Short Introduction to Functional Programming

Luca Abeni

December 30, 2022

Contents

Chapter 1

For Imperative Programmers

1.1 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 1.1 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

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

and this equation can result in a new implementation of the gcd function, as shown in Figure 1.2. 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 1.3.

Comparing Figure 1.1 (called "imperative implementation") with Figure 1.3 (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)

```
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 1.1: Euclide's algorithm implemented in C.

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

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

Figure 1.2: Euclide's algorithm implemented through recursion.

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

Figure 1.3: 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 1.4: 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 1.4: 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 $(3+1)*(2+5) = 28$.

1.2 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 1.5. A more functional version (because it uses only expressions, and not commands) of the factorial is instead shown in Figure 1.6

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 1.5: Recursive implementation of the factorial function.

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

Figure 1.6: 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) → (4 == 0) ? 1 : 4 ∗ fact(4 − 1) → 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) → 4 ∗ ((3 == 0) ? 1 : 3 ∗ fact(3 − 1)) → 4 ∗ (3 ∗ fact(2)).

Repeating this process again, we get: 4 ∗ (3 ∗ fact(2)) → 4 ∗ (3 ∗ ((2 == 0) ? 1 : 2 ∗ fact(2 − 1))) →

→ 4 ∗ (3 ∗ (2 ∗ fact(1))) → 4 ∗ (3 ∗ (2 ∗ ((1 == 0) ? 1 : 1 ∗ fact(1 − 1)))) →
→ 4 ∗ (3 ∗ (2 ∗ (1 ∗ fact(0)))) →
→ 4 ∗ (3 ∗ (2 ∗ (1 ∗ ((0 == 0) ? 1 : 0 ∗ fact(0 − 1))))) →
→ 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 1.7 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 1.8.

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 1.9.

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)

```
#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 1.7: Solution to the problem of the Towers of Hanoi.

```
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 1.8: 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 1.9: 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 1.9 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 1.10 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.

1.3 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 1.11.

Since "`fact(3)`" will also be recursively invoked, the stack evolves as shown in Figure 1.12, 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 1.10: Functional solution to the “Towers of Hanoi” problem written in C++.

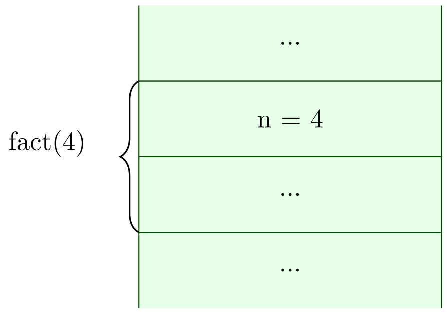

Figure 1.11: Stack frame for the invocation `fact(4)`.

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

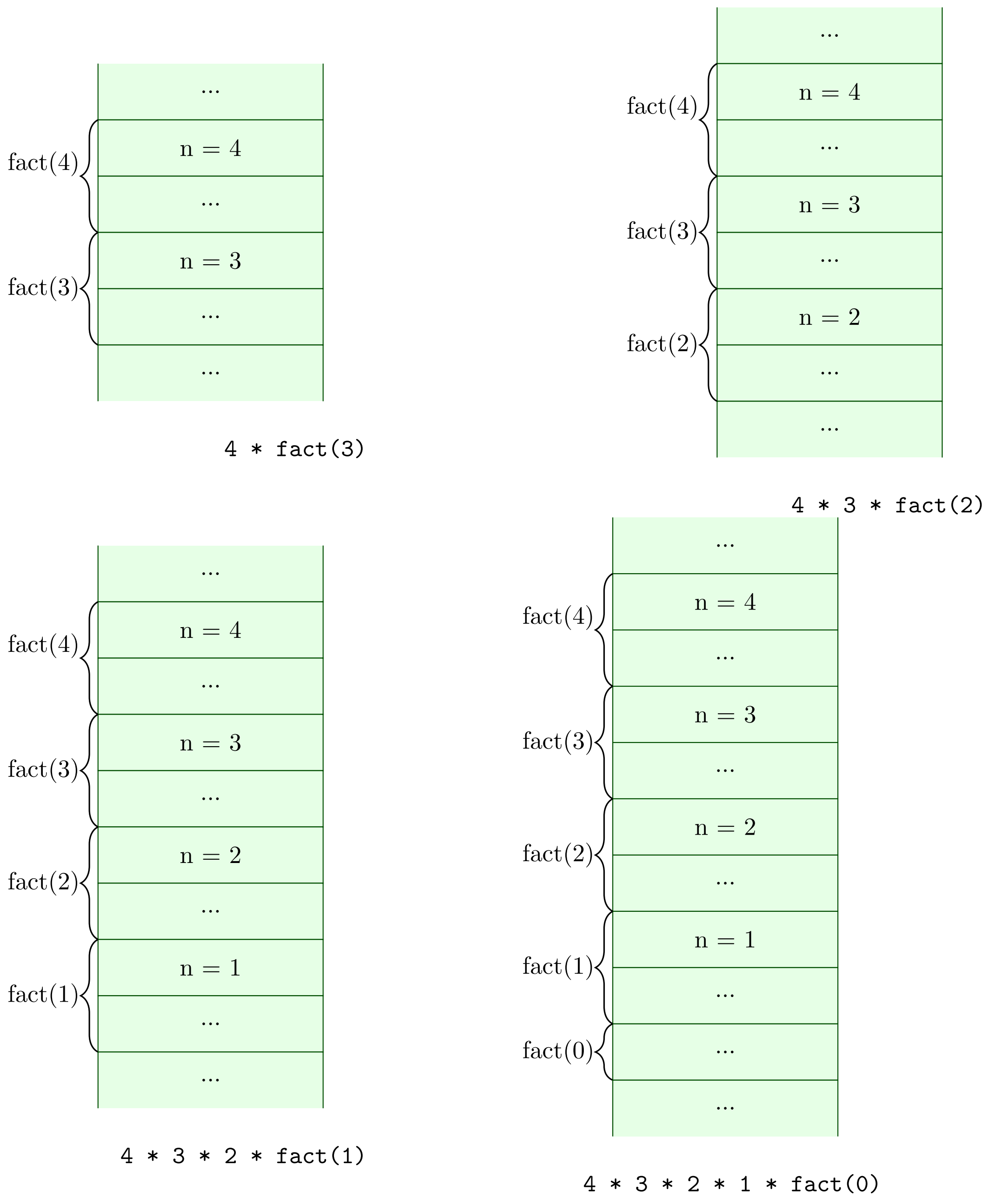

Figure 1.12: Evolution of the stack when `fact(4)` is invoked.

