f \circ g$ as $(x \longmapsto f\ (g\ x))$, i.e. the function which, when given some x, returns $f\ (g\ x)$.

ML dialects (cf. below) respect these notions. But they also allow non-functional programming styles, and, in this sense, they are functional but not *purely functional*.

2.1 History of functional languages

Some historical points:

- 1930: Alonzo Church developed the λ-calculus [6] as an attempt to provide a basis for mathematics. The λ-calculus is a formal theory for functionality. The three basic constructs of the λ-calculus are:
  - variable names (e.g. x, y,...);
  - application (MN if M and M are terms);
  - functional abstraction ($\lambda x.M$).

  Terms of the λ-calculus represent functions. The pure λ-calculus has been proved inconsistent as a logical theory. Some *type systems* have been added to it in order to remedy this inconsistency.
- 1958: Mac Carthy invented Lisp [22] whose programs have some similarities with terms of the λ-calculus. Lisp dialects have been recently evolving in order to be closer to modern functional languages (Scheme), but they still do not possess a type system.
- 1965: P. Landin proposed the ISWIM [18] language (for "If You See What I Mean"), which is the precursor of languages of the ML family.
- 1978: J. Backus introduced FP: a language of combinators [3] and a framework in which it is possible to reason about programs. The main particularity of FP programs is that they have no variable names.
- 1978: R. Milner proposes a language called ML [11], intended to be the *metalanguage* of the LCF proof assistant (i.e. the language used to program the search of proofs). This language is inspired by ISWIM (close to λ-calculus) and possesses an original type system.
- 1985: D. Turner proposed the Miranda [36] programming language, which uses Milner's type system but where programs are submitted to *lazy evaluation*.

Currently, the two main families of functional languages are the ML and the Miranda families.

2.2 The ML family

ML languages are based on a sugared[1] version of λ-calculus. Their evaluation regime is *call-by-value* (i.e. the argument is evaluated before being passed to a function), and they use Milner's type system.

The LCF proof system appeared in 1972 at Stanford (Stanford LCF). It has been further developed at Cambridge (Cambridge LCF) where the ML language was added to it.

From 1981 to 1986, a version of ML and its compiler was developed in a collaboration between INRIA and Cambridge by G. Cousineau, G. Huet and L. Paulson.

[1] i.e. with a more user-friendly syntax.

In 1981, L. Cardelli implemented a version of ML whose compiler generated native machine code.

In 1984, a committee of researchers from the universities of Edinburgh and Cambridge, Bell Laboratories and INRIA, headed by R. Milner worked on a new extended language called Standard ML [28]. This core language was completed by a module facility designed by D. MacQueen [23].

Since 1984, the Caml language has been under design in a collaboration between INRIA and LIENS[2]). Its first release appeared in 1987. The main implementors of Caml were Ascánder Suárez, Pierre Weis and Michel Mauny.

In 1989 appeared Standard ML of New-Jersey, developed by Andrew Appel (Princeton University) and David MacQueen (Bell Laboratories).

Caml Light is a smaller, more portable implementation of the core Caml language, developed by Xavier Leroy since 1990.

2.3 The Miranda family

All languages in this family use *lazy evaluation* (i.e. the argument of a function is evaluated if and when the function needs its value—arguments are passed unevaluated to functions). They also use Milner's type system.

Languages belonging to the Miranda family find their origin in the SASL language [34] (1976) developed by D. Turner. SASL and its successors (KRC [35] 1981, Miranda [36] 1985 and Haskell [15] 1990) use *sets of mutually recursive equations* as programs. These equations are written in a *script* (collection of declarations) and the user may evaluate expressions using values defined in this script. LML (Lazy ML) has been developed in Göteborg (Sweeden); its syntax is close to ML's syntax and it uses a fast execution model: the G-machine [16].

[2]Laboratoire d'Informatique de l'Ecole Normale Supérieure, 45 Rue d'Ulm, 75005 Paris