Chapter 11
The Functional Paradigm

In this chapter, we present the main properties of functional programming. In functional programming, computation proceeds by rewriting functions and not by modifying the state. The fundamental characteristic of the languages in this paradigm, at least in their "pure" form, is precisely that of not possessing the concept of memory (and therefore side effect). Once an environment is fixed, an expression always denotes the same value.

We will discuss the pure paradigm in the first sections, explaining the fundamental aspects. Functional programming languages, however, merge these "pure" ingredients in a context that adds many other mechanisms; we will review them in Sect. 11.3.

We will touch on the SECD machine, an abstract machine for higher-order functional languages which constitutes the prototype of many real implementations.

We will, at this point, be in a position to discuss the reasons why the functional programming paradigm is interesting with respect to ordinary imperative languages.

The chapter concludes with a more theoretical section which provides a succinct introduction to the λ-calculus, a formal system for computability which inspires all functional languages and which has, since the time of ALGOL and LISP, been a constant model for the design of programming languages.

11.1 Computations without State

Though developed and abstract, all conventional languages base their computational model on the transformation of the *state*. The heart of this model is the concept of *modifiable variable*, that is, a container with a name to which, during the computation, can be assigned different values, while the same association is always maintained in the environment. Correspondingly, the principal construct in conventional languages is *assignment*, which modifies the value contained in a variable (but does not modify the association between the name of the variable and the location to which it corresponds; it modifies the r-value but not the l-value, which is fixed once and for all when the variable is declared).

Conventional languages differ in the their level of abstraction, their types, in the constructs which allow the manipulation of variables, but they all share the same computational model. This is an abstract view of the underlying, conventional physical machine. Computation proceeds by modifying values stored in locations. This is an extremely important model. It is called the *von Neumann Machine*, named after the Hungarian-American mathematician who, in the 1940s, saw that the Turing Machine (see box on page 61) could be engineered into a physical prototype, thereby originating the modern computer.

This is not, however, the only possible model upon which to base a programming language. It is possible to compute without using modifiable variables, that is without referring to the concept of state. The computation proceeds not by modifying the state but by *rewriting* expressions, that is by changes that take place only in the environment and do not involve the concept of memory. If there are no modifiable variables, there is no longer a need for assignment, that is the principal command in conventional languages. The entire computation will be expressed in terms of the sophisticated modification of the environment in which the possibility of manipulating (higher-order, see Sect. 7.2) functions plays a fundamental role.

Without assignment, iteration also looses its real sense. A loop can only repeatedly modify the state until the values of certain variables satisfy a guard. The reader already knows (see Chap. 6) that iterative and recursive constructs are two mechanisms permitting infinitely long (and possibly divergent) computations. In the stateless computational model, iteration disappears, recursion remains and becomes the fundamental construct for sequence control.

Higher-order functions and recursion are the basic ingredients of this stateless computational model. The programming languages which presuppose this model are called *functional languages* and the paradigm that results from this is called the *functional programming paradigm*.

This is a paradigm as old as the imperative one. Since the 1930s, beside the Turing Machine, there has existed the λ-calculus, an abstract model for characterising computable functions that is based on exactly the concept we have been briefly explaining. LISP was the first programming language explicitly inspired by the λ-calculus and many others have followed in the intervening years (Scheme, ML, in all its different dialects, Miranda, Haskell, to name only the most common). Among these, only Miranda and Haskell are "purely functional"; the others also have imperative components (often to assist programmers accustomed to conventional languages) but the structure of these languages assign a subordinate role to them.

In this chapter, we undertake an introduction to the pure functional paradigm; it is not a superficial one and we will discuss different general aspects of the paradigm. To make the treatment more concrete, it is desirable to refer to a specific language rather than to use a neutral pseudocode. ML has been chosen for this because, through its coherence and elegance of design, it is the most suited of them all to didactic presentation. It is not our aim to introduce the language (for which, the reader should refer to the bibliography), but only to use the syntax in an instrumental fashion to discuss some general questions that apply also to other functional languages.

11.1.1 Expressions and Functions

In the usual mathematical practice, there is some ambiguity about when we are defining a function and when we are applying it to a value. It is not uncommon to encounter expressions of the kind:

> Let $f(x) = x^2$ be the function that associates with x its square. If now we have $x = 2$, it follows that $f(x) = 4$.

The syntactic expression $f(x)$ is used to denote two things that are quite different: the first time, it serves to introduce the *name*, f, of a specific function; the second time, it serves to denote the result of *applying* the function f to a specified value. In mathematical practice, this ambiguity is completely innocuous because the context helps us to distinguish which of the two uses is intended. The same does not hold for an artificial language (such as a programming language) which describes functions. Here, it is appropriate to distinguish the two cases.

When a mathematician asserts that they are defining the function, $f(x)$, in reality, they are defining the function f with one *formal* parameter, x, which serves to indicate the transformation that f applies to its argument. To distinguish linguistically between the name and the "body" of the function, following ML syntax, we can write:

```
val f = fn x => x*x;
```

The reserved word `val` introduces a declaration. The declaration is used to extend the environment with a new association between a name and a value. In our case, the name f is bound to the transformation of x into $x * x$. In all functional languages, functions are *expressible* values; that is, they can be the result of the evaluation of a complex expression. In our case, the expression on the right of the $=$ and introduced by `fn` is an expression that denotes a function.

For the application of a function to an argument, we retain the traditional notation, writing `f(2)` or `(f 2)`, or `f 2`, for the expression that results from the application of a function f to the argument 2. We can also use `val` to introduce new names, as in:

```
val four = f 2;
```

The introduction of specific syntax for an expression which denotes a function has an important consequence. It is possible to write (and possibly apply) a function without having necessarily to assign it a name. For example, the expression

```
(fn y => y+1) (6);
```

has the value 7 which results from the application of the (anonymous) function `fn y => y+1` to the argument `6`. To make the notation less imposing, we assume (as does ML) that application can be denoted by simple juxtaposition (that is without parentheses) and that is associates to the left (prefix notation). If `g` is the name of a function,

```
g a1 a2 ... ak
```

will mean:

```
(...((g a1) a2)... ak).
```

Nothing prevents one functional expression from appearing inside another, as in

```
val add = fn x => (fn y => y+x);
```

The value `add` is a function which, given an argument, `x`, returns an anonymous function which, given an argument `y`, returns `x+y`. We can use `add` in many different ways:

```
val three = add 1 2;
val addtwo = add 2;
val five = addtwo 3;
```

Note that, in particular, `addtwo` is a function which is obtained as the result of the evaluation of another expression.

The notation which uses `val` and `fn` is important but is a little verbose. ML also allows the use of a more compact notation which resembles the usual way of defining functions in a programming language. The first function, `f`, which we defined (the one which computes the square of its argument) could also be defined as follows:

```
fun f x = x*x;
```

In general, a definition of the form:

```
fun F x1 x2 ... xn = body;
```

is only syntactic sugar (that is, in programming language jargon, only a nicer abbreviation) for:

```
val F = fn x1 => (fn x2 => ... (fn xn => body)...);
```

As a last example of functions which manipulate functions, this time in the form of a formal parameter, consider the definition:

```
fun comp f g x = f(g(x));
```

This returns the function composed of its first two arguments (which are, in their turn, functions).

Finally, every functional program permits the definition of recursive functions. Assuming that we have available a *conditional expression* (which in ML is written with the familiar `if then else` syntax), we can define the usual factorial as:

```
fun fatt n = if n=0 then 1 else n*fatt(n-1);
```

11.1.2 Computation as Reduction

If we exclude arithmetic functions (which we can assume predefined with the usual semantics) and the conditional expression, at the conceptual level, we can describe the procedure used to transform a complex expression into its value (*evaluation*) as the process of *rewriting*. We call this process *reduction*. In a complex expression, a subexpression of the form "function applied to an argument" is textually replaced by the body of the function in which the formal parameter is replaced, in its turn, by the actual parameter.[1] We can compute a simple expression using this computational model (we use → to indicate a reduction step).

```
fact 3 → (fn n => if n=0 then 1 else n*fact(n-1)) 3
       → if 3=0 then 1 else 3*fact(3-1)
       → 3*fact(3-1)
       → 3*fact(2)
       → 3*((fn n => if n=0 then 1 else n*fact(n-1)) 2)
       → 3*(if 2=0 then 1 else 2*fact(2-1))
       → 3*(2*fact(2-1))
       → 3*(2*fact(1))
       → 3*(2*((fn n => if n=0 then 1 else n*fact(n-1)) 1))
       → 3*(2*(if 1=0 then 1 else 1*fact(1-1))
       → 3*(2*(1*fact(0))
       → 3*(2*(1*((fn n => if n=0 then 1 else n*fact(n-1)) 1)))
       → 3*(2*(1*(if 0=0 then 1 else n*fact(n-1))))
       → 3*(2*(1*1))
       → 6
```

Note how, with the exception of arithmetic calculations and the use of the conditional expression, all of the rest of the computation proceeds by symbolic manipulation of strings: no variables, no update of stacked variables. Figure 11.1 contains another example of pure symbolic manipulation. (It is a complicated way to write the identity function!) (While studying the figure, recall that `fun` is just an abbreviation for `val ... fn`.)