```
/* 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 1.13: Example of mutual recursion to test if a number is even or odd.

```
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 1.14: 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 1.13, 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 1.14. 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:
fact(4) → fact_tr (4, 1) → (4 == 0) ? 1 : fact_tr (4 − 1, 4 ∗ 1) →
→ fact_tr (3, 4) → (3 == 0) ? 4 : fact_tr (3 − 1, 3 ∗ 4) →
→ fact_tr (2, 12) → (2 == 0) ? 12 : fact_tr (2 − 1, 2 ∗ 12) →
→ fact_tr (1, 24) → (1 == 0) ? 24 : fact_tr (1 − 1, 1 ∗ 24) →
→ fact_tr (0, 24) → (0 == 0) ? 24 : fact_tr (0 − 1, 0 ∗ 24) → 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 "

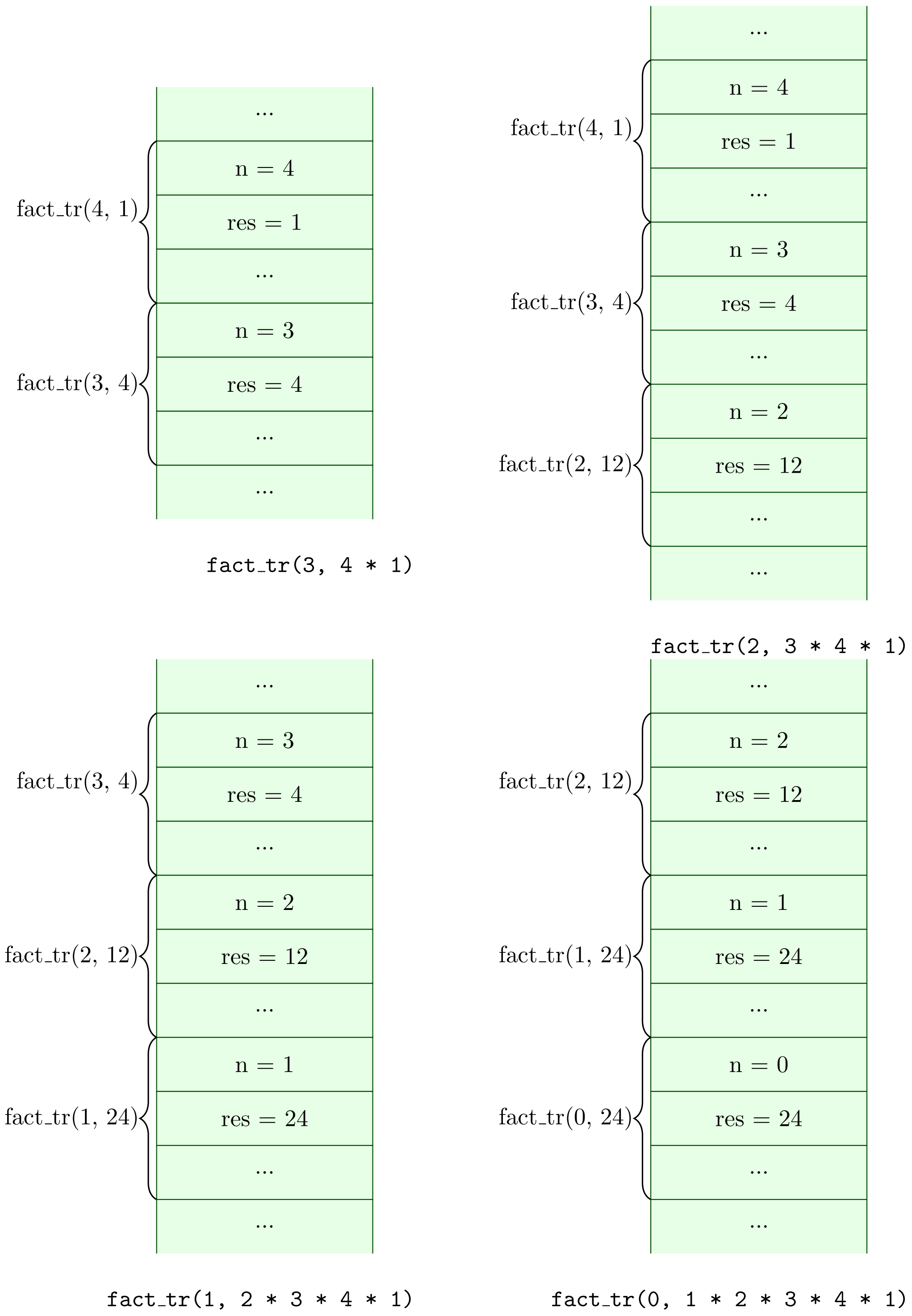

Figure 1.15: Evolution of the stack for the invocation of`fact(4)`.

```
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 1.16: 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 1.17: 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 1.15 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 1.14. This code is shown in Figure 1.16.

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 1.18: C function summing 2 integers.

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

  return s;
}
```

Figure 1.19: Attempt at curryifying the `sum2()` function (Figur2 1.18) 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 1.17.

1.4 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 1.20: Using function `sum2_c()` (Figure 1.19).

```
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 1.21: Issue with the "`sum_c()`" function of Figur3 1.19.

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 1.18 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 1.19. 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 1.20, 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 1.21 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 1.22: Correct Currying of the `sum()` function (Figur2 1.18) 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 1.23: 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 1.24: 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 1.18) is shown in Figure 1.22. 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, $f(a,b) \to f'(a) = f_a : f_a(b) = f(a,b)$).

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 $f : \mathcal{R} \to \mathcal{R}$ (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 1.23.

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 $f : \mathcal{R} \to \mathcal{R}$ 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 1.25: 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 1.24. The smartest readers will surely have noticed that `derivative()` is nothing more than the "curryified" form of `compute_derivative()`[5]!

As another example, Figure 1.25 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).

1.5 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) → 4 ∗ fact(3) → ...

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) → 4 ∗ fact(4 − 1) →
→ 4 ∗ ((4 − 1 == 0) ? 1 : (4 − 1) ∗ fact(4 − 1 − 1)) →
→ 4 ∗ ((4 − 1) ∗ fattoriale(4 − 1 − 1)) → ...

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).

1.6 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...).

1.7 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".

Chapter 2

First Meeting with the Lambda Calculus

The λ calculus is a formalism (or, if we prefer to see it from a CS point of view, a programming language) which allows us to define the fundamental concepts of functional programming: functions, definition of functions and application of functions.

If we see λ calculus as a programming language, we can notice how it introduces the basic mechanisms needed to write functional programs without introducing the abstractions that characterize higher-level functional programming languages. Thus, λ calculus can be seen as the FP version of the Assembly language. It is interesting, however, to note that even lower-level functional languages exist, because they introduce even fewer abstractions (for example, we will see that the function definition construct can be omitted).

From a different point of view, the λ calculus can be seen as the theoretical foundation for functional programming, as it can be proved to be turing-complete. This result has a remarkable importance, because it means that the functional programming paradigm allows to implement any computable algorithm (ie: it has the same expressive power as the imperative programming paradigm).

Summing up, the basic elements of λ calculus are just names, the concept of abstraction (definition of functions), and function application. Therefore, higher-level concepts such as data types, global environment, and the like do not exist. Since there are no different types of data, the basic elements of the λ calculation are generic "functions", which receive another function as an argument and generate a function as a result. The domain and codomain of these functions are generic expressions (better, λ-expressions) and are not explicitly specified.

We will see that there is a typed version of the λ calculus, in which the type of a function is specified by the function's domain and codomain (as traditionally done in the various analysis or algebra courses). Paradoxically, however, this formalism loses the expressive power of the original λ calculus and is no longer Turing-complete.

The basic idea of λ calculus is to express algorithms (or to code programs) as expressions, called λ-expressions in the following. As we will see, the execution of a program then consists in evaluation of a λ-expression (using a simplification mechanism called "reduction"). So let's see how λ-expressions are composed The incredibly simple syntax reflects the fact that λ-expressions are built starting from the three simple concepts already mentioned:

1. Variables (which actually represent functions). They constitute the terminal elements of the language and are indicated by names (identifiers), which will be represented by single letters written in *italic* (for example, "x", "y" or "f") in the following

2. Abstractions, which allow you to specify that a variable "x" is an argument in the following expression. Technically, an abstraction is said to "bind" a variable into an expression (the reason for this terminology will become clear later). In practice, an abstraction "creates" (informally speaking) a function starting from a λ expression, specifying the argument of the function (the bound variable)

3. Functions applications. They represent the inverse operation of abstraction and allow to transform an abstraction and a λ expression into a single λ expression by removing the bound variable (actually, the whole abstraction is removed)

Using BNF notation, the syntax of a λ-expression is:

```
<expression> ::= <name>                           ; lowercase letter in italic
               | ( λ<name>.<expression>)          ; abstraction
               | (<expression> <expression>)      ; function application
```

This is equivalent to the following inductive definition:

- A name / variable / function (indicated with a single italic letter below) is a λ expression
- If e is a λ expression and x is a name, then $(\lambda x.e)$ is a λ expression
- If e_1 and e_2 are λ expressions, then $(e_1 e_2)$ is a λ expression

Based on the definitions above, the following are examples of λ expressions:

- x (variable / function)
- $(\lambda x.(xy))$ (abstraction: bind variable "x" in the expression "xy", transforming that expression into a function of variable "x")
- $((xy)z)$ (application: apply function "x" to "y", then apply the result to "z")
- $(\lambda x.(xy))z$ (more complex expression)

According to what explained so far, any abstraction or application of a function should be enclosed in parentheses (to make the syntax less ambiguous); actually, the following conventions are assumed to reduce the number of parentheses used:

- Function application is left-associative: $((xy)z)$ is therefore equivalent to xyz
- The "λ" operator is right-associative and has lower precedence than function application: $(\lambda x.(xy))$ is therefore equivalent to $\lambda x.xy$

Another convention often used in the literature is that the variables linked by several immediately successive abstractions are grouped together; for example, $\lambda x.\lambda y.f$ can be written as $\lambda xy.f$. However, this convention will not be used in the following and we will keep only one variable for λ.

