Small Introduction to Functional Programming for Imperative Programmers

Luca Abeni

December 9, 2022

1 Programming Styles and Paradigms

Functional programming is a programming style (or *paradigm*); although some programming languages help in using this programming style (or even force the programmer to use it), it is possible to develop programs using the functional programming paradigm independently from the used programming language (it is even possible to use a programming languages that are traditionally considered imperative, like the C programming language).

To better understand what functional programming really is (and how to use this programming style), consider Euclide's algorithm to compute the great common divisor (gcd) of two natural numbers. This algorithm is something like: "*given two natural numbers a and b, if $b = 0$ then return a; otherwise, assign the value of b to a, assign $a\%b$ (reminder of the division a/b) to b, and restart from beginning*".

A simple implementation of this algorithm using an imperative programming paradigm can be obtained by simply "traslating" the algorithm to a computer programming language. As an example, Figure 1 shows the algorithm's translation into the C programming language. This program is a sequence of instructions (actions) operating on values stored in memory or I/O devices (some kind of mutable shared state) and directly maps to the architecture of a modern computer (which is based on an evolution of the so-called "Von Neumann architecture").

Looking at the code, it is immediately possible to understand that it correctly implements the algorithm mentioned above, but it is not immediately possible to understand *why this function correctly computes the gcd of the two aruments* (in other words, why does the Eclide algorithm work? Can we prove that it correctly computes the gcd?). The behaviour of the algorithm can be more or less formally explained as follows:

- The greatest common divisor between a and 0 is a, because 0 can be divided by any number, with reminder 0
- The greatest common divisor between a and $b \neq 0$ is equal to the maximum common divisor between b and $a\%b$ (this can be proved by induction)

From the mathematical point of view, this can be written as

$$gcd(a, b) = \begin{cases} \text{a} & \text{if b = 0} \\ \text{gcd(b, a \% b)} & \text{otherwise} \end{cases}$$

and this equation can result in a new implementation of the gcd function, as shown in Figure 2. This implementation is however not structured, because it has an entry point and 2 different exit points (there are 2 `return` statements). This "issue" can be addressed by using the so-called "arithmetic if", as shown in Figure 3.

Comparing Figure 1 (called "imperative implementation") with Figure 3 (called "functional implementation"), some important differences can be immediately noticed:

- While the imperative implementation executes modifying the values stored in some local variables "`a`", "`b`", and "`tmp`", the functional implementation does not change the value stored in any variable (that is, it does not use the assignment operator "`=`")
- While the imperative implementation uses a "`while(b != 0)`" loop, the functional implementation uses recursion (using the "`b == 0`" condition to stop recursion / as an inductive base)

```c
unsigned int gcd(unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int tmp;

    tmp = b;
    b = a % b;
    a = tmp;
  }

  return a;
}
```

Figure 1: Euclide's algorithm implemented in C.

```c
unsigned int gcd(unsigned int a, unsigned int b)
{
  if (b == 0) {
    return a;
  }

  return gcd(b, a % b);
}
```

Figure 2: Euclide's algorithm implemented through recursion.

```c
unsigned int gcd(unsigned int a, unsigned int b)
{
  return (b == 0) ? a : gcd(b, a % b);
}
```

Figure 3: Purely functional implementation of the Euclide's algorithm.

The second property of the functional implementation (using recursion instead of iteration) is a consequence of the first one (not changing the values stored in variables): a "`while`" loop is based on a predicate ("`b != 0`", in this case) which is evaluated (repeating the execution of the loop at every evaluation) until it becomes false. The truth value of such a predicate depends on the values stored in one or more variables (variable "`b`", in this case); if such values do not change, predicate will be always true or always false, making the "`while()`" loop useless (the loop will repeat forever, or will never be executed). For this reason, if variables are immutable the iteration constructs like `while()` and similar cannot be used.

A functional implementation of an algorithm is based on *pure functions*, which are mathematical functions (relations $f \subset \mathcal{D} \times \mathcal{C}$ between a set $\mathcal{D}$ called domain and a set $\mathcal{C}$ called codomain, associating *at most* an element of the codomain to each element of the domain). These functions are characterized by the absence of *side effects*. More formally, $f(x) = y$ means $(x, y) \in f$ and indicates that:

- function f always associates the same value $y \in \mathcal{C}$ to value $x \in \mathcal{D}$
- computing y from x is the only effect of the function

Since modifying the value stored in a variable is a side effect of the assignment operator "`=`", pure functions cannot use assignment operators. Hence, **the functional programming paradigm does not allow to use mutable variables**. As just explained, the first consequence of this requirement is that loops cannot be used and are replaced by recursive invocations. All the commands having side effects cannot be used (only expressions without side effects are used), and programs are not executed modifying some mutable state, but evaluating expressions. For this reason, the traditional "`if`" selection command (which selects the alternative execution of two different commands) is not used, but the *arithmetic if*

```
int f(int v)
{
  static int acc;

  acc = acc + 1;

  return v + acc;
}
```

Figure 4: Example of non-pure function, having a side effect.

expression (the "`... ? ... : ...` construct of the C language) is used instead. This expression evaluates to one of two different expressions depending on the truth value of a predicate. A noticeable consequence of using arithmetic if expressions (and not selection commands) is that the "`else`" branch must always be present.

Multiple evaluations of the same expression (or multiple invocations of the same pure function) must always generate the same result. This property looks intuitive and natural for mathematical functions, but it is not respected by functions having side effects. As an example, look at Figure 4: some consecutive invocations f(2); f(2); f(2); will generate different results (2, 3 e 4). To understand why this can be an issue, consider the expression (f(2) + 1) ∗ (f(2) + 5): if the invocation of $f(2)$ on the left is evaluated first, the result of the expression is $(2 + 1) * (3 + 5) = 24$, otherwise it is $(3 + 1) * (2 + 5) = 28$.

2 Recursion and Iteration

While the basic constructs of imperative programming are (according to the structured approach) the command sequence, the selection command (conditional execution, `if`) and the loop (for example, `while`), the basics constructs of functional programming are function invocation, the arithmetic if expression, and recursion.

In particular, the functional equivalent of a loop (iteration) is recursive function invocation (recursion): every imperative algorithm that requires a (finite or infinite) cycle, when coded according to the functional paradigm results in a recursion (again, finite or infinite).

The recursion technique (closely related to the mathematical concept of *induction*) is used in computer science to define some kind of "entity"[1] based on itself. Focusing on recursive functions, a function $f()$ is defined by expressing the value of $f(n)$ as a function of other values computed by $f()$ (typically, $f(n-1)$).

In general, recursive definitions are given "by case", ie they are composed of several clauses. One of these is the so-called *basis* (also called *inductive basis*); then there are one or more clauses or *inductive steps* which allow to generate / calculate new values starting from existing values. The basis is a clause of the recursive definition that does not refer to the "entity" being defined (for example: the greatest common divisor of a and 0 is a, etc...) and stops the recursion: without an inductive basis, the function evaluation results in an infinite recursion.

Basically, a function $f : \mathcal{N} \to \mathcal{X}$ can be defined by defining a function $g : \mathcal{N} \times \mathcal{X} \to \mathcal{X}$, a value $f(0) = a$ and imposing that $f(n+1) = g(n, f(n))$. More in detail, a function can be defined by recursion when its domain is the set of natural numbers (or a countable set); the codomain can instead be a generic set $\mathcal{X}$. The inductive basis defines the value of the function for the smallest value belonging to the domain (for example, $f(0) = a$, with $a \in \mathcal{X}$), while the inductive step defines the value of $f(n + 1)$ based on the value of $f(n)$. As mentioned earlier, this can be done by defining $f(n + 1) = g(n, f(n)))$. Note that the domain of $g()$ is the set of pairs containing an element fo the domain and an element of the codomain of $f()$, while the codomain of $g()$ is equal to the codomain of $f()$.