Finally, the reader will have no have difficulty in convincing themselves that, given the definition

```
fun r x = r(r(x));
```

each computation which involves an evaluation of *r* resolves into an infinite rewriting. We say, in such a case, that the computation *diverges* and that the result is undefined.

[1]The reader will certainly have recognised in this description the "copy rule" we already discussed in the context of the by-name parameter passing. For the time being, we will be vague about the exact semantics to assign to this process—all of Sect. 11.2 will be devoted to it.

```
fun K x y = x;
fun S p q r = p r (q r);
val a = ...;

S K K a → (fn p => (fn q => (fn r => p r (q r)))) K K a
        → (fn q => (fn r => K r (q r))) K a
        → (fn r => K r (K r)) a
        → K a (K a)
        → (fn x => (fn y => x)) a (K a)
        → (fn y => a) (K a)
        → a
```

Fig. 11.1 Some definitions and a computation using rewriting

11.1.3 The Fundamental Ingredients

In these first subsections, we have introduced all the fundamental ingredients of the pure functional paradigm. We can make this precise and summarise the main concepts in the following way.

From the syntactic viewpoint, a language such as the one under consideration has no commands (there being no state to modify using side effects) but only expressions. Apart from possible values and primitive operators for data (such as integer, boolean, characters, etc.) and the conditional expression, the two principal constructs for defining expressions are:

- *Abstraction* which, given any expression, `exp` and an identifier, `x`, allows the construction of an expression `fn x => exp` denoting the function that transforms the formal parameter `x` into `exp` (the expression `exp` is "abstracted" from the specific value bound to `x`).
- The *application* of an expression, `f_exp`, to another expression, `a_exp`, which we write `f_exp a_exp`, which denotes the application of the function (denoted by) `f_exp` to the argument (denoted by) `a_exp`.

There are no constraints on the possibilities of passing functions as arguments to other functions, or to returning functions as the results of other (higher-order) functions. As a consequence, there is perfect *homogeneity* between programs and data.

From the semantic viewpoint, a program consists of a series of value definitions, each of which inserts a new association into the environment and can require the evaluation of arbitrarily complex expressions. The presence of higher-order functions and the possibility of defining recursive functions makes this definition mechanism flexible and powerful.

To a first approximation, the semantics of computation (*evaluation*) refers to no linguistic aspects other than the ones introduced so far. It can be defined using simple symbolic rewriting of strings (*reduction*), which repeatedly uses two main operations to simplify expressions until they reach a simple form which immediately denotes a value. The first of these operations is the simple search through the environment. When an identifier is determined as being bound in an environment, replace the identifier by its definition. For example, in Fig. 11.1, we used this operation for the first and the fifth reduction. In what follows this step will not be explicitly considered again. We will consider a name as a simple abbreviation for the value associated with it.

The second operation, which is more interesting, deals with a functional expression applied to an argument and uses a version of the copy rule (which in this context is called the β-rule).

Definition 11.1

Redex A *redex* (which stands for a *red*ucible *ex*pression) is an application of the form `((fn x => body) arg)`.

Reductum The *reductum* of a redex `((fn x => body) arg)` is the expression which is obtained by replacing in `body` each (free) occurrence of the formal parameter, `x`, by a copy of `arg` (avoiding variable capture[2], see p. 175).

β-rule An expression, `exp`, in which a redex appears as a subexpression is reduced (or rewrites, simplifies) to `exp1` (notation: `exp` $\rightarrow$ `exp1`), where `exp1` is obtained from `exp` by replacing the redex by its reductum.

From an implementation viewpoint, finally, every abstract machine for functional languages adopts extensive garbage-collection mechanisms because, in the case in which functions are returned as values by other functions, we know that the local environment must be preserved for an unlimited period of time (see Sect. 7.2.2).

If these are the cardinal notions of a functional language, on a more attentive reading, these few lines raise more problems than they solve. We undertake a deeper analysis of these semantic concepts in the next section.

11.2 Evaluation

In the brief semantic description with which we concluded the last section, we did not provide details about two fundamental aspects:

- What is the termination condition for reduction (that is what does "a simple form which immediately denotes a value" mean?)
- What precise semantics must be given to the β-rule, not just concerning possible variable capture (which we already described in the context of parameter passing) but more importantly, the order to follow during rewriting should more than one redex be present in the same expression.

[2] In the context of functional programming, identifiers bound to values are often referred to as "variables". The reader is by now experienced enough to allow us to continue with this traditional use without being confused by the absence of modifiable variables in pure functional programming languages.

11.2.1 Values

A *value* is an expression which cannot be further rewritten. In a functional language, there are values of two kinds: values of primitive type and functions. There is little to say about values of simple types. If the language provides some primitive types (integers, booleans, characters, etc.) is clear that a set of primitive values is associated with each type thus defined and that these values do not admit evaluation (for example, constants of type integer, boolean, characters, etc.). In the example of the previous subsection, we terminated the evaluation of `fact` when we reached primitive values of integer type: `3`, `2`, `1`.

Functional values are more interesting. Let us consider the following definition:

```
val G = fn x => ((fn y => y+1) 2);
```

We have said that a definition entails the evaluation of the expression on the right of the equality and the binding of the value thus derived to the name on the left of the =. But, in this case, it is not immediately clear what the value to be associated with `G` is. Do we have:

```
fn x => 3
```

in which we have rewritten the body of `G` by evaluating the redex it contains, or do we have

```
fn x => ((fn y => y+1) 2)
```

in which there has been no evaluation in the body of `G`? The first case appears to be the one that more respects the informal semantics that we gave at the end of the last subsection. The second, on the other hand, is closer to the usual definition of function in a conventional language, in which the body of a function is only evaluated when it is called.

Although at first sight, this can appear strange, it is the second of the above that is adopted for all functional languages in common use. Evaluation does not occur "under" an abstraction. Every expression of the form

```
fn x => exp
```

represents a value, so redexes possibly contained in `exp` are *never* rewritten until the expression is applied to some argument.

11.2.2 Capture-Free Substitution

To implement capture-free substitution, we saw in Chap. 7 that closures can be used. This is, in effect, the mechanism also used by abstract machines for functional

Fig. 11.2 An expression with more than one redex

```
fun K x y = x;
fun r z = r(r(z));
fun D u = if u=0 then 1 else u;
fun succ v = v+1;

val v = K (D (succ 0)) (r 2);
```

languages (see Sect. 11.4). For the elementary description we are giving at present, though, a syntactic convention suffices. In every expression, there are never two formal parameters with the same name and the names of the possible variables that are not formal parameters are all distinct from those of the formal parameters.

Using this convention, there will never be variable capture in the simple examples that we consider.[3]

11.2.3 Evaluation Strategies

In Chap. 6, while discussing expressions, we saw how every language must fix a specific strategy (that is a fixed order) for the evaluation of expressions. The presence of higher-order functions in functional languages makes this question even more fundamental. To clarify the problem, consider the definitions in Fig. 11.2. Which value is associated with v and how is it determined?

The β-rule on its own is not of much use because, in the right-hand part of the definition of v, there are 4 redexes (after the expansion of the names with the values associated with them by the definitions):

```
K (D (succ 0))
D (succ 0)
succ 0
r 2
```

Which of them is reduced first? Every commonly used language uses a leftmost strategy which reduces the redexes starting with the one at the leftmost end. But even having stipulated this, it is not clear which is the leftmost redex of

```
K (D (succ 0))
D (succ 0)
succ 0
```

because these three redexes are superimposed on each other. Having fixed a leftmost evaluation order, we then find that we have three different strategies.

[3]In the more general case, if we do not want to bring into play the concept of closure, for the correct description of computation using rewriting, it is necessary precisely to define the concepts of bound and free variable and substitution, all concepts which we treat formally in Sect. 11.6.

Evaluation by value In evaluation by value (which is also called applicative-order evaluation or *eager* evaluation, or *innermost* evaluation), a redex is evaluated only if the expression which constitutes its argument part is already a value.

More precisely, leftmost evaluation in applicative order works as follows.