It is interesting to notice how the syntax of the λ-calculus allows to distinguish the definition of a function from its application: in the commonly used mathematical notation, the term "$f(x)$" is used both to indicate that the function "$f()$" is applied to the value "x" and to define this function (as in "$f(x) = x^2$"). In the λ-calculus, however, "fx" represents the application of "f" to "x", while "$\lambda x.f$" represents the definition of a function with argument "x".

Another interesting thing to note is that given the absence of a global environment it is not possible to *dynamically create* associations between λ expressions and non-local names. In other words, not only the λ-calculus has no concept of "assignment" of values to a mutable variable, but it also lacks the equivalent of variable declaration (not even immutable variables). For convenience it is possible to use symbolic names for complex λ expressions (see applied λ calculus, below), but these are macro-like, static definitions (not bindings in a global environment that may vary dynamically at runtime).

As a result, only anonymous expressions and *anonymous functions* can be created in the λ-calculus (similar to what was done in standard ML with the `fn` construct, in Haskell with "\" or in C++ with lambda expressions "`[](...){...}`"). This could imply that λ-calculus does not allow defining recursive functions (and consequently is not Turing complete); we will see later how it is instead possible to define functions that require recursion by using the concepts of *fixed point* and *fixed point combinator*.

The only (name,value) bindings that can be created dynamically are the *local* bindings between formal parameters and actual parameters that are created during function application. This means that at least the concept of a local environment exists in the λ-calculus.

2.1 Semantic of the Lambda Calculus

As previously mentioned, λ-calculus allows programs to be encoded as expressions, which are "executed" by evaluating them via a process called reduction. Informally speaking, we can say that this process is based on the meanings that have been associated with the various basic elements of a λ expression. To define the semantics of the λ-calculus in a more formal way, it is first necessary to introduce some basic concepts, such as "free variables" and "bound variables".

Intuitively, a variable "x" is bound by a construct "$\lambda x.E$" (where E is a generic λ expression), while it is free in an expression "E" if in "E" there is no λx abstraction that binds "x". To give a more formal definition, we must refer to the recursive definition of λ expressions: in particular, if $\mathcal{B}(E)$ represents the set of bound variables in "E" and $\mathcal{F}(E)$ represents the set of free variables in " E", we can say that:

- For each variable "x", $\mathcal{F}(x) = \{x\}$ and $\mathcal{B}(x) = \emptyset$
- $\mathcal{F}(E_1E_2) = \mathcal{F}(E_1) \cup \mathcal{F}(E_2)$; $\mathcal{B}(E_1E_2) = \mathcal{B}(E_1) \cup \mathcal{B}(E_2)$
- $\mathcal{F}(\lambda x.E) = \mathcal{F}(E) - \{x\}$; $\mathcal{B}(\lambda x.E) = \mathcal{B}(E) \cup \{x\}$

Basically, this definition says that if an expression is composed of only one variable, that variable is free; composing two expressions (applying one expression to another) does not change the state of the variables (free variables remain free and bound variables remain bound) and the operator "$\lambda x.E$" binds the variable " x" in the expression "E" (it removes "x" from the set of free variables of "E" and adds it to the set of bound variables). The λ operator is said to bind variable "x" in "$\lambda x.E$" because when the expression "$\lambda x.E$" is applied to an expression $E1$ a binding between x and $E1$ is created in the local environment of E.

Based on this simple recursive definition it is possible to compute the set of free variables and bound variables for each λ expression. A λ expression which contains no free variables but is composed only of bound variables) is called a "combinator" and has the important property that the result of its evaluation only depends on the arguments (current parameters) used to evaluate it. More formally, a λ expression E is a combinator if $\mathcal{F}(E) = \emptyset$.

We can now define the concept of α equivalence between two λ expressions. Informally, two λ expressions E_1 and E_2 are α equivalent ($E_1 \equiv_\alpha E_2$) if they differ only in the parameters' names. This means that when defining a function the name of the function argument is not important (using a more familiar mathematical notation, $f_1(x) = x^2$ and $f_2(y) = y^2$ represent the same function); so, for example, $\lambda x.xy \equiv_\alpha \lambda z.zy$. The correct definition of α equivalence is obviously more complex, because, for example, $\lambda x.x\lambda x.xy$ is not α equivalent to $\lambda z.z.\lambda x.zy$ but is α equivalent to $\lambda z.z.\lambda x.xy$. Basically, $\lambda x.E$ is α equivalent to $\lambda z.E[x \rightarrow z]$, where $E[x \rightarrow z]$ represents the expression "E" with variable "x" is replaced by expression "z" only if it is free:

- If "x" and "y" are variables and E is a λ expression, $x[x \rightarrow E] = E$ and $y \neq x \Rightarrow y[x \rightarrow E] = y$
- Given two λ expressions E_1 and E_2, $(E_1E_2)[x \rightarrow E] = (E_1[x \rightarrow E]E_2[x \rightarrow E])$
- If "x" and "y" are variables and E is a λ expression,
  - $y \neq x \wedge y \notin \mathcal{F}(E') \Rightarrow (\lambda y.E)[x \rightarrow E'] = \lambda y.(E[x \rightarrow E'])$
  - $y = x \Rightarrow (\lambda y.E)[x \rightarrow z] = \lambda y.E$

Looking back at the previous example, we can see how the rule "$y = x \Rightarrow (\lambda y.E)[x \rightarrow z] = \lambda y.E$" allows us to obtain the correct result: $\lambda x.x\lambda x.xy \equiv_\alpha \lambda z.(x\lambda x.xy)[x \rightarrow z] = \lambda z.x[x \rightarrow z](\lambda x.xy)[x \rightarrow z] = \lambda x.z\lambda x.xy$ as expected. It is also interesting to note that the rule "$y \neq x \wedge y \notin \mathcal{F}(E') \Rightarrow (\lambda y.E)[x \rightarrow E'] = \lambda y.(E[x \rightarrow E'])$" contains the condition "$y \notin \mathcal{F}(E')$": this condition is needed to avoid wrong substitutions like $(\lambda x.xy)[y \rightarrow x] = \lambda x.xx$ which would lead to α equivalences like $\lambda y.\lambda x.xy \equiv_\alpha \lambda x.\lambda x.xx$, clearly incorrect. This phenomenon, in which a free variable "y" in "$\lambda x.xy$" becomes bound after a substitution is called *variable capture* (because a simple substitution transforms a variable free in a bound variable) and should be avoided during substitutions. The substitution mechanism $E[x \rightarrow y]$ defined above is then called *capture-avoiding substitution* and can be used to formally define the α equivalence relation:

$$\lambda x.E \equiv_\alpha \lambda y.E[x \rightarrow y]$$

As suggested by the name, α equivalence is an equivalence relation: $E_1 \equiv_\alpha E_2$ is therefore a *symmetric*, *reflexive* and *transitive* relation between λ expressions:

- $E \equiv_\alpha E$
- $E_1 \equiv_\alpha E_2 \Rightarrow E_2 \equiv_\alpha E_1$
- $E_1 \equiv_\alpha E_2 \wedge E_2 \equiv_\alpha E_3 \Rightarrow E_1 \equiv_\alpha E_3$

Capute-avoiding substitutions play a fundamental role in the λ-calculus , as they are used in the reduction mechanism to simplify λ expressions as well as for α equivalences. Informally speaking, reducing a λ expression consists in applying functions (removing abstractions) as in $(\lambda x.xy)z \rightarrow zy$. This procedure may appear simple, but it hides a series of complications; for example, the reduction $(\lambda x.(x\lambda y.xy))y \rightarrow y\lambda y.yy$ is clearly wrong, because the "y" variable is bound in the process (this is one of the reasons why the capture-avoiding substitution mechanism was defined earlier!). Thus, whenever you have an abstraction ("$\lambda x.E$") applied to an expression E_1, you can use a **capture-avoiding** substitution of E_1 in E (replacing x) to reduce the λ expression eliminating the abstraction.