The typical example of recursive function is the factorial function, shown in Figure 5. A more functional version (because it uses only expressions, and not commands) of the factorial is instead shown in Figure 6

This example can be useful to see that the execution of a program written according to the functional paradigm can be seen as a sequence of "simplifications" or "substitutions" (technically, *reductions*) similar to the ones used to evaluate an arithmetic expression. For example, consider the computation of `fact(4)`. The definition of the `fact()` function has been defined modifies the environment by introducing a binding

[1] The term "entity" is used here informally to generically refer to functions, sets, values, data types, ...

```
unsigned int fact(unsigned int n)
{
  if (n == 0) {
    return 1;
  }

  return n * fact(n - 1);
}
```

Figure 5: Recursive implementation of the factorial function.

```
unsigned int fact(unsigned int n)
{
  return (n == 0) ? 1 : n * fact(n - 1);
}
```

Figure 6: Functional implementation of the factorial function.

between the name "`fact`" and a denotable entity (the body of the factorial function). To evaluate the expression "`fact(4)`" it is therefore possible to search the environment for such a binding and replace "`fact`" with its definition, using ' '`4`" (actual parameter) instead of the formal parameter "`n`": fact (4) $\rightarrow$ (4 == 0) ? 1 : 4 $*$ fact(4 $-$ 1) $\rightarrow$ 4 $*$ fact (3) where the first step replaces the name "`fact`" with the function body (according to the binding found in the environment) and replaces the name of the formal parameter "`n`" with the value of the actual parameter "`4`". The second step evaluates "`4 == 0`"; since this boolean predicate is false ($4 \neq 0$), the second expression "4 $*$ fattoriale (4 $-$ 1)" is evaluated. Moreover, since $4 - 1 = 3$ "`fact(4 - 1)` is replaced by "`fact(3)`". At this point, the reduction process can start again, searching again for a binding for "`fact`" in the environment and performing the same replacements as above: 4 $*$ fact (3) $\rightarrow$ 4 $*$ ((3 == 0) ? 1 : 3 $*$ fact(3 $-$ 1)) $\rightarrow$ 4 $*$ (3 $*$ fact (2)).

Repeating this process again, we get: 4 $*$ (3 $*$ fact (2)) $\rightarrow$ 4 $*$ (3 $*$ ((2 == 0) ? 1 : 2 $*$ fact(2 $-$ 1))) $\rightarrow$

$\rightarrow$ 4 $*$ (3 $*$ (2 $*$ fact (1))) $\rightarrow$ 4 $*$ (3 $*$ (2 $*$ ((1 == 0) ? 1 : 1 $*$ fact(1 $-$ 1)))) $\rightarrow$
$\rightarrow$ 4 $*$ (3 $*$ (2 $*$ (1 $*$ fact (0)))) $\rightarrow$
$\rightarrow$ 4 $*$ (3 $*$ (2 $*$ (1 $*$ ((0 == 0) ? 1 : 0 $*$ fact(0 $-$ 1))))) $\rightarrow$
$\rightarrow$ 4 $*$ (3 $*$ (2 $*$ (1 $*$ 1))) = 24.

From a logical point of view the computation of the factorial only required to:

1. search in the environment (to apply a function to its actual parameters, you need to find the binding between the function name and its body in the environment)
2. replace some text (the application of a function to its arguments is obtained by replacing the formal parameters with the actual parameters in the body of the function found in item 1)
3. compute arithmetic operations (this includes both "simple" arithmetic operations such as products and subtractions as well as evaluating arithmetic ifs)

This example shows that according to the functional paradigm the a program is executed by reduction, i.e. by textual replacement of expressions and sub-expressions: a function applied to an argument is replaced by the body of the function and the formal parameter is replaced by the actual parameter. Conceptually, this reduction process does not require the execution of assembly instructions or programs, but only manipulations of text strings.

Clearly, this concept of computation as reduction is applicable only to pure functions (functions without side effects): if `fact()` had side effects (such as the modification of global variables, or similar) it would not be possible to replace " factorial (4)" with "4 $*$ factorial (3)" (because the function's side effects would be lost). This is why the absence of side effects is (as already anticipated) a fundamental requirement for the functional programming paradigm. Eliminating mutable variables is an easy way to eliminate a whole large class of side effects (the side effects due to I/O remain, but these cannot be eliminated without rendering the program useless).

The functional programming paradigm, which has been presented in these pages as an alternative to the "traditional" imperative paradigm, has the same expressive power as the imperative paradigm (i.e.: any algorithm that can be codified using an imperative approach can also be implemented using the functional approach). Readers more accustomed to the imperative approach may object that "giving up" modifiable variables and iteration seems to complicate program development and that consequently the functional approach may appear a bit unnatural. Actually this is more a problem of getting used to and once you understand the logic of functional programming, developing programs according to this approach will be easier. Furthermore, some problems are more easily solved using recursion and are not exactly simple to solve using a purely imperative approach. For example, consider the problem of the "Towers of Hanoi".

The problem consists in moving a tower composed of N disks (of decreasing size) from a peg to a different one, using a third "spare" (or "support") peg for the movements. The rules of the game state that only one disc can be moved at a time and that a larger disc cannot be placed on top of a smaller one. While developing a non-recursive solution to the problem is not very simple (and requires the use of complex data structures), a recursive solution is trivial. In practice, the problem of moving N disks from peg a to peg b (using peg c as spare/support) can be decomposed ias:

- Move $N-1$ disks from peg a to peg c
- Move remaining (largest) disk from peg a to peg b
- Move the $N-1$ disks from peg c to peg b

Now, while the second step of the algorithm (moving a disk from a peg to another) is simple, the first and third steps require moving $N-1$ disks and are not directly implementable. But if we know how to move N disks, we can (recursively!) use the same algorithm to move $N-1$ disks (and this will require moving $N-2$ disks, then one disk and then $N-2$ disks again). And this recursion can be invoked multiple times (to be exact, $N-1$ times) until the problem is reduced to moving one single disk.

Figure 7 shows a simple implementation of this algorithm using the C language. In this case the condition for terminating the recursion (inductive basis) corresponds to moving one single disk (as in the algorithm described above). An alternative implementation could have used the condition "`n == 0`" (no disks to move) as an inductive basis, resulting in the `move()` function shown in Figure 8.

An important consideration to make about the proposed code is that although it uses recursion it is not yet implemented according to a purely functional approach: the `move()` function uses a sequence of commands (`move( )` and `move_disk()` have no return value) and only works thanks to the side effects of these commands (printing to the screen via `printf()`). To implement `move()` as a pure function, these side effects have to be removed (i.e., eliminate the call to `printf()` from `move_disk()`, adding a return value to `move()` and `move_disk()`). The simplest solution is to make `move()` and `move_disk()` return a string containing their output (in other words, the sequence of moves must not be printed to the screen via `printf()`, but saved in the return value of the function). A possible implementation (unfortunately not very readable due to the syntax of the C language) is shown in Figure 9.