1. Scan the expression to be evaluated from the left, choosing the first application encountered. Let it be `(f_exp a_exp)`.
2. First evaluate (recursively applying this method) `f_exp` until it has been reduced to a value (of functional type) of the form `(fn x => ...)`.
3. Then evaluate the argument part, `a_exp`, of the application, so that it is reduced to a value, `val`.
4. Finally, reduce the redex `((fn x => ...)val)` and goto to (1).

Considering Fig. 11.2, case (1) first chooses the application `K (D (succ 0))`. Now some elementary applications of (1), (2) and (3) will serve to show that `K`, `D` and `succ` are already values (recall that we are implying that a name is an abbreviation for the expression associated with it). The first redex to be reduced is then `succ 0` (that is, `((fn v => v + 1) 0))` which will be completely evaluated and yields `1`. Then the redex `(D 1)` (that is `((fn u => if u =0 then 1 else u) 1))` is reduced to give the value `1`. Then `(K 1)` is evaluated to produce the value `(fun y ==> 1)`. At this point in the evaluation, the expression has become

```
(fun y => 1) (r 2)
```

Since the functional part of this application is already a value, the strategy prescribes that the argument `(r 2)` is evaluated. The evaluation leads to rewriting to `r (r 2)`, then to `r (r (r 2))` and so on in a divergent computation.

No value is therefore associated with `v` because the computation diverges.

Evaluation by name In the evaluation by name strategy (which is also called *normal order* or *outermost*), a redex is evaluated before its argument part.

More precisely, leftmost evaluation in normal order proceeds as follows.

1. Scan the expression to be evaluated from the left, choosing the first application encountered. Let it be `(f_exp a_exp)`.
2. First evaluate `f_exp` (recursively applying this method) until it has been reduced to a value (of functional type) of the form `(fn x => ...)`.
3. Reduce the redex `((fn x => ..)a_exp)` using the β-rule and goto (1).

Considering Fig. 11.2, the first redex to be reduced is therefore:

```
K (D (succ 0))
```

It is rewritten to:

```
fn y => D (succ 0)
```

which is a functional value. The expression now is of the form:

```
(fn y => D (succ 0)) (r 2)
```

for which the strategy prescribes reducing the outermost redex, so we obtain:

```
D (succ 0)
```

Now reducing this expression, we obtain:

```
if (succ 0)=0 then 1 else (succ 0)
```

and then:

```
if 1=0 then 1 else (succ 0)
```

and finally

```
succ 0
```

from which we obtain the final value, `1`, which is then associated with `v`.

Lazy evaluation In evaluation by name, a single redex might have to be evaluated more than once because of some duplication that has occurred during rewriting. In the example that we are discussing, the redex `(succ 0)` is duplicated because of the function `D` and is reduced twice[4] in the conditional expression that forms the body of `D`. This is the price that must be paid for postponing the evaluation of an argument until after the application of a function (and it is really this that allows evaluation by name to obtain a value where evaluation by value would diverge). But it is very expensive in terms of efficiency (when the duplicated redex requires a significant amount of computation).

To obviate this problem and maintain the advantages of evaluation by name, the lazy strategy proceeds like that by name but the first time that a "copy" of a redex is encountered, its value is saved and will be used should any other copies of the same redex be encountered.

The by-name and lazy strategies are examples of the *call by need* strategies in which a redex is reduced only if it is required by the computation.

11.2.4 Comparison of the Strategies

In the last subsection, the by-name strategy produces a value when the by-value strategy diverges. It is sensible to ask the question as to whether it can be the case that the two strategies produce *distinct* values for the same expression.

[4]The purist does not speak of copies of redexes but will rather say that the redex `(succ 0)` has given way to two *residuals*, each of which has been reduced independently of the other.

An answer to this question is given by the following theorem which expresses one of the most important characteristics of the *pure* functional paradigm. We say that an expression in the language is *closed* if all of its variables are bound by some `fn`. Let us recall, then, from Sect. 11.2.1, that by *primitive value*, we mean a value of a primitive type (integer, boolean, character, etc.), excluding, therefore, functional values.

Theorem 11.2 *Let* `exp` *be a closed expression. If* `exp` *reduces to a primitive value,* `val`, *using any of the three strategies of Sect.* 11.2.3, *then* `exp` *reduces to* `val` *following the by-name strategy. If* `exp` *diverges using the by-name strategy, then it also diverges under the other two strategies.*

Let us note that the theorem excludes the case in which a (closed) expression can yield a primitive value, `val`, in one strategy and yield *another* primitive value, `val2`, using another strategy.[5] Two strategies can therefore differ only by the fact that one determines a value while the other diverges, as we saw in our example. We cannot go into the details of how the theorem can be proved but it is important to stress that the following basic property is fundamental to its validity:

> Once any strategy has been fixed in a given environment, the evaluation of all occurrences of a single expression always yields the same value

This property, which is immediately falsified when there are side effects, is taken by many authors as the *criterion* for a pure functional language: a language is purely functional if it satisfies this condition. This is a very important property which makes it possible to *reason* about a functional program and to which we will return in Sect. 11.5. Let us again note, before going on, that it is just by virtue of this property that lazy evaluation is correct. The value obtained by the evaluation of an expression does not change when another copy of the same expression is encountered (clearly, in the meanwhile, the environment has not been changed).

In the light of the preceding theorem, it can be asked what interest there is in the by-value strategy, given that by-name is the most general of all possible strategies. Moreover, reasons of efficiency would seem to suggest that we only adopt call-by-need strategies, given that it is not clear why useless redexes should be reduced, with the consequent waste of computing time. The point is that the efficiency case is not so simple. To implement a call-by-need strategy is, in general, more expensive that a simple call-by-value strategy. The by-value strategy has efficient implementations

[5]The theorem is no longer valid if the hypothesis that the value is *primitive* is removed. Having assumed that evaluation is not performed under abstraction, in fact, the two strategies can yield (functional) values that are distinct. By way of an example, define `I = fn x => x` and `P = fn x => (fn y => y x)`. The term `P (I I)` reduces to `fn y => y (I I)` under the by-name strategy, while, using the by-value strategy, it reduces to `fn y => y I`. This difference is not very important in programming languages (where we mainly want to compute values of primitive type). In Sect. 11.6, we will define an abstract calculus in which reduction also occurs in the presence of abstractions. This calculus satisfies the *confluence* property (stronger and more general than Theorem 11.2), which we will discuss on page 361.

on conventional architectures, even if, sometimes, it performs unnecessary work (every time that an argument that is not required in the body of the function is evaluated, as happens with the second argument of the function `K` in Fig. 11.2). This last case, moreover, can be treated in an efficient manner with abstract evaluation strategies that attempt to identify useless arguments.

Among the functional languages that we have cited at the beginning of this chapter, LISP, Scheme and ML use a by-value strategy (also because they include major imperative aspects), while Miranda and Haskell (which are pure functional languages) use lazy evaluation.

11.3 Programming in a Functional Language

The mechanisms we described in the last section are sufficient to express programs for every computable function (they constitute, indeed, a Turing-complete language). This is the nucleus of every functional language, which, however, is too austere for us to be able to use as a real programming language. Every functional language, therefore, embed this nucleus in a wider context which provides mechanisms of different kinds, each aiming to make programming simpler and more expressive.

11.3.1 Local Environment

The mechanism of global definitions in the environment that we have used this far has too little structure for a modern language. As in conventional languages, it is appropriate to provide explicit mechanisms to introduce definitions with limited scope, as for example:

```
let x = exp in exp1 end
```

This introduces the binding of `x` to the value of `exp` in a scope which includes only `exp1`.

To tell the truth, the presence of functional expressions already introduces nested scopes and associated environments which are composed of the formal parameters of the function bound to the actual parameters. From the point of view of evaluation, we can indeed consider the construct

```
let x = exp in exp1 end
```

as syntactic sugar for

```
(fn x => exp1) exp.
```

The use of local scopes is too important, however, and justifies the introduction of special syntax for them.

11.3.2 Interactiveness

Every functional language has an interactive environment. The language is used by entering expressions which the abstract machine evaluate and whose value it returns. Definitions are particular expressions which modify the global environment (and may return a value).

This model immediately suggests interpreter based implementations, even if there are efficient implementations that use compilation to generate compiled code the first time a definition is stored in the environment.