More formally, a *redex* (*red*ucible *ex*pression) is defined as a λ expression of the type $(\lambda x.E)E_1$ and $E[x \rightarrow E_1]$ is defined as is its reduced. Based on this, the β reduction "$\rightarrow_\beta$" can be defined as the replacement of a redex by its reduction:

$$(\lambda x.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

It might appear that some redexes cannot be reduced because capture-avoiding reduction cannot be used (for example, a variable "y" is bound in "E" and "y" appears among the free variables of E_1 - in this case, a simple replacement would capture the free "y"). Consider the λ expression $(\lambda y.\ lambdax.xy)(xz)$: this expression clearly represents a redex, so one could try to reduce it using the β reduction mechanism, which would lead to $(\lambda x.xy)[y \rightarrow (xz)]$. Note however that $x \in \mathcal{F}(xz)$, so none of the rules presented in the definition of the capture-free substitution mechanism can be used (again: $(\lambda x.xy)[y \rightarrow (xz)] = \lambda x.x(xy)$ is not a capture-avoiding substitution, because it would capture the red " x"). How can this kind of redex be reduced, then? The concept of α equivalence comes to our aid, allowing us to rename the variables which are bound in E so that they do not appear among the free variables of E_1. In the previous example:

$$(\lambda y.\lambda x.xy)(xz) \equiv_\alpha (\lambda y.\lambda k.ky)(xz) \rightarrow_\beta (\lambda k.ky)[y \rightarrow (xz)] = \lambda k.k(xz)$$

this time the reduction does not capture any free variable, and is correct (the capture-avoiding substitution $(\lambda k.ky)[y \rightarrow (xz)]$ is now possible because $k \notin \mathcal{F}(xz)$).

The β reduction is not an equivalence relation, as it does not have the reflexive property: $E_1 \rightarrow_\beta E_2$ does not imply $E_2 \rightarrow_\beta E_1$. However, an equivalence relation (called β equivalence "$\equiv_\beta$") can be created by computing the β reduction reflexive and transitive closure. In practice, $E_1 \equiv_\beta E_2$ means that there is some chain of β reductions which "connect" E_1 and E_2 (β reducing E_1 and E_2 multiple times it is possible to arrive at the same expression E).

More formally, the β equivalence $\equiv_\beta$ is defined as:

- $E_1 \rightarrow_\beta E_2 \Rightarrow E_1 \equiv_\beta E_2$
- $\forall E, E \equiv_\beta E$
- $\forall E_1, E_2 : E_1 \equiv_\beta E_2, E_2 \equiv_\beta E_1$
- $E_1 \equiv_\beta E_2 \wedge E_2 \equiv_\beta E_3 \Rightarrow E_1 \equiv_\beta E_3$

Finally, it is interesting to note that a generic λ expression usually contains multiple redexes and the rules of the λ computation do not define an order for applying the possible β reductions. In this case it is possible to reduce the expression following any order for the β reductions (provided that the parentheses and the rules of associativity and precedence between operators are respected).

The order in which to evaluate the various redexes is decided by defining an evaluation strategy in addition to the reduction rules of the λ-calculus (for example, proceed to the right starting from the leftmost redex, or start from the "innermost" redex, etc...). The various evaluation strategies (lazy vs eager, by name vs by value, etc...) used by higher level programming languages derive from this evaluation strategy.

An important theorem (the Church-Rosser Theorem) proves that if a λ expression E can be reduced to E_1 by 0 or more β reductions and E can be reduced to $E_2 \neq E_1$ by 0 or more β reductions, then there exists E_3 such that both E_1 and E_2 can be reduced to E_3 by 0 or more β reductions ($E \rightarrow_\beta ... \rightarrow_\beta E_1 \wedge E \rightarrow_\beta ... \rightarrow_\beta E_2 \Rightarrow \exists E_3 : E_1 \rightarrow_\beta ... \rightarrow_\beta E_3 \wedge E_2 \rightarrow_\beta ... \rightarrow_\beta E_3$).

An important corollary of this theorem is that if E is reducible to a normal form (λ expression which no longer contains any redex), then this normal form does not depend on the order of β reductions. In other words, every λ expression E has at most 1 normal form. Note the use of the term "*at most*", as there are λ expressions which cannot be reduced to a normal form (the reduction process never ends). A typical example is the combinator $\Omega = \omega\omega$, where $\omega = \lambda x.xx$:

$$\Omega = \omega\omega = (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (xx)[x \rightarrow (\lambda x.xx)] = (\lambda x.xx)(\lambda x.xx) = \omega\omega = \Omega$$

therefore, $\Omega \rightarrow_\beta \Omega$!!! This is the equivalent of an infinite loop in an imperative language, or an infinite recursion in a functional language. The existence of this type of expression (which generates a non-terminating sequence of reductions) is necessary for the Turing-completeness of the λ-calculus (the Turing machine allows to encode infinite computations; if the λ calculus did not allow to encode infinite reductions, then it could not implement such Turing machine programs).

2.2 Encoding High-Level Languages

After seeing the most important definitions of the λ-calculus and the details of the reduction mechanism, it is quite difficult to understand how such a simple and seemingly inexpressive formalism can be Turing complete. In fact, it might seem that the λ computation could only be useful for manipulating functions or the like.

Making a parallelism with imperative programming, we can remember that programs written in a high-level languages are transformed into Assembly (by a compiler or an interpreter) to be executed by a physical CPU. Just as the λ-calculus allows you to work only with functions (and to perform relatively simple reduction operations on expressions composed only of functions, abstractions and applications), the Assembly language also allows you to operate only on binary numbers (stored in CPU registers or RAM) and has no concept of datatypes or global environment. Yet we have no problem in thinking that a program written in a high-level language with a global environment and strictly typed variables is converted to Assembly: we "just" need to implement all the high-level concepts based on Assembly instructions that operate on registers or memory. Similarly, the same high-level concepts can be implemented by using just functions, abstractions, and function applications.

In general, the various high-level abstractions we will be encoded by using combinators (which, as already said, are λ expressions in which no free variables appear). This is because the encoding must not depend on the context (hence, it must not refer to any symbol that is not a formal parameter / argument of the expression). As an example, some notable combinators known in the literature are:

- The combinator representing the identity function: $I = \lambda x.x$
- The combinator representing function composition: $B = \lambda f.\lambda g.\lambda x.f(gx)$
- $K = \lambda x.\lambda y.x$
- $S = \lambda f.\lambda g.\lambda x.fx(gx)$
- $\omega = \lambda x.xx$
- $\Omega = \omega\omega$
- $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

The importance of the Y combinator is fundamental for the λ-calculus, as it is a so-called *fixed point combinator* (this will be discussed later). The K and S combinators are interesting as they allow to define a subset of the λ calculus (called "SK calculus" or "SKI calculus") in which the abstraction operator λ is not explicitly used (!!!) (this will also be explained later) Before going on, note that the "=" symbol that has been informally used above to define the various combinators represents some kind of equality / equivalence (indicating, for example, that writing "I " is equivalent to writing "$\lambda x.x$") and is not a formally defined construct in the λ-calculus. Remember that the λ-calculus does not allow to modify some kind of non-local environment or to create bindings between names and symbols (again: in the λ calculation, there is no construct that allows you to create links between names and symbols at a global level!).

After these necessary premises, we can start to see how to implement the high-level constructs we are interested in using the λ calculus. The first thing to do is find a "λ encoding" for the natural numbers (and for the operations that can be performed on them). These encodings can then be used to encode integers (natural numbers with sign), rational numbers, real numbers, and so on.

Then, λ expressions will be used to also represent boolean values, basic logical operations and the so-called "arithmetic if" (which allows to evaluate an expression E_1 or an expression E_2 depending on the truth value of a boolean expression). Finally, more complex data structures can be implemented to show how higher-level data types can be represented by λ expressions.

Recalling that the λ-calculus is a functional formalism, it is quite clear that it will be necessary to use an inductive definition of the natural numbers, similar to Peano's:

- 0 is a natural number
- Given a natural number n, the successor of n (computable as $succ(n)$) is a natural number

The idea is therefore to use a λ expression to represent the natural number 0 and define a combinator which, applied to the representation of a natural number n, computes the representation of $n+1$. The *Church numerals* implement this idea:

- 0 is represented by the λ expression $\lambda f.\lambda x.x$
- The $succ()$ function (which calculates the successor of a natural number n) is represented by the λ expression $\lambda n.\lambda f.\lambda x.f(nfx)$

This encoding has the interesting property that the natural number $n \in N$ is represented by the function f applied n times to x: $n \equiv \lambda f.\lambda x.\overbrace{f(\ldots f}^{n}(x)\ldots)$ (to simplify the notation, the expression "$\overbrace{f(\ldots f}^{n}(x)\ldots)$" is sometimes written as "$f^n(x)$").