In this solution, the utility function `concat()` receives two strings (in C, array of characters) in input and returns a string containing (as the name suggests) the concatenation of the two. There are a few important things to note:

- Using the functional notation for `concat()` (instead of an infix operator, as in other languages) reduces the readability of the code. However, the reader should not be confused by long chains "`concat(concat(concat(...)))`", which do nothing but concatenate long sequences of strings
- The `move()` and `move_disk()` functions are now *pure functions*, as they don't have side effects
- All the side effects are now in the `main()` function, which performs I/O... It is clear that if the program has to communicate with the external environment then some I/O (which is a side effect) is needed; a program can therefore never be "purely functional", but will tend to concentrate all side effects in specific points (in this case, the `main()` function; in functional languages, the Read-Evaluate-Print loop or some runtime support)
- Confirming the fact that they are pure functions, `move()` and `move_disk()` do not modify the contents of variables (they do not use assignments, or other commands, but are composed only of expressions)

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void move_disk(const char *from, const char *to)
{
  printf("Move a disk from %s to %s\n", from, to);
}

void move(unsigned int n, const char *from, const char *to, const char *via)
{
  if (n == 1) {
    move_disk(from, to);

    return;
  }

  move(n - 1, from, via, to);
  move_disk(from, to);
  move(n - 1, via, to, from);
}

int main(int argc, char *argv[])
{
  unsigned int height = 0;

  if (argc > 1) {
    height = atoi(argv[1]);
  }
  if (height <= 0) {
    height = 8;
  }

  move(height, "Left", "Right", "Center");

  return 0;
}
```

Figure 7: Solution to the problem of the Towers of Hanoi.

```c
void move(unsigned int n, const char *from, const char *to, const char *via)
{
  if (n == 0) {
    return;
  }

  move(n - 1, from, via, to);
  move_disk(from, to);
  move(n - 1, via, to, from);
}
```

Figure 8: Alternative solution.

- The most expert readers might have noticed that the program has memory leaks: `concat()` dynamically allocates memory for the returned string, but that memory is never freed. This fact is a

```
const char *concat(const char *a, const char *b)
{
  char *res;

  res = malloc(strlen(a) + strlen(b) + 1);
  memcpy(res, a, strlen(a));
  memcpy(res + strlen(a), b, strlen(b));
  res[strlen(a) + strlen(b)] = 0;

  return res;
}

const char *move_disk(const char *from, const char *to)
{
  return concat(concat(concat(concat("Move␣disk␣from␣", from),
                "␣to␣"), to), "\n");
}

const char *move(int n, const char *from, const char *to, const char *via)
{
  return (n == 1) ?
      move_disk(from, to)
    :
      concat(concat(move(n - 1, from, via, to),
                move_disk(from, to)), move(n - 1, via, to, from));
}
```

Figure 9: Functional solution to the problem of the Towers of Hanoi.

consequence of the previous item (the content of the strings is never modified, but the concatenation of two strings occurs by dynamically allocating memory for the result string) and shows how the functional programming paradigm requires a *garbage collector* (which is in fact always included in abstract machines that implement functional programming languages)

Most of the drawbasks of the code shown in Figure 9 are not due to the functional programming style, but are a consequence of the C programming language syntax (for example, the C language has no real strings, but only array of characters). Figure 10 shows a re-implementation in C++ (which is similar to C, but provides a more advanced support for strings). From the figure it can be seen that this code is much more readable and looks more natural.

3 Recursion and Stack

As seen, to evaluate an expression by substitution/reduction it is not conceptually necessary to introduce the concepts of invocation of subroutines, stacks, activation records and the like. On the other hand, if the abstract machine that executes the program (or, evaluates the expression) is implemented on a hardware architecture based on the Von Neumann model (like all modern PCs) it will use the subroutine call mechanism (Assembly instruction `call` on Intel architectures, etc...) to invoke the execution of a function.

Returning to the previous example (factorial function), every time the `fact()` function invokes itself a new activation record (or stack frame) is pushed to the stack, increasing its size (this also applies to generic - non recursivo - invocations of other functions). Each recursive invocation of the function will add an activation record (containing the actual parameter with which `fact()` was invoked, some links to previous stack frames, and some space to store the return value) on the stack. Hence, invoking "`fact(4)`" results in the situation shown in Figure 11.

Since "`fact(3)`" will also be recursively invoked, the stack evolves as shown in Figure 12, growing at each recursive invocation. This means that computing the factorial of a large enough number `n` will require a large amount of memory. Note that the activation record corresponding to "`fact(n)`" cannot

```
#include <cstdlib>
#include <iostream>
#include <string>

std::string move_disk(std::string from, std::string to)
{
  return "Move disk from " + from + " to " + to + "\n";
}

std::string move(int n, std::string from, std::string to, std::string via)
{
  return (n == 1) ?
      move_disk(from, to)
    :
      move(n - 1, from, via, to) +
                move_disk(from, to) + move(n - 1, via, to, from);
}

int main(int argc, char *argv[])
{
  int height = 0;
  std::string res;

  if (argc > 1) {
    height = atoi(argv[1]);
  }
  if (height <= 0) {
    height = 8;
  }

  res = move(height, "Left", "Right", "Center");

  std::cout << res;