11.3.3 Types

As in conventional languages, the type system is an aspect of primary importance in functional languages, as well. Every functional language we have cited provides the usual primitive types (integers, booleans, characters) with operations on their values. With the exception of Scheme, which is a language with dynamic type checking, all the others have elaborate static type systems. These type systems allows the definition of new types such as pairs, lists, "records" (that is tuples of labelled values). For example, in ML, we can define a function `add_p` which takes as arguments a pair of integers and returns their sum:

```
fun add_p (n1,n2) = n1+n2;
```

Note the fundamental difference between `add_p` and `add` which we defined on page 336. `add_p` requires a pair of integers and it would make no sense to provide just one value for `n1` and none for `n2`. Instead, `add` it is a function which inputs *one* number and returns a function that takes another number and returns the sum of both.[6]

An important part of the type system in functional languages is that dedicated to the types of functions, given that functions are denotable and expressible values (and often also storable, if the language includes imperative aspects). In typed functional languages, this also means that some functional expressions are illegal because they cannot be typed. For example, the following higher-order function is illegal in a typed language:

```
fun F f n = if n=0 then f(1) else f("pippo");
```

The reason for this is that the formal parameter, `f`, must simultaneously have types `int -> 'a` and `string -> 'a` (where `'a` denotes a type variable, that is a generic type that is not yet instantiated).

[6]The passage from a function requiring, as argument, a pair (or, more generally, a tuple) to a unary higher-order function, is called the *curryfication* of a function. The operation is named after Haskell Curry, one of the founding fathers of the theory of the λ-calculus.

Another illegal expression (because it violates typing) is self application:

```
fun Delta x = x x;
```

The expression, `(x x)`, is illegal because there is no way to assign a unique and consistent type to `x`. Given that it occurs on the left as an application, it must have a type of the form `'a -> 'b`. Since, then, it also appears as the argument to a function (on the right of the application), it must have the type that the function requires: therefore, `x` must be of the type `'a`. Putting the two constraints together, we have it that `x` must, at the same time, be of type `'a` and of type `'a -> 'b` and there is no way to "unify" these two expressions.

In languages without a strong typing system, such as Scheme, the function `Delta` is, instead, legal. In Scheme, we can also apply `Delta` to itself. The expression (obviously written in Scheme syntax)

```
(Delta Delta)
```

constitutes a simple example of a divergent program. The lack of a static type check makes it possible in Scheme to write expressions such as:

```
(4 3)
```

Here, the aim is to apply the integer `4` to the integer `3`. Since the left-hand part of this application is not a function, the abstract machine will generate an error at execution time.

In the case of ML, the most interesting aspect of its type system is its support for polymorphism (see Sect. 8.8 and, in particular, the box on page 243).

11.3.4 Pattern Matching

One of the most annoying aspects of recursive functional programming is handling terminal cases by means of explicit "if" expressions. Let us take, for example, the (inefficient) function which returns the *n*th term of the Fibonacci series:

```
fun Fibo n = if n=0 then 1
             else if n=1 then 1
                  else Fibo(n-1)+Fibo(n-2);
```

The mechanism of *pattern matching* present in some languages such as ML and Haskell, allows us to give a definition (which is equivalent to the above) as follows:

```
fun Fibo 0 = 1
 |  Fibo 1 = 1
 |  Fibo n = Fibo(n-1)+Fibo(n-2);
```

The "|" character is read "or". Each branch of the definition corresponds to a different case of the function. The most interesting part of this definition is the formal parameters. They are no longer constrained to be identifiers but can be *patterns*, that is, expressions formed from variables (the last is an example), constants (in the other cases) and other constructs that depend on the language's type system (we will see an example using lists below). A pattern is a kind of schema, a model against which to match the actual parameter. When the function is applied to an actual parameter, it is compared with the patterns (using the order in which they appear in the program) and the body corresponding to the first pattern which matches with the actual parameter is chosen.

The pattern-matching mechanism is particularly flexible when structured types are used, for example lists. In ML syntax, a list is denoted by square brackets with the elements separated by commas. For example

```
["one", "two", "three"]
```

is a list of three strings. The operator, `::` denotes "cons" (that is the operator that adds an element to the head of a list):

```
"zero"::["one", "two", "three"]
```

is an expression whose value is the list:

```
["zero", "one", "two", "three"].
```

Finally, `nil` is the empty list. Thus

```
"four"::nil
```

has value `["four"]`. Using pattern matching, we can define a function which computes the length of a generic list such as:

```
fun length nil = 0
 |  length e::rest = 1 + length(rest);
```

Note that the names used in the pattern are used as formal parameters to indicate parts of the actual parameter. It should be clear the advantage in terms of conciseness and clarity with respect to the usual definition:

```
fun length list = if list = nil then 0
                  else 1 + length(tl list);
```

This definition, moreover, requires the introduction of a *selection* function (which we have written `tl`) to obtain the list that is obtained by removing the head element. The selection operation is, on the other hand, implicit in pattern matching.

An important constraint is that a variable cannot appear twice in the same pattern. For example, consider the following "definition" of a function which, applied to a list, returns `true` if and only if the first two elements of the list are equal:

```
fun Eq nil = false
 |  Eq [e] = false
 |  Eq x::x::rest = true
 |  Eq x::y::rest = false;
```

This definition is syntactically illegal because the third pattern contains the variable `x` twice, where it is used to test the equality of the first two elements. The pattern-matching mechanism is one way to check that the *form* of an actual parameter agrees with a pattern, not that the values it contains bear certain relationships. Otherwise stated, pattern matching is *not* unification, a much more general mechanism (and more complex to implement) and which we will discuss in the context of the logic programming paradigm (Chap. 12).[7]

11.3.5 Infinite Objects

In the presence of by-name or lazy strategies, it is possible to define and manipulate *streams*, that is, data structures that are (potentially) infinite. In this section, we will give a small example of how this can be done. We cannot give the example in ML because it uses a by-value strategy. The example is possible, though, in Haskell. So as not to load the reader with different syntax, we will write the terms with the same concrete syntax as ML but with the stipulation that evaluation must be understood as lazy.

First, we have to clarify the concept of value for data structures such as lists. In a language that uses eager evaluation, a value of type `T list` is a list whose elements are *values* of type `T`. In a lazy language, this is not a good notion of value because it might require the evaluation of useless redexes, contrary to the call-by-need philosophy. To see the reason, define the functions:

```
fun hd x::rest = x;
fun tl x::rest = rest;
```

The functions return, respectively, the first element (that is, the head) and the rest (that is, the tail) of a non-empty list (in the case of the empty list, the abstract machine will generate an error when doing pattern matching).

Let us now consider the expression

```
hd [2, ((fn n=>n+1) 2)].
```

[7]The implementation of pattern matching is nothing other than the "obvious" translation into a sequence of `if`s. In the case of `Eq`, the third pattern would require checking the equality of two elements *of arbitrary type*. If we have values for which the language does not define equality (as happens, for example, for functional values), it would not be possible to implement an equality test.

To calculate its value (that is, 2), it is not necessary to reduce the redex that comprises the second element of the list because it will never be used in the body of the hd function. For these reasons, in a by-need context, a value of type list is any expression of the form:

```
exp1 :: exp2
```

where exp1 and exp2 can also contain redexes.

It is then possible to define a list by recursion, as for example:

```
val infinity2 = 2 :: infinity2;
```

The expression infinity2 corresponds to a potentially infinite list whose elements are all 2 (using eager evaluation, such a list would be divergent).

This value is perfectly manipulable under lazy evaluation. For example,

```
hd infinity2
```

is an expression whose evaluation terminates with the value 2. Another expression with value 2 is

```
hd (tl (tl (tl infinity2))).
```

As a last example of a stream, the following function constructs an infinite list of natural numbers starting with its argument, n:

```
fun numbersFrom n = n :: numbersFrom(n+1);
```

We can define a higher-order function that applies its functional argument to all the elements of a list, as in:

```
fun map f nil = nil
 |  map f e::rest = f(e)::rest;
```

We can now produce the infinite list of all the squares starting from n*n:

```
fun squaresFrom n = map (fn y => y*y) (numbersFrom n);
```

11.3.6 Imperative Aspects

Many functional languages also include imperative mechanisms which introduce a notion of state that can be modified by side effects.

In ML, there are real modifiable variables (called "reference cells") with their own types. For every type T in the language (including functional types, therefore), the type T ref is defined and its values are modifiable variables that can contain values of type T. A modifiable variable of type T, initialised to the value v (of

Side effects in LISP

LISP, the first functional language, is based on a data structure called a *dotted pair* (or cons cell, see the box on page 121). Dotted pairs are composed of two parts: the *car* (for *contents of the address register*) and *cdr* (for *contents of the decrement register*). An expression of the form:

```
(cons a b)
```

allocates a dotted pair and initialises it so that its `car` points to the value of `a` and its `cdr` points to the value of `b`. The two components of a dotted pair can be selected using the functions `car` and `cdr`. So,

```
(cdr (cons a b))
```

has the value `b`. These are characteristics that belong to the pure part of LISP.

LISP has, in addition, imperative mechanisms. For example, there are functions such as `(rplaca x a)` and `(rplacd x a)` which, respectively, *assign to the car* and *assign to the cdr* of the dotted pair `x` the value of `a`:

```
(cdr (rplacd (cons 'a 'b) 'c))
```

This has the value `'c`. Other side-effecting functions are `set` and `setq`.