As an exercise, we can try to compute the representation of the natural number 1 as $1 = succ(0)$:

$$1 = succ(0) = (\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.x)$$

$$(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.x) \rightarrow_\beta (\lambda f.\lambda x.f((nf)x))[n \rightarrow (\lambda f.\lambda x.x)] = \lambda f.\lambda x.f(((\lambda f.\lambda x.x)f)x) \rightarrow_\beta$$

$$\rightarrow_\beta \lambda f.\lambda x.f(((\lambda x.x)[f \rightarrow f])x) = \lambda f.\lambda x.f((\lambda x.x)x) \rightarrow_\beta \lambda f.\lambda x.f((x)[x \rightarrow x]) = \lambda f.\lambda x.f(x)$$

Similarly, it is possible to compute the encoding of 2:

$$2 = succ(1) = (\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.f(x))$$

$$(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.f(x)) \rightarrow_\beta (\lambda f.\lambda x.f((nf)x))[n \rightarrow (\lambda f.\lambda x.f(x))] = \lambda f.\lambda x.f(((\lambda f.\lambda x.f(x))f)x) \rightarrow_\beta$$

$$\rightarrow_\beta \lambda f.\lambda x.f(((\lambda x.f(x))[f \rightarrow f])x) = \lambda f.\lambda x.f((\lambda x.f(x))x) \rightarrow_\beta \lambda f.\lambda x.f(f(x))$$

Without pretending to give rigorous proofs, let's try to better understand the encoding of $succ()$: informally speaking, $succ$ must transform $\lambda f.\lambda x.f^n(x)$ in $\lambda f.\lambda x.f(f^n(x))$. This can be done:

1. By somehow "removing" the two abstractions $\lambda f.\lambda.x.$ from the encoding of n
2. Then applying f to the expression obtained above
3. Then adding back the two abstractions $\lambda f.\lambda.x.$ removed in step 1
4. And finally abstracting everything from the number n

The first step (removal of the abstractions $\lambda f.\lambda.x.$) can easily be accomplished by applying the natural number encoding to f and x: in fact, $(\lambda f.\lambda x.f^n(x))f \rightarrow_\beta \lambda x.f^n(x)$ and $((\lambda f.\lambda x.f^n(x))f)x \rightarrow_\beta (\lambda x.f^n(x))x \rightarrow_\beta f^n(x)$. Thus, if the function n represents the encoding of a natural, then $(nf)x$ is an expression containing f applied n times to x. As mentioned (step 2), f must be applied to this expression once again, obtaining $f((nf)x)$; after step 3 we obtain $\lambda f.\lambda x.f((nf)x)$ and abstracting everything with respect to n (so that n is an argument of the combinator $succ$ and not a free variable) we get $\lambda n.\lambda f.\lambda x.f((nf)x)$ which is just the encoding of $succ$ presented above.

Based on the definitions of Church numerals, it is possible to define the encoding of the various operations on the natural numbers. For example, the sum can be encoded using a combinator which, applied to the encodings of two natural numbers n and m, generates the encoding of their sum. The expression of this combinator is $\lambda m.\lambda n.\lambda f.\lambda x.(mf)((nf)x)$ and can be obtained in this way:

1. First, we apply n to f and x to remove the abstractions $\lambda f.\lambda x.$, similar to what we did for encoding $succ$
2. Next, apply m to f to remove the abstraction $\lambda f.$
3. Next, applying the result of mf (that is, "m" without "λf") to the result of $((nf)x)$ (which is "$f^n x$"), we add m "$f($" to the left of "$f^n x$". The result is "$f^{n+m}x$"
4. As done for $succ$, we abstract again with respect to f and x to add the $\lambda f.\lambda x.$ removed in step 1

5. Finally, we abstract with respect to n and m, in order to obtain a combinator

It is then possible to define the encoding of the other operations on natural numbers, but the the details are omitted here for the sake of brevity. Coding a "*pred*" operator (which calculates the predecessor of a natural number) is possible, but not easy (and there are strange anecdotes involving Alonso Church - inventor of the λ calculus - one of his PhD students - who solved the *pred* encoding problem - and a barber). Without going into details, the encoding of this operator involves transforming the encoding of n into a pair containing the encoding of n and the encoding of $n-1$, and then taking the second element of the pair. The encoding of $(n, n-1)$ is generated from the encoding of n by starting from the encoding of $(0, 0)$ and iterating n times a function $\hat{f}$ which transforms the pair (n, m) into $(n+1, n)$. Now, remembering that the encoding of n is a combinator that applies n times its first argument to its second argument, it becomes clear that $(n, n-1)$ can be obtained applying n to $\hat{f}$ and then applying the resulting function to the encoding of $(0, 0)$. At this point, as mentioned, the encoding of $n-1$ can be obtained by applying to the result a function that returns the second element of a pair. As usual, everything must be abstracted from n. In light of this, it is therefore important to understand how to encode pairs using the *lambda*-calculus.

The "(a, b)" pair can be encoded as $\lambda z.zab$ and in general the function that generates the encoding of the "(a, b)" pair from "a" and "b"is $\lambda x.\lambda y.\lambda z.zxy$. Given the encoding of a pair, it is possible to obtain the first element using the function "first" = "$\lambda z.z(\lambda x.\lambda y.x)$" and the second element using the function ' 'second" = "$\lambda z.z(\lambda x.\lambda y.y)$".

As already mentioned, in addition to natural numbers and arithmetic operations, the λ-calculus allows you to encode everything needed to implement any algorithm. It is hence important to encode the boolean values `true` and `false`, and the selection operation (arithmetic if). A simple encoding for `true` could be "$\lambda t.\lambda f.t$", while `false` could be encoded as "$\lambda t.\lambda f.f$": informally speaking, `true` and `false` are encoded as λ expressions with two arguments, which return the first or second argument.

The selection function (arithmetic if), on the other hand, can be encoded as "$\lambda c.\lambda a.\lambda b.cab$": it is a λ-expression that receives 3 arguments "c", "a" and "b", where "c" is the encoding of a boolean value. If "c" is the encoding of `true`, then the expression evaluates to "a", otherwise it evaluates to "b":

$$(\lambda c.\lambda a.\lambda b.cab)(\lambda t.\lambda f.t) \rightarrow_\beta \lambda a.\lambda b.(\lambda t.\lambda f.t)ab \rightarrow_\beta \lambda a.\lambda b.(\lambda f.a)b \rightarrow_\beta \lambda a.\lambda b.a$$

And

$$(\lambda c.\lambda a.\lambda b.cab)(\lambda t.\lambda f.f) \rightarrow_\beta \lambda a.\lambda b.(\lambda t.\lambda f.f)ab \rightarrow_\beta \lambda a.\lambda b.(\lambda f.f)b \rightarrow_\beta \lambda a.\lambda b.b$$

Based on these encodings it is then possible to implement the boolean operators `and` ($\lambda p.\lambda q.pqp$), `or` ($\lambda p.\lambda q.ppq$), and so on[1].

It is then possible to encode boolean predicates such as "is zero" (which receives the encoding of a natural number as an argument and evaluates to `true` if the number is 0), "less than" , "equal" and similar.

Although the encodings of values and operations presented so far allow to implement generic functions (a mechanism to implement / encode recursion or iteration is still missing, but will be shown shortly), the resulting lambda expressions risk to be too complex. For example, the simple arithmetic expression "$2 + 3$" is encoded as "$2 + 3 \equiv (\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(f(fx)))$"!!! And the encoding of the function "$f(a) = a+2$" is "$\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$" . To simplify the expressions, it is possible to use a notation sometimes known as "applied lambda calculus", in which the encodings of the various values and operations are replaced with more common mathematical symbols. Thus, "+" is a synonym for "$(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))$", "2" is a synonym for "$(\lambda f.\lambda x.f(fx))$" and so on. It then becomes possible to write "$\lambda a.a + 2$" instead of the lambda expression mentioned above.

At this point, the most important thing that seems to be missing is a loop construct (or rather, recursion, since we are talking about functional programming!). As already mentioned, the lack of a global environment seems to make it impossible to implement recursion. But the latest surprise of λ-calculus, the concept of fixed point combinator, comes to our aid.