  return 0;
}
```

Figure 10: Functional solution to the "Towers of Hanoi" problem written in C++.

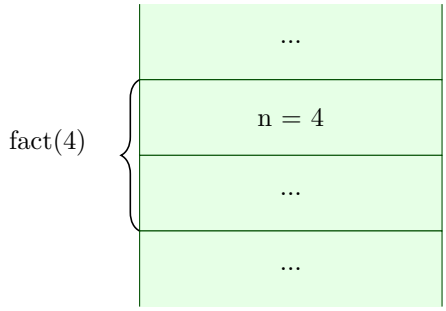

Figure 11: Stack frame for the invocation `fact(4)`.

be removed from the stack until "`fact(n - 1)`" returns, because it contains the value "`n`" by which the result of "`fact(n - 1)`" must be multiplied. Basically, when "`fac(0)`" is evaluated, the previous stack frames contain all the numbers to be multiplied; as the various instances of `fact()` return, their stack frames are removed from the stack one after the other (after using the value of "`n`" contained in the stack frames). The various stack frames are then needed until the related "`fact`" instance finishes, and they

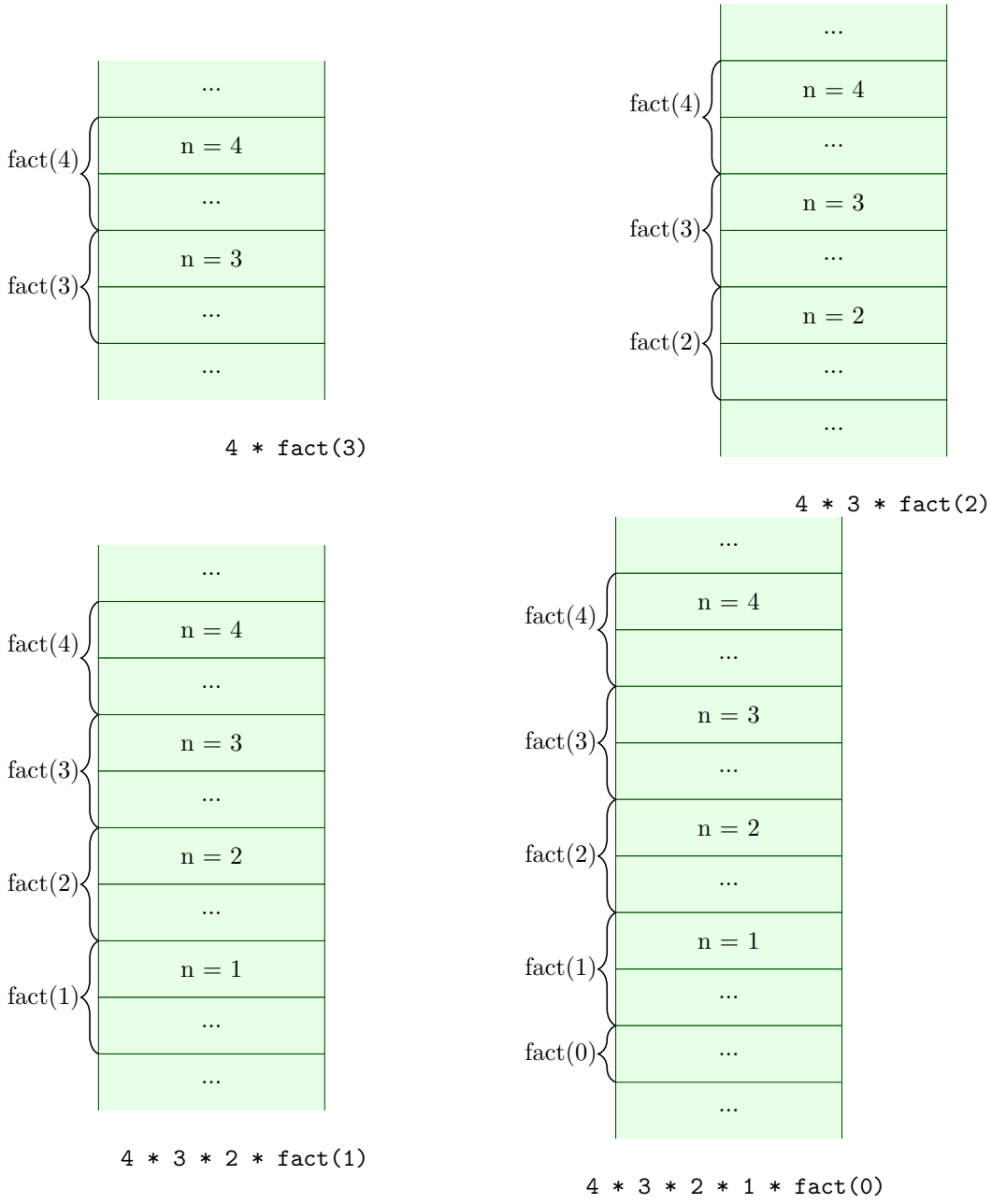

Figure 12: Evolution of the stack when `fact(4)` is invoked.

```c
/* Try 24635743... */

unsigned int even(unsigned int n);
unsigned int odd(unsigned int n)
{
  if (n == 0) return 0;
  return even(n - 1);
}

unsigned int even(unsigned int n)
{
  if (n == 0) return 1;
  return odd(n - 1);
}
```

Figure 13: Example of mutual recursion to test if a number is even or odd.

```c
unsigned int fact_tr(unsigned int n, unsigned int res)
{
  return (n == 0) ? res : fact_tr(n - 1, n * res);
}

unsigned int fact(unsigned int n)
{
  return fact_tr(n, 1);
}
```

Figure 14: Tail recursive version of the factorial function.

cannot be removed from the stack before that.

This problem seems to compromise the real usability of functional programming techniques, since long chains of recursive calls would lead to excessive memory consumption (while long iterations generally have low and constant memory consumption). As another example, look at Figure 13, that presents two function to test if a number is even or odd in a funny (mutually recursive) way. The proposed solution uses a mutual recursion between the `even()` and `odd()` functions, based on the idea that 0 is even, 1 is odd (inductive bases) and every number $n > 1$ is even if $n - 1$ is odd or is odd if $n - 1$ is even (inductive step). By looking at the code it is immediately possible to realize that calling `even()` or `odd()` on large numbers could end up in a stack overflow. In fact, the program is compiled with `gcc evenorodd.c` and tested for small numbers, everything seems to work... But when trying big enough numbers (for example 24635743, as suggested in the comment) a segmentation fault is generated due to stack overflow. However, if the program is compiled with `gcc -O2 evenorodd.c`, it works correctly with any input number! This happens because of the so-called "tail call optimization" (enabled by the "`-O2`" switch of tt gcc), which allows, under appropriate hypotheses, to replace function invocations with simple jumps (thus transforming recursion into iteration).

To better understand how this optimization works, let's consider the "tail recursive" version of the factorial, shown in Figure 14. Intuitively, the `fact_tr()` function uses a second argument to "accumulate" the result: multiplication by "`n`" occurs *before* the recursive call (to compute the value of the second actual parameter) and not after. This means that the "`n`" value will not be needed when the recursive call returns and it is hence not necessary to save it... The expression "`fact(4)`" is evaluated as follows: $\text{fact}(4) \rightarrow \text{fact\_tr}(4,\ 1) \rightarrow (4 == 0)\ ?\ 1 :\ \text{fact\_tr}(4 - 1,\ 4 * 1) \rightarrow$

$\rightarrow\ \text{fact\_tr}(3,\ 4) \rightarrow (3 == 0)\ ?\ 4 :\ \text{fact\_tr}(3 - 1,\ 3 * 4) \rightarrow$

$\rightarrow\ \text{fact\_tr}(2,\ 12) \rightarrow (2 == 0)\ ?\ 12 :\ \text{fact\_tr}(2 - 1,\ 2 * 12) \rightarrow$

$\rightarrow\ \text{fact\_tr}(1,\ 24) \rightarrow (1 == 0)\ ?\ 24 :\ \text{fact\_tr}(1 - 1,\ 1 * 24) \rightarrow$

$\rightarrow\ \text{fact\_tr}(0,\ 24) \rightarrow (0 == 0)\ ?\ 24 :\ \text{fact\_tr}(0 - 1,\ 0 * 24) \rightarrow 24$

The key observation here is that when "`fact_tr(0, 24)`" returns, "`fact_tr(1, 24)`" can immediately return the result of `fact_tr(0, 24)`... The same holds for "`fact_tr(2, 12)`", "`fact_tr(3, 4)`" and "

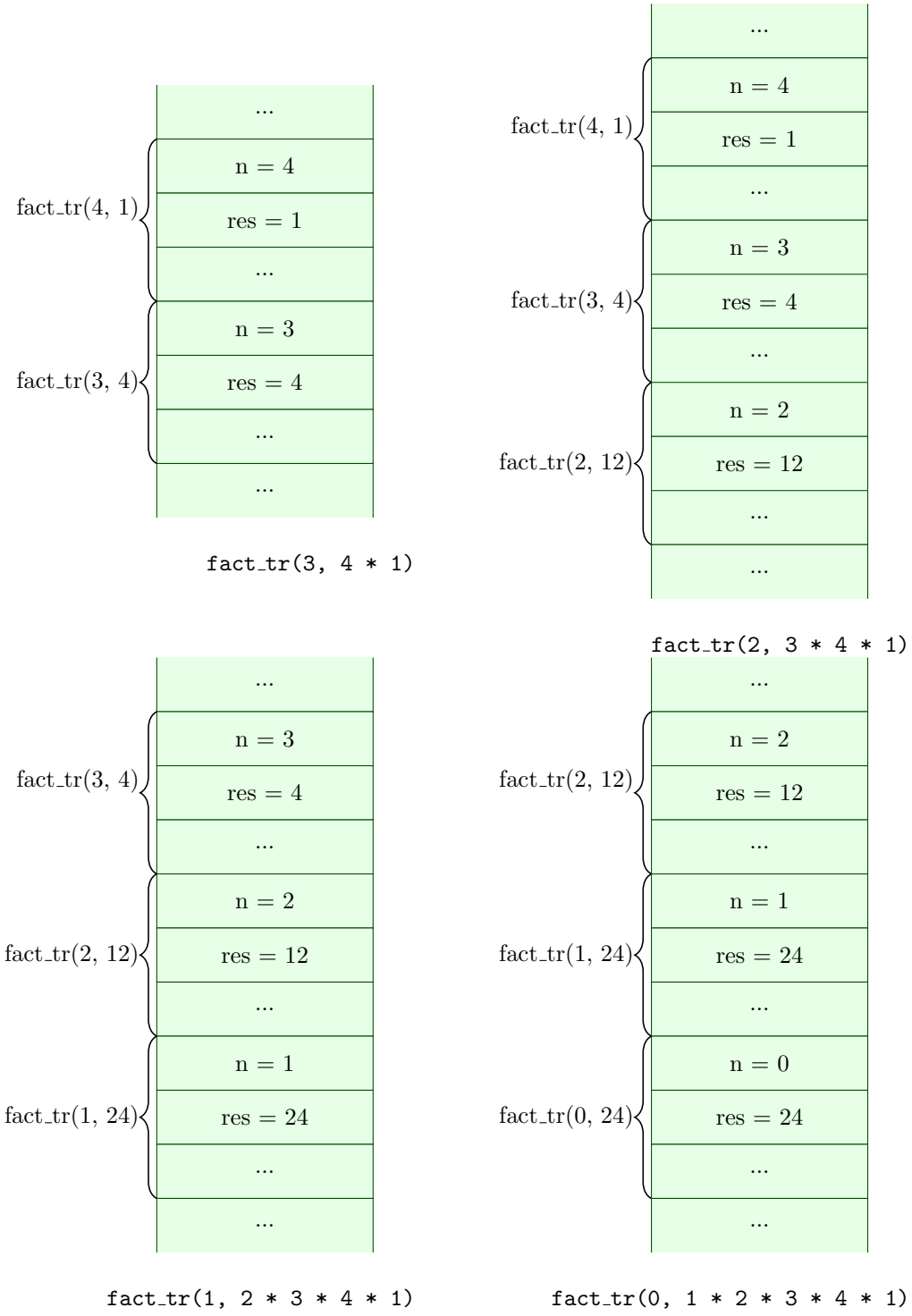

Figure 15: Evolution of the stack for the invocation of`fact(4)`.