---

type `T`) is created by:

```
ref v
```

The usual constructs that associate names and values can also be used on values of type `T ref`. For example:

```
val I = ref 4;
```

This creates a reference cell of type `int ref`, which is initialised to `4` and has the name `I`. This is what, in imperative languages, we call "the modifiable variable (with name) `I`." Obviously, `I` is the name of the reference cell and not of the value it contains (it is an l-value). To obtain its r-value, it is necessary to dereference it explicitly using the `!` operator:

```
val n = !I + 1;
```

The expression `!I` is of type `int`. In general, if `V` is an expression of type `T ref`, `!V` is of type `T`. The line above associates the name `n` with the value `5`. At risk of being pedantic, let us observe that `n` is not a modifiable variable, it is an ordinary name bound to a value in the environment.

Reference cells can be modified using assignments:

```
I := !I + 1;
```

Note the difference with the definition of the value n given above. Here, we are modifying (using a side effect) the r-value of a reference cell that already exists. The type of an imperative construct of this kind is `unit` (recall Sect. 8.3.7).

The presence of modifiable variables clearly introduces the possibility of aliasing. Let us indeed consider the following definitions:

```
val I = ref 4;
val J = I;
```

The second line associates the value of the name I (that is the reference cell—the l-value) with the name J. We here have a classic situation of aliasing: I and J are different names for the same l-value. After execution of the fragment

```
I := 5;
val z = !J;
```

the name z is associated with the value 5.

The language definition does not specify how to implement an assignment between modifiable variables. From the examples given so far using integers, one could imagine a traditional implementation with the r-value being copied from the source to the destination of the assignment. However, values with quite different memory sizes can be stored in reference cells. Let us consider, for example, modifiable variables that contain lists and strings:

```
val S = ref "pear";
val L = ref ["one", "two", "three"];
```

The name S is of type `string ref`, while L is of type `(string list) ref`. The two modifiable variables associated with the names S and L can contain, respectively, *any* string and *any* list. For example, we can perform two assignments:

```
S := "this␣is␣a␣string␣much␣longer␣than␣previously";
L := "zero" :: "four" :: "five" :: !L;
```

The new values of L and S require more memory than their previous values did. It is not possible to implement these assignments using a simple (and traditional) value copy. The abstract machine will, in this case, copy references (or pointers) to these values. All of this is, though, completely invisible to the language user. The implementation handles the two space requirements by allocating the necessary memory for each case and then modifies the references.

As well as modifiable variables, ML also provides imperative control constructs, such as sequential composition (`;`) and loops.

It is appropriate to note that, in the presence of these imperative features, both Theorem 11.2 and the property that we cited immediately after it *do not apply*. This is particularly relevant in the case of a lazy evaluation strategy which would become incorrect in the presence of side effects. It is for this reason that languages such as Miranda and Haskell, which use lazy evaluation, do not admit side effects of any kind (they are purely functional languages).

11.4 Implementation: The SECD Machine

The technology for implementing functional programming languages is today sophisticated and cannot be adequately described in this book. The techniques for passing functions as parameters we described in Chap. 7 constitute the heart of these implementations. In order to give more details on how to handle higher-order functions, we will limit ourselves to summarising the SECD machine, a prototype proposed in 1964 by Peter Landin for evaluation *by value*. Many other abstract machines have developed out of the SECD machine.

We will describe the SECD machine in an abstract manner, using a very simple, purely functional language that uses only abstraction and application. The primitive elements of the language are the names of constants and variables and some primitive functions (for example, we could have constants `0`, `1`, `true`, etc. and primitive functions `succ` and `pred`). The important part of the machine is not concerned with these aspects of the language but with the presence of higher-order functions. Let us assume, for simplicity, that we are only dealing with unary functions. Let us assume that *Var* is a non-terminal from which can be derived a denumerable number of names (of variables) and that *Const* and *Fun* are, respectively, the non-terminals from which are derived the appropriate constants and primitive functions. We can state the grammar of our example language as:

$$exp ::= Const \mid Var \mid Fun \mid (exp \quad exp) \mid (\texttt{fn} \quad Var \quad \texttt{=>} \quad exp).$$

The SECD machine has four main components:

- A **S**tack to store the partial results which will be used during the computation.
- An **E**nvironment, that is a list of associations between names and values. We have already noted that "there is no evaluation under an abstraction". The value of an abstraction is a *closure*, that is a pair formed of the abstraction and the environment in which the free variables in its body are to be evaluated. To simplify the concept, let us assume that a closure is a *triple* composed of an environment, an expression (which will always be derived from the body of an abstraction) and from a variable (representing the variable that was bound in the abstraction). We will therefore write a closure as $cl(E, exp, x)$, which will represent the value in the environment, E, of the abstraction ($\mathsf{fn}\, x => exp$).
- A **C**ontrol, represented by a stack of expressions to be evaluated. Among the expressions is also the special operator, @, which is read "app" and which indicates

that an application should be executed (because the two expressions that comprise an application have already been evaluated and are stored in the next two positions on the stack).

- A storage area called the **D**ump, that is a stack in which previous states of the machine have been saved when the computation was suspended to evaluate internal redexes.

We write $[a_1, \ldots, a_n]$ to denote the stack composed of the elements $a_1, \ldots, a_n$ with a_1 at its top. [] is the empty stack. If P is a stack, $top(P)$ denotes the topmost element, while $tl(P)$ denotes P with the top element removed.

A state of the machine is composed of a quadruple, (S, E, C, D). A dump is always composed of a structure of the form:

$$(S_1, E_1, C_1, (S_2, E_2, C_2, (S_3, E_3, C_3, [\,]))).$$

This state of the dump has been obtained by first saving the state

$$(S_3, E_3, C_3, [\,])$$

then the state

$$(S_2, E_2, C_2, (S_3, E_3, C_3, [\,]))$$

which includes the previous dump, and so on.

To evaluate an expression, *exp*, the machine starts its operation in the state $([\,], [\,], [exp], [\,])$ and terminates when both the control, C, and the dump, D, are empty.

The operation of the SECD is described by a transition function which allows it to pass from one state to a succeeding one using the following rules. The machine selects the rule according to the expression on the control list (imagine that control is described using abstract syntax). Let us assume, therefore, that the machine finds itself in the generic state, (S, E, C, D), then:

1. A constant, c (e.g., 1, 0, *true*), is on top of the control list, that is, there is an expression that does not need to be evaluated. In this case, the new state of the machine is $(c :: S, E, tl(C), D)$ in which the (immediate) value of the expression is pushed onto the stack and the expression just evaluated is removed from the stack.
2. There is a variable, x, on the top of the control list. In this case, the value is the one which the environment associates with x. Therefore, the new state of the machine is $(E(x) :: S, E, tl(C), D)$.
3. On the top of the control list, there is an application of the form $(exp_f\ exp_a)$. In this case, since the SECD implements eager evaluation, it is necessary to follow the evaluation procedure we described in Sect. 11.2.3. The new state is therefore $(S, E, exp_f :: exp_a :: @ :: tl(C), D)$ which expresses the fact that the next expression to be evaluated is the functional part of the application (which will have as its value a closure, as we will see), followed by the argument part of the application, followed, finally, be the special expression, @, which will force the real application of the function to its argument.

4. On the top of control list, there is an abstraction, (fn x => exp). We know that we are already in the presence of a value which must however be "closed" with the current environment. The new state of the machine is therefore $(cl(E, exp, x) :: S, E, tl(C), D)$.
5. The special value @ is on the top of the control list. We then know, by construction, that on the stack there are two values that represent, respectively, an argument (on the top of the stack) and a function to apply (the second element). There are two subcases which depend on the function to be applied:

   - The function is a *primitive function*, f, such as *succ*. The function must be directly applied to its argument and the new state of the machine is $(f(top(S)) :: tl(tl(S)), E, tl(C), D)$.
   - The function on the top is a closure of the form $cl(E_1, exp, x)$. This is one of the central points of the SECD machine; it brings the dump into play. It is necessary to use the copy rule. The computation passes to *exp* in the environment of the closure, E_1, (and not in the current environment, E) modified by the binding of the formal parameter. But the computation performed up to now must not be forgotten. It is frozen on the dump. The new state of the machine is therefore $([], E_1[x \leftarrow top(S)], [exp], (tl(tl(S)), E, tl(C), D))$.

6. The control list is empty. This can happen when the computation has terminated (in which case, the dump is also empty). It can also happen when a computation started by the previous rule (the one for applying a closure to a value) is finished. In this second case (that is, non-empty dump of the form (S_1, E_1, C_1, D_1)), it is necessary to resume the computation saved on the dump, restoring, at the same time, the value just computed (which is on the top of the stack). The new state of the machine is $(top(S) :: S_1, E_1, C_1, D_1)$.