As known, if the "imperative" version of an algorithm contains a loop, its implementation according to the functional paradigm is based on recursion: in other words, the implementation of a function calls the function itself (the typical example is the factorial). But the λ-calculus allows you to define only anonymous functions (λ abstractions) and without a global environment, a function cannot call itself, as it has no name. In other words, a recursive function contains at least one free variable (therefore it is not a combinator), which indicates the name of the function itself, to be called recursively. The first step

[1]The reader can verify the correctness of the encodings of `and` and `or` as a simple exercize.

```
unsigned int factorial(unsigned int n)
{
  unsigned int i res = 1;

  for (i = 2; i <= n; i++) {
    res = res * i;
  }

  return res;
}
```

Figure 2.1: Iterative implementation of the `factorial()` function.

```
unsigned int factorial(unsigned int n)
{
  if (n == 0) return 1;
  return n * fattoriale(n - 1);
}
```

Figure 2.2: Recursive implementation of the `factorial()` function.

to implement recursion in the λ-calculus is therefore to eliminate this free variable (thus transforming the recursive function into a combinator). This can be done by passing the name of the function as an argument. Therefore, if $f = E$ is an expression that recursively calls f (itself), it is transformed into $f_c = \lambda f.E$, binding the variable f, which then becomes the first argument of f_c.

In other words, f can be seen as the result obtained by passing f as an argument to f_c: $f = f_c f$, where in this case "=" means " $\equiv_\beta$" (β equivalent). This is not simply a syntactic trick, but allows us to reformulate our problem as an equation whose solution f is the recursive function we are looking for. Such an equation $f \equiv_\beta f_c f$ is solved by finding the "fixed point" of f_c. The existence of *fixed point combinators* (combinators that given a function f_c compute its fixed point $f = f_c f$) shows us that recursion is implementable in λ-calculus, even if only anonymous functions exist.

The most famous fixed point combinator is $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

As an example, let's see how to use the fixed point combinator Y to calculate the factorial function. A possible imperative implementation of the factorial is shown in Figure 2.1, while the traditional recursive implementation is shown in Figure 2.2. Figure 2.3 shows a "more functional" implementation of this function. A first attempt (not too successful, actually) of conversion to λ-calculus could be

$$\texttt{factorial} = \lambda n.\texttt{cond}\ \ (n = 0)1(n * (\texttt{factorial}\ \ (\text{pred}\ n))$$

Note that this function, which might look strange when talking about pure λ-calculus (as it contains functions like "`pred`",predicates like $n = 0$, and expressions like $n - 1$, which are not part of the pure λ-calculus), was written using the applied λ-calculus. Remember that the selection "`cond`" (arithmetic if) can be replaced with the λ-expression $\lambda c.\lambda a.\lambda b.cab$, the predicate "$n = 0$" can be replaced with the λ-expression encoding of "is zero" and `pred` can be replaced with its encoding mentioned earlier.

As already noted several times, this is a strange form of definition because it requires the presence of a binding for its own name in the global environment. This problem is solved by defining the function f_c as

$$\lambda f.\lambda n.\texttt{cond}(n = 0)1(n * (f(\texttt{pred}\ \ n))$$

and finding the function `factorial` such that `factorial`$= f_c$`factorial` (fixed point of f_c). This function f can be calculated using the fixed point combinator Y: `factorial`$= Y f_c$.

Finally, it should be noted that the encodings of data types[2] and high-level functions presented above are not unique. For example, using Church numerals it is possible to define operations such as addition or predecessor in different ways (of course all the definitions will be functionally equivalent).

Going further, it can be seen that Church encoding is only one of the possible way to encode high-level data structures and functions and other alternative encodings are possible. For example, the so-called

[2]Actually, only the encoding of natural numbers has been presented... But it is possible to encode in λ-calculus any type of data.

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

Figure 2.3: Functional implementation of the `factorial()` function.

Scott's numerals propose an alternative encoding of natural numbers (and operations on them) to that of Church:

- The natural number 0 is represented by the λ expression $\lambda f.\lambda x.f$
- The function `succ` which calculates the next of a natural number n is encoded by the λ expression $\lambda n.\lambda f.\lambda x.xn$

Although Scott's coding is less known (and less used!) than Church's, in some cases it has advantages (for example, it allows simplifying the definition of the predecessor function `pred` which is codable as $\lambda n.n(0)(\lambda x.x)$).

2.3 Removing Abstractions

As seen, the main constructs of the λ-calculus are abstraction (definition of functions) and function application. Actually, it is possible to define a minimal functional programming language even without using the abstraction mechanism, provided that an adequate set of predefined functions is provided. What you get in this way is a "*combinatory calculus*", so called because of the predefined combinators (the set of predefined functions mentioned above) on which it is based.

The syntax of an expression of this type of calculus can be defined as:

```
<expression> ::= <name> ; lowercase letter
               | <Combinator> ; default function
               | (<expression> <expression>) ; application
```

where `<name>` is the name of a variable and `<Combinator>` is a built-in function (with a well-defined behavior). This is equivalent to the following inductive definition:

- A name / variable (denoted by a single lowercase letter) is an expression of combinatory calculus
- A combinator (denoted by a single uppercase letter) is an expression of combinatory calculus
- If e_1 and e_2 are expressions of the calculus, then (e_1e_2) is an expression too

Note how all the variables appearing in an expression are free variables (because there is no concept of abstraction that can bind them).

The details of a combinatory calculus clearly depend on the predefined combinators and it is clear that not all the combinations of combinators result in a turing complete calculus.

The most important of the combinatory calculus is probably the "SK calculus" (sometimes known as "SKI calculus"), where the default combinators are S and K (plus optionally the identity combinator I) defined by the following properties:

$$\begin{aligned} Kxy &= x \\ Sxyz &= xz(yz) \\ Ix &= x \end{aligned}$$

The combinator I is often used to simplify calculus expressions, but it is not strictly necessary, as it can be obtained as $I = SKK$: $(SKK)x = SKKx = Kx(Kx) = x$.

The S and K combinators presented in Section 2.2 ($S = \lambda f.\lambda g.\lambda x.fx(gx)$ and $K = \lambda x.\lambda y.x$) enjoy the properties described above (and the proof of this is is left to the reader). It is therefore easy to convert an expression of the SK calculus (or SKI calculus) into a λ-expression. Since it is also possible to convert any λ-expression into an expression of the SK calculus, the SK calculus has the same expressive power as the λ-calculus and is therefore Turing-complete.

The conversion of a generic λ-expression E into an expression of the SK calculus can be performed proceeding by cases. In particular, E can be:

- An identifier x, which maps to the expression x of the SK calculus
- An application E_1E_2, which maps to the expression E_1E_2 of the SK calculation
- An abstraction $\lambda x.E'$, which must be converted into an expression E'' of the SK calculus proceeding by case:
  - If E' is an identifier, it can be x, in which case $E = \lambda x.x$ maps to $E'' = I = SKK$, or a symbol $y \neq x$, in which case $E = \lambda x.y$ maps to $E'' = Ky$
  - If E' is an application $E'_1E'_2$, $E = \lambda x.E'_1E'_2$ must be converted into E'' such that $(\lambda x.E'1E'_2)v = E''v$. This implies that

$$(\lambda x.E'_1E'_2)v = E''v \Rightarrow E'_1[x \to v]E'_2[x \to v] = E''v \Rightarrow$$

$$(\lambda x.E'_1v)(\lambda x.E'_2v) = E''v \Rightarrow S(\lambda x.E'_1)(\lambda x.E'_2)v = E''v \Rightarrow$$

$$E'' = S(\lambda x.E'_1)(\lambda x.E'_2)$$

  - If E' is an abstraction $\lambda y.E'_1$, $E = \lambda x.\lambda y.E'_1$ must be converted into E'' by recursively applying this procedure at E'_1.

Applying this reasoning, one can convert any λ-expression into an expression based only on free variables, the operator S and the operator K.

Another important property of the SK calculus is that every λ-expression that contains no free variables (that is, a combinator) can be converted to an SK calculus expression that contains no variables. In other words, to model combinators it is possible to remove the first clause (a variable name is an expression of the SK calculus) from the definition of the SK calculus.