```
fact_tr:
        movl    %esi, %eax
        testl   %edi, %edi
        je      .L2
        subq    $8, %rsp
        imull   %edi, %esi
        subl    $1, %edi
        call    fact_tr
        addq    $8, %rsp
.L2:
        ret
```

Figure 16: Tail-call factorial compiled without optimizations.

```
fattoriale_tr:
        movl    %esi, %eax
        testl   %edi, %edi
        je      .L2
        imull   %edi, %esi
        subl    $1, %edi
        jmp     fact_tr
.L2:
        ret
```

Figure 17: Tail-call factorial compiled with optimizations.

`fact_tr(4, 1)`" (everyone of these functions can immediately return the value returned by its recursive invocation). Thus, activation records containing the value of "`n`", the return value and the return address are no longer needed: "`fact_tr(0, 24)`" can directly return the result 24 to the original caller "`fact(4)`", without passing through "`fact_tr(1, 24)`", "`fact_tr(2, 12)`", etc... Figure 15 shows the details of the stack evolution resulting from the invocation of "`fact(4)`", clarifying even more that during the execution of an instance of `fact_tr()` the activation records corresponding to the previous instances contain data that will not be used anymore (and is therefore unnecessary / useless!!!). Basically, when "`fact_tr(0, 24)`" is evaluated, all the data necessary for the calculation are contained in the current parameters and the activation records of "`fact_tr (1, 24)`"..."`fact_tr(4, 1)`" that are on the stack are not accessed. Such activation records are removed from the stack one after the other (when the various instances of `fact_tr()` return) without having to do any further operations on them: when "`fact_tr(n - 1, ...)`" terminates, "`fact_tr(n, ...)`" directly returns its return value, without performing any further operations on it. This means that when the recursive call returns, each instance of `fact_tr()` can terminate immediately, passing the return value received from the recursive call directly to its caller. The various stack frames can then be removed from the stack at recursion time (before the associated function terminates), effectively turning a recursive call into a simple jump.

This optimization is possible whenever a function returns as return value the result obtained by calling another function (basically, "`return otherfunction(...)`"). In practice, statements like "**return** f(n);" are not compiled as invocations to the subroutine `f()`, but as jumps to the body of that function. This allows you to implement recursion without causing excessive stack consumption, making it feasible to use the functional programming paradigm (provided you write code that uses tail calls).

To understand how tail call optimization works in practice, consider the x86_64 assembly code generated by `gcc -O1` when compiling the function `fact_tr()` of Figure 14. This code is shown in Figure 16.

The first instruction (`movl`) copies the second argument (contained in the `%esi` register) into the `%eax` register (which will be used for the return value); the second instruction (`testl`) checks whether the first argument (contained in the register `%edi`) is 0: in this case, it immediately jumps to the exit of the function (label `.L2`) returning the value contained in `%eax` (into which the second argument was just copied). If, instead, the first argument is > 0, the function multiplies the second argument by the first and then invokes itself recursively (the statements `subq` and `addq` applied to `%rsp` are required due to Intel 64-bit ABI calling conventions). Note that upon return from the recursive call (statement

```
int sum(int a, int b)
{
  return a + b;
}
```

Figure 18: C function summing 2 integers.