Figure 11.3 shows how the SECD machine computes the value of the expression $(F\ 3)$, where:

$$F = \texttt{fn}\, x \texttt{ => } (sqrt\ ((\texttt{fn}\ y \texttt{ => } (succ\ x))\ x)).$$

The computation is shown as a table in which every row represents a transition of the machine. The rule being used for the transition is indicated in the rightmost column.

11.5 The Functional Paradigm: An Assessment

Having reached the end of our presentation on the functional paradigm, it is appropriate to ask ourselves wherein resides the interest in it and functional languages. Such a question, in reality, must be asked on at least two different planes:

- That of practical programming.
- That of programming language design.

We will try to give an answer on these two levels but it is appropriate to divert our attentions for an instant to clarify the context in which we can talk correctly about "practical programming."

S	**E**	**C**	**D**	**r**
$[]$	$[]$	$[(F\ 3)]$	$[]$	
$[]$	$[]$	$[F, 3, @]$	$[]$	3
$[ch_1]$	$[]$	$[3, @]$	$[]$	4
	where $ch_1 = cl([], (sqrt\,((\texttt{fn}\ \ y\ \ \texttt{=>}\ \ (succ\,x))\,x)), x)$			
$[3, ch_1]$	$[]$	$[@]$	$[]$	1
$[]$	$[x \leftarrow 3]$	$[(sqrt\,((\texttt{fn}\ \ y\ \ \texttt{=>}\ \ (succ\,x))\,x))]$	d_1	5
	where $d_1 = ([], [], [], [])4$			
$[]$	$[x \leftarrow 3]$	$[sqrt, ((\texttt{fn}\ \ y\ \ \texttt{=>}\ \ (succ\,x))\,x), @]$	d_1	3
$[sqrt]$	$[x \leftarrow 3]$	$[((\texttt{fn}\ \ y\ \ \texttt{=>}\ \ (succ\,x))\,x), @]$	d_1	1
$[sqrt]$	$[x \leftarrow 3]$	$[(\texttt{fn}\ \ y\ \ \texttt{=>}\ \ (succ\,x)), x, @, @]$	d_1	1
$[ch_2, sqrt]$	$[x \leftarrow 3]$	$[x, @, @]$	d_1	4
	where $ch_2 = cl([x \leftarrow 3], (succ\,x), y)$			
$[3, ch_2, sqrt]$	$[x \leftarrow 3]$	$[@, @]$	d_1	2
$[]$	$[x \leftarrow 3, y \leftarrow 3]$	$[(succ\,x)]$	d_2	5
	where $d_2 = ([sqrt], [x \leftarrow 3], [@], d_1)$			
$[]$	$[x \leftarrow 3, y \leftarrow 3]$	$[succ, x, @]$	d_2	3
$[succ]$	$[x \leftarrow 3, y \leftarrow 3]$	$[x, @]$	d_2	1
$[3, succ]$	$[x \leftarrow 3, y \leftarrow 3]$	$[@]$	d_2	2
$[4]$	$[x \leftarrow 3, y \leftarrow 3]$	$[]$	d_2	5
$[4, sqrt]$	$[x \leftarrow 3]$	$[@]$	d_1	6
$[2]$	$[x \leftarrow 3]$	$[]$	d_1	5
$[2]$	$[]$	$[]$	$[]$	6

Fig. 11.3 A computation using the SECD machine

Program Correctness The computing beginner often holds that the design and writing of an efficient program are the most important operations on which their occupation is based. Software Engineering, however, has largely shown, both in theory, as in ample experimental studies, that the most critical factors in a software project are its correctness, readability, maintainability and its dependability. In economic terms, these factors account for more than fifty percent of the total cost; in social terms, software maintenance (which depends in a critical way on its readability) can involve hundreds of different people over a period of tens of years. In ethical terms, the life, or health, of many hundreds of people can depend on the reliability and correctness of a software system.

In particular, the time is still far away when we will be able to produce software with correctness guarantees comparable to those with which a civil engineer releases their own products (bridges, columns, structures). The civil engineer, indeed, has at his disposal a whole corpus of applied mathematics with which to "calculate" the structures. If a bridge must take a certain load, be subjected to certain winds, can be exposed to seismic events of given magnitude, etc., it must have given dimensions, must be built using materials with certain characteristics, etc. These characteristics are not determined by the designer's taste but are calculated using appropriate mathematical techniques. The information-system designer is far from having at their disposition a mathematics even remotely on a par with those available to the building designer.

One of the reasons for the delay in being able to produce such a level of correctness guarantee is that to reason about programs with side effects is particularly difficult and expensive in terms of computation time. On the contrary, there are standard techniques, based on appropriate variants of induction, that allow reasoning about side-effect free programs.

Here, then, is a first reason for the study of pure functional languages. If reliability, readability, correctness are more important than efficiency, there is no doubt that functional programming generates more readable software, whose correctness is easier to establish and, therefore, is more reliable.

Program schemata Higher-order functions, which are commonly used in the functional paradigm, have important pragmatic advantages. A typical way of using higher-order is that of exploiting it to define *general programming schemata* from which specific programs can be obtained by instantiation. Let us consider, for example, a simple program that adds the elements of a list of integers. We can write this in ML as:

```
fun addl nil = 0
 |  addl n::rest = n + addl(rest);
```

If we now want the product of a list, we can write:

```
fun prodl nil = 1
 |  prodl n::rest = n * prodl(rest);
```

It is clear that these two programs are instances of a single program schema, which is generally called *fold*:

```
fun fold f i nil = i
 |  fold f i n::rest = f(n, fold f i rest);
```

Here, `f` is the binary operation to be applied to successive elements of the list ($+$ or $*$), `i` is the value to use in the terminal case (the neutral element of the operation) and the third parameter is the list over which to perform the iteration.

If we ignore the fact that $+$ and $*$ are infix operators rather than prefix, we can immediately see that we can define our two functions using *fold*:

```
val suml = fold + 0;
val prodl  = fold * 1;
```

Here, the fact that the function is higher-order is used both to pass to *fold* the function to iterate over the list, and to make `fold + 0` returns a function which requires an argument of list type.

The extensive use of program schemata increases the modularity of code and allows correctness proofs to be factored.

Assessment The arguments we have adduced in favour of functional languages do not just come from academic environments. Probably the most passionate defence of the functional paradigm is that in John Backus' 1977 Turing Award acceptance lecture (the award was bestowed for his work on FORTRAN).

Anticipating the times, in some sense, Backus put the concepts of correctness and readability at the centre of the process of software production, relegating efficiency to second place. He also identified *pure* functional programming as the tool with which to proceed in the direction he advocated. Today, we have machines that are much more efficient than those of 1977 but we have not made the necessary progress in the area of software correctness techniques.

No-one has however really experimented with large-scale purely functional programs in such a way that a real comparison with imperative programs can be made. Significant experiments have been conducted at IBM using the FP language defined by Backus, and in academic research centres using the language Haskell. All the other functional languages in use have conspicuous imperative aspects which, although used in ways and places very different from those of ordinary imperative languages, dissipate the advantages of functional languages in terms of correctness proofs (but not in terms of readability and schema reuse). In conclusion, we must say that, from the practical point of programming (understood in the light of the entire software production process), the superiority of one paradigm over the other has yet to be shown.

It is on the second level, that of programming language design, that the studies and experience with functional languages has had a significant impact. Many individual concepts and experiments in functional programming have later migrated to other paradigms. Among these concepts, the best known to the reader is that of type system. The concepts of generics, polymorphism, type safety, all originated in functional languages (because is it simpler to study and to implement them in an environment without side-effects).

11.6 Fundamentals: The λ-calculus

At the start of the chapter, we saw that the functional paradigm is inspired by a foundation for computability theory, different to that based on Turing machines (but equivalent to that as far as expressive power is concerned). In this section, we present the important points of this formal system, which is called the λ-calculus.

The syntax of the λ-calculus is extremely austere and can be given in one line using the following grammar, where X is a non-terminal which represents a generic variable, M is the initial symbol, the dot and the two parentheses are terminal symbols:

$$M ::= X \mid (MM) \mid (\lambda X.M).$$

In addition, let us assume that we also have a denumerable set of terminal symbols for variables which, for ease of notation, we will in general write using the last lower-case letters of the alphabet: x, y, z, etc.

The reader will recognise in these clauses both application and abstraction (written using λ instead of `fn`) which we introduced informally in Sect. 11.1.3. We call λ-terms (or, simply, terms) the strings of terminals which are derived from this grammar.