The process which converts an expression "E" into an expression $R_x(E)$ which does not contain the free variable "x" but behaves like $\lambda x.E$ (that is that is, $R_x(E)E_1 = E[x \to E_1]$) is known as *bracket abstraction* and can be used to convert any λ-expression to an expression of the SK calculation eliminating the abstractions $\lambda x.$ one by one. A very simple (although not efficient) bracket abstraction algorithm is based on the following transformations:

1. $R_x(x) = SKK$, where "x" is the variable to be eliminated
2. $R_x(y) = Ky$, where y is a predefined combinator (S or K) or a variable other than the variable x to be eliminated
3. $R_x(E_1E_2) = SR_x(E_1)R_x(E_2)$

To make the algorithm slightly more efficient, the second rule can be replaced by $R_x(E) = KE$, where "E" is an expression that does not contain as a variable free the variable "x" that has to be removed.

To convert a λ-expression into an SK calculus expression, one can proceed by removing the λ abstractions one by one by applying the three rules above. Note the close relationship between this bracket abstraction algorithm and the conversion methodology shown above.

2.4 Typed Lambda Calculus

As noticed, in the "pure" λ-calculus there is no concept of data type. It has been shown how it is possible to use expressions of the untyped λ-calculus to encode the various data types (and the operations on them), but the variables of the λ-calculus represent generic functions (with unspecified domain and codomain).

Although the lack of data types does not impact the expressivity of the formalism (as mentioned, the λ-calculus is Turing complete), it can compromise the readability and simplicity of use, making it easier to introduce programming errors (for this reason the λ-calculus is considered a sort of "Assembly of functional languages"). For example, if you code the function $f(a) = a + 2$ using λ-calculus you get $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$ (not exactly intuitive expression...), which using the applied λ-calculation simplifies to $\lambda y.y{+}2$. However, this encoding has lost a fundamental characteristic of the initial function: the fact that the function operated on numbers! In fact, it is possible to apply $\lambda a.a{+}2$ (which, we recall, is equivalent to $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx)))$ to any λ-expression, even if it doesn't encode a number! If the function is applied to an expression E which encodes a natural number, a λ-expression which encodes a natural number is generated as a result, otherwise a λ-expression E' without any clear interpretation can be generated.

To solve this kind of problems, it is possible to associate a type with each λ-expression (or with each argument). For example, introducing the constraint that $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$ is a function $\mathcal{N} \to \mathcal{N}$ or, better, that in this expression "a" is the encoding of a natural number ($a : \mathcal{N}$).

In this section (which does not claim to be exhaustive, but only to introduce some concepts that can then be further explored by the readers) it will be shown how to extend the original formalism to specify domain and codomain for each function. Of course this can be done in various ways which result in different definitions of typed λ-calculus, the most famous of which are due again to Alonso Church and Haskell Curry. We will then talk about λ-calculus with types "a-la Church" or "a-la Curry".

Surprisingly enough, associating types to functions actually reduce the expressive power of the formalism, which is no longer Turing complete. This happens because it can be proved that the reduction of any "well-typed" expression (this concept will be introduced intuitively in the next few pages) by a typed λ-calculus always ends (it is therefore no longer possible to express infinite recursions). For a simple intuition of this fact try to compute the type of the Y operator, which is necessary to encode recursive functions.

First of all, to define a typed λ-calculus we need to introduce the concept of type; this can be done by introducing a set $\mathcal{P}$ of basic types, or *primitive types*, and a rule for defining new datatypes starting from existing ones (this is analogous to what was done to define the expressions of the λ-calculus, created starting from a set of base names and 2 rules which allow to create new expressions starting from valid expressions). Since we are defining types for a λ-calculus, a new type γ could be defined starting from two types α and β: $\gamma = \alpha \to \beta$ (γ is the type of functions having α as domain and β as codomain). In other words, the set $\mathcal{T}$ of possible data types can be generated through the following inductive definition:

- A primitive type name refers to a type: $\alpha \in \mathcal{P} \Rightarrow \alpha \in \mathcal{T}$
- If α and β are types, then $\alpha \to \beta$ is also a type: $\alpha, \beta \in \mathcal{T} \Rightarrow \alpha \to \beta \in \mathcal{T}$

As it is easy to understand from this definition, the number of possible types (the cardinality of the set $\mathcal{T}$ of types) is infinite.

Given a λ-expression E, its type is computable according to the following rules:

- The type of a free variable x must be known a-priori
- If E_1 and E_2 are expressions with types $\alpha \to \beta$ and α, then the type of E_1E_2 is β: $E_1 : \alpha \to \beta, E_2 : \alpha \Rightarrow E_1E_2 : \beta$
- If E is an expression of type β, $\lambda x.E$ has type $\alpha \to \beta$: $E : \beta \Rightarrow \lambda x.E : \alpha \to \beta$

the third rule can be better clarified by modifying the syntax of the abstraction to specify the type of the argument. In these cases $\lambda x : \alpha.E$ (explicit typing) is used instead of $\lambda x.E$ (implicit typing). Making an analogy with higher level languages, we can consider the C language (in which a variable declaration requires to specify the variable's type), the C++ language (in which the compiler can be asked to infer the tyoe of a variable, by usign the ' '`auto`" keyword), and standard ML or Haskell (where variables can be declared without specifying their type, because the compiler is able to infer it himself).

Also note that to associate a type with an expression E it is necessary to make assumptions about the types of the free variables contained in E (see the first rule above). These assumptions (obviously only necessary for non-closed expressions) are contained in some sort of "type environment", or "type context". In summary, the type of a closed expression can be somehow "computed" (better: inferred) without needing additional information, while the type of an open expression depends on the environment (or context) of the types.

Informally speaking, an expression E is correctly typed if it is possible to associate E with a type $\alpha \in \mathcal{T}$ that is consistent with the rules presented above. For example, the expression $\lambda x : \texttt{int}.x$ is correctly typed (and has type $\texttt{int} \to \texttt{int}$). The expression $I = \lambda x.x$ (or $I = \lambda x : \alpha.x$) is also correctly typed and has type $\alpha \to \alpha$. The combinator $\omega = \lambda x.xx$ is instead not correctly typed: assuming that x has type α ($x : \alpha$), we have that $\omega : \alpha \to \beta$, where β is the type of the expression "xx". But for "xx" to be a valid expression, x must be a function, with domain α (the type of the argument). Thus, $x : \alpha \to \beta$, but also $x : \alpha$, from which we derive $\alpha = \alpha \to \beta$, which is not a valid expression in the type system we have defined (that is, using the type generation rules given above it is not possible to "construct" a type $\alpha \in \mathcal{T}$ that has this property).

Once the concept of types and correctly typed expressions has been introduced, two different approaches can be followed:

- The first approach consists in *defining the semantics of expressions regardless of their type* (in practice, we define β reduction rules that do not depend on the types of the expressions). The concept of type is then only used a posteriori to "reject" incorrectly typed expressions as invalid
- The second approach is to specify semantics only to correctly typed expressions. In other words, if an expression is not correctly typed (that is, its type cannot be generated with the rules presented above), it doesn't even make sense to try to reduce it.

According to the first approach, which results in the so-called "λ-calculus with types a-la Curry", the introduction of types is used to "eliminate" from the calculus the expressions which ' 'do not behave as desired" (for example, expressions whose reduction does not finish). But the reduction of such expressions is still defined. In essence, types simply add extra constraints on expressions, which characterize "valid expressions".

In the second approach, which results in the so-called "λ-calculus with types a-la Church", the type of an expression is considered fundamental for its semantics (it is not possible define the semantics of an incorrectly typed expression). Hence, the reduction rules for λ-expressions explicitly refers to the expressions' types.

In the literature, implicit typing is sometimes used in the a-la Curry calculus and explicit typing in the a-la Church calculus, but this is not strictly necessary.

Regardless of whether implicit or explicit typing is used, it is possible to reduce a typed λ-calculus expression using the traditional β reduction rule of the untyped λ-calculus, after removing type annotations (": α" and similar) from bound variables. Following the "a-la Church" approach, this can only be done provided that the correct typing of the expression has been verified.

As an alternative, once a type is associated to each expression, the β reduction rule must be updated to take it into account: if in the typeless λ-calculus

$$(\lambda x.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

in typed lambda calculus, reduction is possible only if "x" and "E_1" have the same type:

$$E : \alpha \Rightarrow (\lambda x : \alpha.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

.