```
int (* sum_c(int a))(int b)
{
  int s(int b) {
    return a + b;
  }

  return s;
}
```

Figure 19: Attempt at curryifying the `sum2()` function (Figur2 18) using the C language.

following the `call`) no further statements are executed and the function terminates immediately. So, it is possible to avoid pushing successive useless return addresses onto the stack (when an instance of `fact_tr()` returns, all previous instances will return immediately, one after the other, without executing Assembly instructions between various "`ret`"). This can be done simply by eliminating the instructions that manipulate the stack (register `%rsp`) and replacing the `call fact_tr` with a `jmp fact_tr`, as shown in Figure 17.

4 Functions and Spices

In mathematical analysis, we are used to functions $f : \mathcal{D} \to \mathcal{C}$ which map one or more elements of the domain $\mathcal{D}$ into at most an element of the codomain $\mathcal{C}$. In practice, f is a relation (subset $f \subset \mathcal{D} \times \mathcal{C}$ of the pairs having the first element in $\mathcal{D}$ and the second element in $\mathcal{C}$) with $(x_1, y_1) \in f \wedge (x_1, y_2) \in f \Rightarrow y_1 = y_2$. If f has more than one argument, the domain $\mathcal{D}$ is represented as a cartesian product of other sets: for example, a function from pairs of real numbers to real numbers will be $f : \mathcal{R} \times \mathcal{R} \to \mathcal{R}^2$.

From a computer science point of view, we are instead used to considering multi-argument functions in which each argument is represented by a formal parameter. For example, "**int** f(**int** a, **float** x, **unsigned int** z)" is a function that takes three arguments: the first one is an integer number, the second one represents the approximation of a real number and the third one is a positive integer number (that is, a natural number). This C function can hence be considered equivalent to $f : \mathcal{Z} \times \mathcal{R} \times \mathcal{N} \to \mathcal{Z}$. In some programming languages it is possible to use a *tuple* (a structured type) to group all arguments and better represent values belonging to $\mathcal{Z} \times \mathcal{R} \times \mathcal{N}$.

Several mathematicians showed that any function with multiple arguments can be represented using multiple functions with one single argument. For example, using the so-called *currying*[3] a function with n arguments can be represented as a "chain" of functions having one argument. The "trick" to obtain this result is that each function of this "chain" returns a function (and not a "simple value"). For example, a function $f : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ can be represented as $f : \mathcal{R} \to (\mathcal{R}\ rightarrow \mathcal{R})$.

This fact has some important consequences, both theoretical and practical. The first theoretical consequence is that a mathematical formalism (such as for example the λ-calculus) which considers only functions having a single argument can be perfectly generic, provided that these functions can return functions and accept functions as arguments (such functions are often called *higher-order functions*). The second consequence, which has many practical implications, is therefore the introduction of *higher order functions*, i.e. functions that can manipulate other functions (as parameters or as return value). For this reason, in functional programming languages there is uniformity between code and data, in the sense that functions are values that can be stored into variables and expressed[4].

[2] typically, advanced analysis courses consider functions $f : \mathcal{R}^n \to \ calR^m$).

[3] Note that the name of this technique comes from Haskell Curry, not a spice!

[4] Remember that a value can be stored when it can be assigned to a variable and is expressible when it can be generated as a result of an expression.

```
int main()
{
  int (*f)(int b);

  f = sum_c(3);

  printf("3 + 2 = %d\n", f(2));

  return 0;
}
```

Figure 20: Using function `sum2_c()` (Figure 19).

```
int main()
{
  int (*f1)(int b);
  int (*f2)(int b);

  f1 = sum_c(3);
  f2 = sum_c(4);

  printf("3 + 2 = %d\n", f1(2));
  printf("4 + 2 = %d\n", f2(2));

  return 0
}
```

Figure 21: Issue with the "`sum_c()`" function of Figur3 19.

Notice that from a computer scientist's point of view the currying technique can requires the usage of functions that return functions (and not "simple" function pointers) as a return value. This fact makes it impossible, for example, to use currying in programming languages such as C. To understand this, consider the "`sum()`" function shown in Figure 18 and try to generate its "curryified form". Such a function should receive an integer `a` as input and generate as a result a function that given an integer `b` adds `a` to `b`. Using a small extension to the C language implemented by gcc, which allows you to nest function definitions, you could think about coding this function as in Figure 19. Ignoring the strange syntax of the C language that makes the code difficult to read/understand (and the fact that a definition of `s()` is nested inside the definition of `sum_c()`, which is not allowed by the standard C language), this code implements a function `sum_c` which takes a formal parameter `a` of type `int` and returns a pointer to a function that takes a formal parameter of type `int` and returns a value of type `int`. The function is then usable as shown in Figure 20, where `f` is a variable of type "pointer to function from integer to integer".

Although gcc can compile this code (which even appears to work correctly in some cases), there is a major conceptual error: `sum_c()` returns a *pointer* to the `s()` function which uses a non-local variable `a` (here, "non-local" refers to the environment of "`s()`"). This variable is the actual parameter of `sum_c()`, which is stored on the stack during the `sum_c()` lifetime... But it is removed when from the stack `sum_c()` returns! The invocation `f(2)` will therefore access some memory that has been freed (and potentially re-used), generating undefined behavior. Although some simple tests may work correctly, the example of Figure 21 will show all the limits of the previously proposed solution.

The problem can only be solved by making the `sum_c()` function return a *real function* (also including its non-local environment!) and not simply a function pointer. Technically, this solution can be implemented using a *closure*, i.e. an "(environment, pointer to function)" pair where the environment will contain the binding between the name " tt a" and a variable which is not allocated on the stack, but in some other memory structure. The typical solution is to allocate such variables in the heap; this clearly creates the risk of memory leaks (which are not encountered with the "traditional" allocation of activation records on the stack) and makes it necessary to implement a garbage collector (when the

```
int->int sum_c(int a)
{
  int s(int b) {
    return a + b;
  }

  return s;
}
```

Figure 22: Correct Currying of the `sum()` function (Figur2 18) using a pseudo-language similar to C.

```
double compute_derivative(double (*f)(double x), double x)
{
  const double delta = 0.001;

  return (f(x) - f(x - delta)) / delta;
}
```

Figure 23: Computation of the value of the derivative of function `f()` in a point `x`, in C.

```
double->double derivative(double f(double x))
{
  double f1(double x)
  {
    const double delta = 0.001;

    return (f(x) - f(x - delta)) / delta;
  }

  return f1;
}
```

Figure 24: Computing the derivative of a function.

closure is no longer used, the activation record allocated on the heap can be deallocated). On the other hand, we have already seen how the functional programming paradigm requires to use garbage collectors.

Using a syntax like "`<type1> -> <type2>`" to represent the type of a function that takes an argument of type "`<type1>`" and returns a result of type " `<type2>`", the correct solution to the above problem (encode the "currified form" of the "`sum()`" function of Figure 18) is shown in Figure 22. In general, a function the curryified form of a function"*type3* `f(`*type1* `a,` *type2* `b)`" is " em type2`->`*type3* `fc(`*type1*`a)`", with `f(a,b) = (fc(a))(b)` (mathematically, $f(a,b) \rightarrow f'(a) = f_a : f_a(b) = f(a,b)$).

In summary, multi-argument functions and single-argument higher-order functions have the same expressive power; many functional programming languages (providing higher order functions) are limited to only one formal argument/parameter per function. ML and Haskell adopt this approach: even if there is a simplified syntax that allows defining functions as if they had multiple arguments (for example, using the `fun` keyword in Standard ML) this is then converted by the abstract machine into the definition of the "curryified" function. For example, in Standard ML **fun** f p1 p2 p3 ... = exp; is equivalent to **val rec** f = **fn** p1 => **fn** p2 => **fn** p3 . . . => exp;.

Maybe the concept of currying can be better understood by looking at the following example. Consider the implementation of a `compute_derivative()` function which (as the name suggests) computes an approximation of the derivative of a given function at a specified point. The function therefore receives as arguments a function $f : \mathcal{R} \rightarrow \mathcal{R}$ (or, a pointer to it) and a real value $x \in \mathcal{R}$ in which to calculate the derivative of f, returning an approximation $d \in \mathcal{R}$ of the value of the derivative of f in point x. A simple implementation of `compute_derivative()` in the C programming language is shown in Figure 23.

Now, try to implement a `derivative()` function which returns the derivative function instead of calculating its value in a point x. The `derivative()` function thus receives a function $f : \mathcal{R} \rightarrow \mathcal{R}$ as