Syntactic conventions, free and bound variables The "official" books on the λ-calculus introduce many additional conventions to simplify the notation. For example, application associates to the left, or:

$$M_1 M_2 \cdots M_n \quad \text{stand for } (\cdots(M_1 M_2)\cdots M_n).$$

The scope of a λ extends as far as possible to the right (that is, $\lambda x.xy$ stands for $(\lambda x.(xy))$ and not for $((\lambda x.x).y)$. We will seek to use these conventions as little as possible by using parentheses where necessary. We will also speak of *subterms*, with the obvious meaning (for example, (xy) is a subterm of $(\lambda x.(xy))$ but λx is not).

The abstraction operator *binds* the variable upon which is acts,[8] in the double sense that, semantically, consistent renaming of the bound variable does not modify the semantics of an expression and, syntactically, possible substitutions have no effect on bound variables. We formalise these aspects in the following definitions. First, we define, for an arbitrary expression, M, the set of its *free variables*, which we denote by $Fv(M)$, and of its bound variables $Bv(M)$:

$$\begin{array}{ll} Fv(x) = \{x\} & Bv(x) = \emptyset, \\ Fv(MN) = Fv(M) \cup Fv(N) & Bv(MN) = Bv(M) \cup Bv(N), \\ Fv(\lambda x.M) = Fv(M) - \{x\} & Bv(\lambda x.M) = Bv(M) \cup \{x\}. \end{array}$$

Substitution We can formally define the concept of substitution without variable capture which constitutes the core of the copy rule. We define, therefore, the notation, $M[N/x]$ which we read as the substitution of N for the free occurrences of x in M:

$$\begin{array}{ll} x[N/x] = N, & \\ y[N/x] = y & \text{when } x \neq y, \\ (M_1 M_2)[N/x] = (M_1[N/x] M_2[N/x]), & \\ (\lambda y.M)[N/x] = (\lambda y.M[N/x]) & \text{when } x \neq y \text{ e } y \notin Fv(N), \\ (\lambda y.M)[N/x] = (\lambda y.M) & \text{when } x = y. \end{array}$$

The definition is simple. Practically, to substitute N in place of x in M, we "push" the substitution down towards the leaves of the syntax tree, that is to the variables, and check if the leaf is labelled with x or with some other variable. But the definition is subtle, since it avoids capture of the variables in N by some λ. This is the reason why the last clause requires that $y \notin Fv(N)$. What happens if, on the other hand, y *is present in* N? We can rename the variable bound by λ. As we said above, indeed,

[8]Remember what was said in Sect. 7.1 about formal parameters to procedures.

it is not the name of the variable that counts when a λ is applied, but only the way in which it is used. In other words, we will consider as equivalent two terms that differ only by the names of their bound variables. For example, $\lambda x.x$ and $\lambda y.y$, or $\lambda x.\lambda y.x$ and $\lambda v.\lambda w.v$.

Alpha equivalence The intuitive idea that two expressions are equivalent when they differ only in the names of their free variables is formalised in the concept of α-equivalence:

$$\lambda x.M \equiv_\alpha \lambda y.M[y/x] \quad y \text{ fresh}$$

where by "y fresh", we mean that y is a new variable not present in M.

Two terms that differ by just the replacement of some subterm of one term by an α-equivalent subterm will be considered identical.

Computation: β reduction Computation proceeds by rewriting according to the prescriptions of *β-reduction*:

$$(\lambda x.M)N \to_\beta M[N/x].$$

More generally, we will say that M β-reduces to N (and we write $M \to N$) when N is the result of the application of one step of β-reduction to some subterm of M. A subterm of the form $(\lambda x.M)N$ is a *redex* whose *reductum* is $M[N/x]$. Note that the concept of β-reduction is not deterministic. Whenever there exists more than one redex in the same term, it is not prescribed which one will be chosen.

β-reduction is a relation that is not symmetric. In general, if $M \to N$, it is not the case that $N \to M'$. It is useful to introduce a symmetric relation which we call β-equivalence, defined as the reflexive and transitive closure of β-reduction and denoted by $=_\beta$. Intuitively, $M =_\beta N$ means that M and N are connected through a sequence of β-reductions (not necessarily all in the same direction).

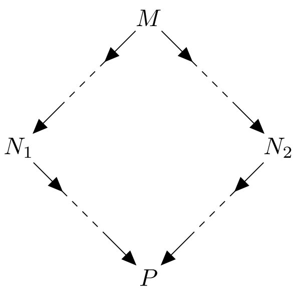
Normal forms When a λ-term that does not contain a redex, β-reduction terminates; such terms are called *normal forms*. For example, $\lambda x.\lambda y.x$ is a normal form, while the term $\lambda x.(\lambda y.y)x)$ is not a normal form because it contains the redex $(\lambda y.y)x$. We have, then:

$$\lambda x.(\lambda y.y)x \to \lambda x.x$$

and $\lambda x.x$ is now a normal form.

Some terms have reductions that terminate (in a normal form). There are however also terms which reduce an infinite number of times without ever producing a normal form, as for example:

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &\to (xx)[(\lambda x.xx)/x]\\ &= (\lambda x.xx)(\lambda x.xx)\\ &\to (\lambda x.xx)(\lambda x.xx)\\ &\to \ldots.\end{aligned}$$



$$\begin{tikzcd} & M \arrow[dl, dashed] \arrow[dr, dashed] & \\ N_1 \arrow[dr, dashed] & & N_2 \arrow[dl, dashed] \\ & P & \end{tikzcd}$$

Fig. 11.4 The confluence property for λ-terms

Confluence A fundamental property of the λ-calculus is that its inherent non-determinism has no dangerous effects. In any way we might choose the redex to reduce inside a term, the final result (the normal form) of a sequence of reductions is always the same. Stated in a more formal manner, we have the following property, called *confluence* and graphically represented in Fig. 11.4.

> If M reduces to N_1 in a number of reduction steps, and M also reduces to N_2 in a number of reduction steps, then there exists a term, P, such that both N_1 and N_2 both reduce to P in a number of steps.

An important consequence of this property is that if a term can be reduced to a normal form, this normal form is unique and independent of the path followed to reach it.

Fixpoint operators We have seen how the λ-calculus formalises many concepts that we have seen in this chapter.[9] We have not however introduced any mechanisms for the definition of recursive functions. The λ-calculus has no notion of external or global environment in which to assign names to terms, as happens when defining a recursive function, for which naming seems to be a fundamental concept.

The point (surprising at first sight) is that the concepts of abstraction and local environment (which is derived from the possibility of nested λ's) are *on their own sufficient* for the definition of recursion or, more precisely, fixed points.

Let us consider the following very simple recursive definition, which, for simplicity, we write in a λ-calculus extended with integers and conditional expressions:[10]

$$Z = \lambda n.\ \text{if}\ n = 0\ \text{then}\ 0\ \text{else}\ 1 + Z(n-1). \tag{11.1}$$

[9]We should rather say: formalises many of the concepts that we have seen in this *book*. The λ-calculus was a fundamental inspiration to the design of programming languages starting with ALGOL and LISP. Concepts such as scope, local environment, call by name, to mention only the most important, have been "imported" into programming languages on the basis of analogous concepts in the λ-calculus.

[10]We use this extension only to simplify the presentation. In the λ-calculus, it is possible to *codify* numbers and conditions using only variables, abstractions and applications.

Our experience with programming languages suggests, perhaps, that we should read this relation as a *definition*:

$$Z \overset{def}{=} \lambda n.\ \text{if } n = 0 \text{ then } 0 \text{ else } 1 + Z(n-1).$$

But this is not the only reading possible (in particular, it is not possible in the λ-calculus, which has no concept of global environment to allow us to use the name of a term inside an expression). A mathematician would probably read it as an *equation* with unknown Z (recall the discussion on inductive definitions in Sect. 6.5). For this very simple equation, it is clear that one solution (rather, in this case, the unique solution) is the identity function:

$$Id(n) = n. \tag{11.2}$$

What we are looking for is a general method which allows us to pass from an equation such as (11.1) to an *algorithm*, that is to a λ-term, which is a solution to this equation when the equality symbol is interpreted as β-equivalence:

$$Z =_\beta \lambda n.\ \text{if } n = 0 \text{ then } 0 \text{ else } 1 + Z(n-1). \tag{11.3}$$

Note that this is to ask more than simply provide an arbitrary function such as (11.2). It requires a term which *in the calculus* satisfies (11.3).[11]

Let us consider, first, the function that is obtained from (11.3) by λ-abstracting on the unknown function:

$$F \overset{def}{=} \lambda f.\lambda n.\ \text{if } n = 0 \text{ then } 0 \text{ else } 1 + f(n-1).$$

Hence our problem is to find a suitable term g such that:

$$g =_\beta F(g).$$

A term, g, which satisfies an equation of this type is called a *fixed point* of F. Therefore our problem of finding a solution to (11.3) can be reformulated as a problem of finding the fixed point of some function.