```
#include <iostream>
#include <functional>

double f(double x)
{
  return x * x + 2 * x + 1;
}

std::function<double (double)> derivative(std::function<double (double)> f)
{
  return [f](double x) {
    const double epsilon = 0.0001;

return (f(x + epsilon) - f(x)) / epsilon;
  };
}

int main()
{
  double x = 2;
  std::function<double (double)> f1;

  std::cout << "f'(" << x << ") = " << (derivative(f))(x) << std::endl;

  f1 = derivative(f);
  std::cout << "f'(" << x << ") = " << f1(x) << std::endl;

  return 0;
}
```

Figure 25: Derivative of a function in C++.

an argument and returns a function $f' : \mathcal{R} \to \mathcal{R}$. Since the return value must be a function (including its non-local environment) and not a simple function pointer, `derivative()` cannot be implemented in C (see previous example with `sum()`). Using a language similar to C but supporting higher order functions (with the syntax described previously), it can be implemented as shown in Figure 24. The smartest readers will surely have noticed that `derivative()` is nothing more than the "curryified" form of `compute_derivative()`[5]!

As another example, Figure 25 shows the implementation of the `derivative()` function in C++ (using functional extensions provided by C++11, such as lambda expressions). From the code, you can see some interesting things. First of all, the "`std::function`" class provided by the C++ language (starting from the C++11 standard) allows you to use a simpler and more intuitive syntax than that of function pointers of C. Furthermore, such a class does not simply store a function pointer, but a complete closure (composed of the function and its non-local environment); this closure will store (within the object of class "`std::function`") the values of "`f`". Finally, it should be noted that the "`[f](double x)`" syntax allows you to define an *anonymous function*, which is stored (together with its environment) in the value returned by "`derivative()`" (this construct is called "lamba function", for reasons that will become clear as you read in the next sections).

5 Functional Programming Languages

As discussed, the functional programming paradigm is characterized by the absence of the mutable variables (to eliminate the side effects associated with them), the consequent use of recursion instead of iteration and the fact that programs are composed of expressions and not of commands (which have side effects). An important consequence of the lack of side effects is the possibility to execute programs

[5] Trying to re-implementing the two functions with any functional language, this becomes even more evident

using some replacement / reduction mechanism instead of modifying the state of the abstract machine. Moreover, expressions and functions are expressible and storable entities, leading to another interesting feature of functional programming: the presence of higher order functions (functions that can operate on other functions, receiving functions as arguments and generating functions as results). Higher-order functions become even necessary if the number of possible arguments for a function is limited to one (and the currying technique is used to implement multi-argument functions).

Although this programming style can also be used with "more traditional" languages, some programming languages, called *functional programming languages*,that try to favor (or even force) its use. The fundamental characteristic of this class of programming languages is therefore the attempt to minimize side effects: although some functional programming languages include the concept of mutable variable, they can also be used without making use of this construct; some functional languages (such as Haskell), then, do not really foresee the existence of mutable variables. Such languages are called *pure functional programming languages*. A fundamental feature of functional languages is the possibility of treating code and data in a homogeneous way (in addition to the traditional data types there is the "function type", there are higher order functions, etc...).

The specific syntax and/or semantics of the various functional programming languages are not discussed here, but for these details, please refer to specific documents.

In functional languages, therefore, there is an environment that contains bindings between names and values (of scalar, structured, or function types). Some bindings are created when a function is invoked (binding between formal parameter and expression passed as actual parameter), but it is often useful (although not strictly necessary) to also have an environment which is not local to any function. Hence, every high-level functional programming language provides some way to map names to values in a global environment. A functional language thus provides:

- A *type system*, which is a set of predefined types, a set of operators to build valid expressions with values of such types, and a set of rules to assign a type to each valid value and to verify if an expression si correctly typed;
- Some way to define functions: an *abstraction* mechanism that given an expression abstracts it with respect to the value of a formal parameter. This mechanism is often implemented as an operator returing a (anonymous) function as a result;
- A "function application" mechanism, used to build valid expressions based on existing values, operators, and on functions defined through abstraction;
- Some mechanism to associate names to values in a global environment (`define` in scheme, `val` in Standard ML, "`=` " in Haskell, etc...)
- Depending on the type system used by the language, some mechanisms can be provided to define new data types based on the existing ones.

The mechanisms and the syntax used by a programming language to implement these items result in different functional programming languages, with different features and abstractions. For example, the type system used by the language can vary, performing more or less strict checks at runtime (as for example in Lisp, scheme and similar languages) or at build time (as for example in Haskell, Standard ML, but also C++). More dynamic (and less "strict") type systems increase the possibility of programming errors (by failing to identify some class of type errors) and introduce overhead (performing some checks at runtime instead of build time). On the other hand, languages with dynamic and relaxed type systems look more flexible and powerful (for example, implementing a Y combinator in Lisp or scheme is much more easier than implementing it in Haskell, Standard ML or C++).

As far as defining functions is concerned, some readers will probably be more accustomed to the approach followed by many imperative programming languages, which generally provide a single mechanism that simultaneously specifies the function body and creates a link in the environment between the body of the function and its name. However, the most common functional programming languages make a distinction between two different mechanisms: abstraction, which generates a function-type value without assigning it a name (a so-called "anonymous function" — consider as an example lambda expressions in C++) and a second mechanism which allows to modify the environment (even the global environment) by associating a name to a generic value (which can be a function or something else). As an important consequence, *when the body of a function is defined this function is not yet associated with a name.* This can clearly create some issues when trying to define a recursive function, as we will see better in the future when talking about λ calculus.

Finally, a fundamental aspect of functional programming languages is (clearly) function invocation. Although it may seem simple and natural to us, it deserves a minimum of discussion: for example, looking at the definition of a "`fact(unsigned int n)`" function such as "$(n == 0)\ ?\ 1 : n * fact(n - 1)$", it is natural to think that "$fact(4)$" is evaluated as

$(4 == 0)\ ?\ 1 : 4 * fact(4 - 1) \rightarrow 4 * fact(3) \rightarrow ...$

by immediately computing $4 - 1 = 3$. However, this is not the only possible way to evaluate the function: a valid alternative could be

$(4 == 0)\ ?\ 1 : 4 * fact(4 - 1) \rightarrow 4 * fact(4 - 1) \rightarrow$

$\rightarrow 4 * ((4 - 1 == 0)\ ?\ 1 : (4 - 1) * fact(4 - 1 - 1)) \rightarrow$

$\rightarrow 4 * ((4 - 1) * fattoriale(4 - 1 - 1)) \rightarrow ...$

by computing the results of the various arithmetic operations only when strictly needed (when the results are actually used).

A functional programming language must therefore clearly specify how and when to evaluate expressions. Different evaluation strategies are possible, and the most famous are:

- *eager* evaluation strategies: when a function `f` is applied to an expression `e` the expression is evaluated (reducing it to an irreducible value) before invoking the function
- *lazy* evaluation strategies: when a function `f` is applied to an expression `e` the function is called without first trying to reduce its actual argument (thus passing an unevaluated expression and not an irreducible value).

Note how the first strategy substantially coincides with passing parameters by value (each actual parameter is evaluated to an irreducible value before invoking the function), while the second strategy coincides with passing parameters by name (a *thunk* is passed as an actual argument to the function — remember that a thunk is an expression with no local variables, which is not evaluated before actually being used).

Generally, pure functional languages (like Haskell) tend to favor lazy evaluations, while non-pure functional languages (functional programming languages with side effects, like ML, Scheme, etc...) are forced to use eager evaluation strategies. To understand why, consider a language with mutable variables (and therefore "not too functional") and a function `bad_bad_function()` defined as