We want now to show that the λ-calculus provides a general method "automatically" to solve these equations. Let us, indeed, consider the following term, which for convenience (and following tradition), we name Y:

$$Y \overset{def}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

This is a term which can be reduced an infinite number of times without reaching normal form. But it can be put to good use. Let us indeed try to apply it to an

[11]The term $\lambda n.n$ is an algorithm for the only function which satisfies (11.1), but it is *not* a solution of (11.3), which expresses a determinate behaviour through reduction.

arbitrary term, M:

$$\begin{aligned} YM &=_\beta (\lambda x.M(xx))(\lambda x.M(xx)) \\ &=_\beta M((\lambda x.M(xx))(\lambda x.M(xx))) \\ &=_\beta M(YM). \end{aligned}$$

In the last step, we have used the fact that was established in the first line, that

$$YM =_\beta (\lambda x.M(xx))(\lambda x.M(xx))$$

and we have then substituted equals for equals. YM is therefore a fixed point of M: Y is a term in the calculus which can compute the fixed point of an arbitrary term.[12] We say that Y is a *fixpoint operator*. It is then quite clear that Y provides solutions to any recursive equation. In particular, it gives us a solution to (11.3):

$$Id \overset{def}{=} YF.$$

The term *Id* thus defined has a reduction behaviour which satisfies the relation with which we started. It is instructive to try to apply this function to a specific argument to see how the mechanism of fixed points is really capable of computing a recursive function. For example:

$$\begin{aligned} Id\,2 &\overset{def}{=} (YF)\,2 \\ &\to F(YF)\,2 \\ &\to \text{if } 2 = 0 \text{ then } 0 \text{ else } 1 + ((YF)(2-1)) \\ &\to 1 + ((YF)\,1) \\ &\to 1 + (F\,(YF)\,1) \\ &\to 1 + ((\text{if } 1 = 0 \text{ then } 0 \text{ else } 1 + ((YF)(1-1)))) \\ &\to 1 + (1 + ((YF)\,0)) \\ &\to 1 + (1 + (\text{if } 0 = 0 \text{ then } 0 \text{ else } 1 + ((YF)\,(0-1)))) \\ &\to 1 + (1 + 0) \\ &\to 2. \end{aligned}$$

At each "recursive call", YF provides a new copy of the body of the function which is used on the next call.

What we have just seen is also true for the definition of recursive values which are not functions. The stream that we defined in Sect. 11.3.5 as:

[12]This shows also that, in the λ-calculus, every term has always at least one fixed point, that computed by Y.

```
val infinity2 = 2 :: infinity2;
```

is nothing more than a fixed point:

$$\texttt{infinity2} \overset{def}{=} Y(\lambda l.\, 2 :: l).$$

Observe that Y is a fixpoint operator with respect to β-equality.[13] If, instead of β-reduction, we use a different strategy, for example, the analogue of reduction by value which we defined for functional languages, Y is no longer able to compute fixed points. For reduction by value, another operator must be used, for example:

$$H \overset{def}{=} \lambda g.((\lambda f.ff)(\lambda f.(g(\lambda x.ffx)))).$$

For reasons of efficiency, implementations of functional programming languages do not use fixpoint operators to implement recursion. However, fixpoint operators play a role that is very important to the theory of these languages.

Expressiveness of the λ-calculus This very simple formal system, constructed only from the concepts of application and abstraction, is not as rudimentary as it seems, if it can express arbitrary recursions. Indeed, it is possible to show that the λ-calculus is a Turing-complete formalism. In the first place, it is possible to encode the natural numbers as λ-terms (a λ-term, $\underline{n}$ is associated with the number n) and, then, show that to every computable function, f, there corresponds a λ-term M_f which computes f: if $f(n) = m$, then $M_f \underline{n} \to \underline{m}$.

11.7 Chapter Summary

In this chapter, we have presented the functional programming paradigm. In its *pure* form, this is a computational model which does not contain the concept of modifiable variable. Computation proceeds by the rewriting of terms that denote functions. The main concepts that we have discussed are:

- *Abstraction*. A mechanism for passing from one expression denoting a value to one denoting a function.
- *Application* A mechanism dual to abstraction, by which a function is applied to an argument.
- *Computation by rewriting* or *reduction*, in which an expression is repeatedly simplified until a form that can not be further reduced is encountered.
- *Redex*. The syntactic structure which is simplified during reduction and is formed from the application of an abstraction to an expression.
- The centrality of *higher order* in this mode of computation.

[13]Clearly it is not unique. For another operator slightly more complex than Y but which satisfies a stronger relation, see Exercise 7.

- The concept of *value* which corresponds to those syntactic forms not otherwise reducible and where reduction therefore terminates.
- The different *evaluation strategies*: by value (or eager), by name, lazy.
- Some mechanisms that the most common functional languages add to the fundamental nucleus. Among these, recall: an interactive environment, a rich system of types, pattern matching, imperative aspects such as variables and assignments.
- The *SECD machine*, a prototype abstract machine for higher-order functional languages using the by-value strategy.
- A comparison of the functional and imperative paradigms, centred on the notion of program correctness.
- The λ-calculus, a simple and powerful formal system that constitutes the formal nucleus of all functional languages.

11.8 Bibliographical Note

Reading the original article by John McCarthy on LISP is still didactically important [7]. An elementary introduction to programming using ML is to be found in [11], while [4] is a more advanced text (which uses the Caml dialect); [9] is the official definition of the language and it is useful to read it together with the commentary [8].

The implementation of functional languages, with particular regard to lazy evaluation is dealt with in [10]. The SECD machine was introduced in the pioneering work of Peter Landin [6].

John Backus' lecture for the Turing Award [1] should be essential reading (at least for its first part) for every student of programming languages.

The lambda calculus was introduced in the 1930s by Alonzo Church. An original reference is [3]. The modern reference text is [2], even if [5] is a clearer and more accessible introduction.

11.9 Exercises

1. Consider the following definitions in ML:

```
fun K x y = x;
fun I x = x;
fun Omega x = Omega x;
```

   State what the result of evaluating the following expression will be.

```
K (I 3) (Omega 1).
```

   What would be the result if ML adopted a lazy strategy?

2. Write a function `map` which applies its first parameter (which will be a function) to all the elements of a list passed as its second argument.
3. Describe formally the operation of the SECD machine, by giving a definition using the structural operational semantics that we introduced in Sect. 2.5.
4. In λ-calculus, we define the code of a natural number, n, as:

$$\underline{n} \overset{def}{=} \lambda f.\lambda x.f^n x \overset{def}{=} \lambda f.\lambda x.f(f \cdots (f x)\cdots),$$

   where, in the body, there are n occurrences of f. Give the definition of a term, *prod* such that:

$$prod\, \underline{n}\, \underline{m} \to \underline{n * m}.$$

5. Apply the technique of calculating fixed points to solve the following equation for the unknown *fact*:

$$fact =_\beta \lambda n.\ \text{if}\ n = 0\ \text{then}\ 1\ \text{else}\ n * fact(n-1)$$

   and describe the reduction which computes the value of its solution applied to argument 2.
6. Apply the technique of calculating fixed points to solve the following equation for the unknown h:

$$h =_\beta \lambda x.\ \text{if}\ x > 100\ \text{then}\ x - 10\ \text{else}\ h(h(x+11)).$$

   Call the solution *Ninetyone*, then calculate *Ninetyone(99)*.
7. Consider the λ-terms:

$$A = \lambda x.\lambda y.y(xxy),$$
$$\Theta = AA.$$

   Show that Θ is a fixed-point operator for which the relation:

$$\Theta M \to \cdots \to M\ (\Theta M).$$

   is true. This relation is stronger than that satisfied by Y, for which only $YM =_\beta M(YM)$ is true. The term, Θ, is called the Turing fixpoint operator.

References

1. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978. doi:10.1145/359576.359579.
2. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, Amsterdam, 1984.
3. A. Church. *The Calculi of Lambda Conversion*. Princeton Univ. Press, Princeton, 1941.
4. G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge Univ. Press, Cambridge, 1998.

5. R. Hindley and P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge Univ. Press, Cambridge, 1986.
6. P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. citeseer.ist.psu.edu/cardelli85understanding.html.
7. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960. doi:10.1145/367177.367199.
8. R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, 1991.
9. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML—Revised*. MIT Press, Cambridge, 1997.
10. S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, New York, 1987. Online at http://research.microsoft.com/Users/simonpj/papers/slpj-book-1987/index.htm.
11. J. D. Ullman. *Elements of ML Programming*. Prentice-Hall, Upper Saddle River, 1994.