```
void bad_bad_function(void)
{
  x++;
}
```

where "`x`" is a global variable. If "`x`" is initialized to 0 when the program starts, what is the value stored into this variable after invoking some_function(bad_bad_function(), bad_bad_function())? If a lazy evaluation strategy is used, it is not possible to answer this question, because the answers depends on how many times "`some_function()`" evaluates its arguments. If an eager evaluation strategy is used, instead, the answer is "2", because "`bad_bad_function()`" is evaluated one time for each actual argument.

Although the previous example essentially concerns a non-problem (functional programming languages shouldn't implement mutable variables), I/O operations represent much more real and serious problems. In fact, any input or output operation is a side effect and in the presence of lazy evaluation risks to introduce non-determinisms in the behavior of the program (which would be the output of some_function(bad_bad_function(), bad_bad_function( )) if `bad_bad_function()` printed something on the screen?). Pure functional languages can address this problem by modeling I/O functions as functions that take an entity "world" as input and output a modified version of that "world". Or can introduce I/O functions that return descriptions of "I/O operations" to be performed by a non-purely-functional engine, without using lazy evaluation (I/O functions are lazily evaluated, but I/O operations are not). Languages based on lazy evaluation then provide various types of mechanisms for serializing the execution of I/O operations, in order to make the program's interactions with the outside world deterministic. As an example, the serialization of the execution of two functions `f1()` and `f2()` is implemented by making `f2()` be invoked by receiving the output of tt f1() (the "world" entity, for example). Since handling I/O in this waycan lead to complex and unintuitive notations, some languages such as Haskell provide a simplified syntax for these mechanisms, inspired to the syntax of imperative languages (to do this, Haskell uses complex mathematical tools such as category theory monads).

The practical effects of using different evaluation strategies are visible, for example, trying to implement the Y combinator (an implementation of Y in Haskell is possible, while an implementation in

Standard ML or Scheme will result in infinite recursion and will require to implement a different combinator, such as Z).

It can be proved that if both lazy and eager evaluation mechanisms reduce an expression to a value, then the value obtained by lazy evaluation and the one obtained by eager evaluation are the same (for this, see the Church-Rosser theorem). Furthermore, if lazy evaluation leads to infinite recursion then eager evaluation also leads to infinite recursion (but on the other hand there are situations where eager evaluation leads to infinite recursion while lazy evaluation allows to reduce the expression to a value - again, see Y combinator).

6 Computation as Reduction

Based on the mechanisms just described, a program written according to the functional programming paradigm can be "executed" through reduction, implemented by repeating 2 operations:

- Search for names in the environment (and text replacement of a name with the corresponding functional value - represented as an abstraction)
- Application of functions (text replacement of formal parameter with actual parameter)

A functional program is therefore represented as a set of definitions and environment modification operations (creation of bindings) which may require the evaluation of expressions to be processed. The computation of this program will be carried out by the abstract machine through a series of text replacements / reductions which will lead to simplification until we arrive at simple forms that cannot be further reduced (called *values*).

To simplify the usage of functional programming techniques, other constructs are often provided (although they are not strictly necessary). Examples are:

- A way to modify the environment (usually, the `let` construct);
- A mechanism similar to the fixed point operator `fix`, which allows you to define recursive functions;
- Some constructs which constitute "syntactic sugar" to define multi-argument functions (hiding the use of currying), etc...

Regarding the construct (usually called `let`) used to modify the environment in which an expression is evaluated, note that this construct is not strictly necessary because it can be implemented via function call and parameter passing. For example, consider a generic construct "`let` *x* `=` *e1* `in` *e2*" (where "*x* " is a generic name while "*e1*" and "*e2*" are two expressions) which binds the name "*x*" to the expression ' '*e1*" when evaluating "*e2*". This can be implemented by defining a function `f()` with formal parameter "*x*" and body "*e2*" and invoking this function with actual parameter "*e1*"[6]. Even better, you can use an anonymous function: using Standard ML syntax "`let` *x* `=` *e1* `in` *e2*" becomes "(**fn** x => e2) e1".

The construct equivalent to the fixed point operator `fix` is instead very useful for simplifying the definition of recursive functions: as previously mentioned (and as it will become clearer by studying the λ calculus), without this mechanism the definition of recursive functions would not be simple: the name of a function cannot be used in its definition, because it is not yet bound to any value in the global environment. To define recursive functions it would be necessary to implement a fixed point combinator (like Y or Z) and apply this operator to the "closed version" of the function to be defined. This mechanism (called `val rec` or `fun` in Standard ML, `letrec` in Scheme, etc...) instead allows you to use recursion directly.

In languages that use more powerful type systems a *pattern matching* mechanism is often provided to manipulate values of user-defined types (for example, distinguishing the various variants of a type, etc...).

7 A Minimal Functional Language

To conclude, the most curious readers might wonder how the simplest possible functional programming language looks like. Such a minimal language does not contain "high-level" features which are useful for

[6]This trick of replacing a variable with a formal parameter and its value with the actual parameter is often used to reimplement imperative code using the functional paradigm.

making the code more readable, but are not strictly necessary. In other words, what can be obtained by removing from a functional programming language all the features that are not essential for Turing-completeness, such as

- the presence of a global environment (which, as mentioned, simplifies the definition of recursive functions, but is not strictly necessary)
- a strong type system with "strict typing" and more complex data types (which increase the readability of the code but are not essential: notice how even in the imperative programming paradigm the Assembly language only uses binary values and does not define any other data type)
- the various constructs that represent "syntactic sugar".

What remains is a language in which programs are (pure!) expressions composed of:

1. variable names (irreducible terms)
2. function definitions (abstraction)
3. applications of functions

For names, the simplest and most minimal choice is to use single lowercase letters, even if sometimes identifiers composed of several characters are used.

Regarding function applications, programmers who are familiar with languages of the C family (C, C++, Java, ...) use "`f(x)`" to represent the application of the "`f`" function to the "`x`" actual parameter (remember that for simplicity we can only consider functions with one argument). The parentheses around the actual parameter are useless though, so you might as well use the "`f x`" syntax, which is often preferred. If function application is left-associative, the composition $g \circ f$ of the functions f and g can then be represented as "`g (f x)`" instead of "`g(f(x))`". The syntax "`g f x` is instead equivalent to "`(g(f))(x)`" (and this, as you will see, makes the currying syntax more natural). Some languages of the LISP family instead expect parentheses around the function application instead of around the actual parameter ("`(f x)`" instead of "`f(x)`"); in this case, $g \circ f$ becomes "`(g (f x))`".

Finally, the language must provide some mechanism for constructing expressions that are evaluated to functions (allowing to somehow "define" a function starting from an expression "`e`" and a formal parameter "`x`"). Regardless of the syntax that the language uses, this construct *abstracts* the expression "`e`" from the specific value of the formal parameter "`x`"; must therefore contain some language-specific keyword (which can be the Greek letter λ, the symbol "\", the word "`fn`", a sequence of square, parentheses and curly brackets, or other), the name of the formal parameter and the expression to be abstracted. Examples could be "$\lambda x.e$", "`\x -> e`", "`fn x => e`", "`[]( auto x) { e }`", "`(lambda (x) (e))`" or similar...

Not having strict typing, this "minimal language" knows only generic "functions" that operate on expressions (they take another generic function as an argument and generate a generic function as a result), without precisely specifiying the domain and codomain of the functions (these sets coincide with the set of expressions that make up the language). The language resulting when the "$\lambda x.e$" syntax is used for abstraction is name λ *calculus*. Surprisingly enough, such a language is still Turing complete: it is possible to encode natural numbers, booleans and other values using only functions, and it is possible to use various types of fixed point combinators to implement recursive functions even if no global environment is provided by the language. For this reason, λ calculus is often considered as a kind of "Assembly of functional programming languages".