Chapter 3

Haskell for Dummies

3.1 Introduction

Imperative programming languages base their computation model on the Von Neumann architecture, describing programs as sequences of commands (instructions) that modify a state (for example, the contents of memory locations identified by variables). On the other hand, functional programming languages encode programs as expressions that are evaluated, generating values as results. Therefore, there is no longer a direct reference to the Von Neumann architecture and the very concept of "state" or mutable variables (variables whose content can be modified) is missing.

As mentioned, programs written in a functional language are executed by evaluating expressions. Informally speaking, there are "complex" expressions, which can be simplified, and "simple" expressions, which cannot be further simplified. An expression that cannot be simplified is a *value*, while a complex expression can be simplified to a value; the operation that computes this value is called *reduction* (or evaluation) of the expression[1].

A complex expression is therefore composed of operations (or functions) applied to values and the order in which these functions and operations are evaluated depends on the language. For example, $4 * 3$ is a "complex" expression composed of the values 4 and 3 and the multiplication operation. By evaluating this expression, it reduces (simplifies) to 12, which is the value of its result. The expression "`if (n == 0) then (x + 1) else (x - 1)`", on the other hand, is more "interesting", because it is not clear if and when the subexpressions "`x + 1`" and "`x - 1`" are evaluated: a programming language could decide to always evaluate the two expressions **before** evaluating the expression `if`, or it could decide to evaluate "`x + 1`" only if "`n == 0`" and evaluate " `x - 1`" only if "`n != 0`".

In case of "eager evaluation", the evaluation of the expression is done by first evaluating the parameters of each operation and then applying the operation to the obtained values, while in case of "lazy evaluation" the various sub-expressions are evaluated only when their value is actually used: therefore, if an expression is passed as a parameter to a function (or as an argument to an operation) but the function does not use that parameter, the value of the expression is not is evaluated.

In languages such as Standard ML that use an "eager" evaluation mechanism, therefore, an expression composed of an operator applied to one or more arguments is evaluated by first reducing its arguments and then applying the operator to the values obtained by evaluating the arguments. Conversely, in languages like Haskell that use a "lazy" evaluation mechanism, an expression composed of an operator applied to one or more arguments is evaluated by reducing the arguments only when the operator actually uses them .

In summary, a program written in a functional language is nothing more than an expression (or set of expressions), which is evaluated when the program executes. Complex programs are often written by defining functions (in the mathematical sense of the term) which are then called by the "main" expression describing the program itself. In theory, evaluating these expressions should have no side effects, but some types of side effects (for example, inputs and outputs) are often very difficult to avoid. From all this one can at least guess how a functional programming language can be informally seen as a "high-level version" of λ calculus (which adds at least the concept of global environment and some syntactic sugar).

In this regard, it is important to note that in languages such as Haskell and Standard ML (unlike other functional languages such as LISP or Scheme) the various expressions that compose a program are

[1]It should be noted that there are expressions for which the reduction process never ends and so the simplification does not arrive at a value. Such expressions can be seen as the "functional equivalent" of infinite loops.

written as operations or functions acting on *values belonging to a type*. The functions themselves are characterized by a type (an arrow between the type of the argument and the type of the result)[2]. In this sense, we can say that languages like LISP or Scheme derive directly from the "simple" λ calculus, while languages like those of the ML family (Standard ML, ocaml, F#, ...) and Haskell derive from typed λ calculus (although they must use a recursive type system to be Turing-complete).

In a statically typed language, the type of an expression and its parameters is determined before executing the code: the compiler/interpreter infers (extrapolates) the types of parameters and expression by analyzing the code, without executing it. Some of the static, strictly typed languages, such as Haskell, complicate things a bit because they use a type system that is also *polymorphic*, i.e., a single expression can assume different types depending on the context in which it is used (but, once the context is fixed, each expression is evaluated — albeit lazyly — to a value belonging to a type, so the typing remains strict). The use of this polymorphic type system coupled with the lazy evaluation used by Haskell can sometimes be misleading into thinking that the language does not use strict typing.

In cases where the type of values and expressions cannot be easily inferred by the compiler (or by the interpreter) or it is not desired to make use of polymorphism, some languages (including, for example, the languages of the ML family and Haskell) may allow the programmer to annotate expressions with their type, using a special syntax (which you will see later).

3.2 Expressions and Data Types in Haskell

The basic data types provided by Haskell are: `()` (sometimes called "unit", which is the equivalent of the "`void`" type in other languages), tt Bool, `Char`, `Int`, `Float`, `Double` and `String`.

In addition to providing these basic types, Haskell allows you to use combinations of types in the form of *tuple*, to define synonyms for existing data types, and to define new data types (whose values are generated by special functions called *constructors*).

The type `()` is composed of a single value, also called `()`[3] and is generally used as part of the result of expressions that would not generate any value (and that are only important for their own side effects). In theory such expressions should not exist in a purely functional programming language, but the need to do Input/Output (which is a side effect) complicates things. Haskell solves the problem by making I/O expressions return both an effect (encoded by the type "`IO` α") and a value (which for output functions is often not relevant, so the type "`()`" is used... The type of the output functions will then be "`IO ()`"). Another way to think of the "`()`" type is to think of it as a "tuple of 0 items" type.

The `Bool` type, on the other hand, is composed of two values (`True` and `False`).

The `Int` type is composed (as the name suggests) of positive and negative integers. Some operators are defined on these numbers: the unary operator `-` and the standard binary operators representing the basic arithmetic operations `*`, `+` and `-`[4]. The division operator `/` is not defined on integers (while there is a function `div` which computes the integer division). Note that unlike other languages Haskell accepts expressions like "`5 / 2`" (instead of reporting an error because `5` and `2` are integer values but the operation "`/`" is not defined on integers). This happens because Haskell implicitly converts `5` and `2` to floating point values `5.0` and `2.0`. Finally, it should be noted that "`div`" is a function, not an operator, so the correct syntax to use it is "`div 5 2`", not "`5 div 2`" (If you wish, Haskell allows you to use binary functions with the infix operator notation as long as you quote them in single quotes: "`5 ‘div‘ 2`").

The `Float` and `Double` types are composed of a set of approximations of real numbers (in single precision for `Float` and in double precision for `Double`). The values of these types can be expressed by integer and fractional parts (for example, `3.14`) or by using the exponential form (for example, `314e-2`). Again, the `-` symbol can be used to negate a number (reverse its sign). Two special values `NaN` (Not a Number) and `Infinity` can be used to indicate values that cannot be represented as real numbers or infinite values (the result of dividing a real number by 0).

The `Char` type is composed of the set of characters. A value of this type is represented enclosed in single quotation marks, as in C; for example, `'a'`.

The `String` type is composed of the set of strings, represented in quotation marks; for example `"test"`. Haskell defines the `++` concatenation operator on strings: `"Hello, " ++ "world" = "Hello, World"`.

[2]Formally, a type can be defined as a set of values and the type of a value indicates the set to which that value belongs
[3]Warning! Note that the type and its only value have the same name, "`()`", and this can sometimes cause some confusion
[4]note that in Haskell the operation `-` (subtraction) and the unary operator `-` which reverses the sign of a number are represented by the same symbol.

In addition to the "classic" operators on the various types of variables, Haskell provides a selection operator `if`, which allows you to evaluate two different expressions depending on the value of a predicate. The syntax of an `if` expression in Haskell is:

```
if <p> then <exp1> else <exp2>
```

where `<p>` is a predicate (an expression evaluating to a boolean value) and `<exp1>` and `<exp2>` are two expressions evaluating to values of the same type or compatible types (note that tt ¡exp1¿ and `<exp2>` must have compatible types because the value of the resulting `if` expression has the same type as `<exp1>` and `<exp2 >`). The expression `if` evaluates to `<exp1>` if `<p>` is true, while it evaluates to `<exp2>` if `<p >` is false.

Although Haskell's `if` operator is often seen as the expression-level equivalent of the `if` selection operation provided by imperative languages such as C, C++, Java, or the like, it is important to note a few differences. For example, the selection constructs of an imperative language allow to execute a block of operations if the predicate is true (`then` block) or a different block of operations if the predicate is false (`else` block). In theory, each of the two blocks (`then` or `else`) can be empty, meaning that there is no operation to perform for a given truth value of the predicate. Haskell's `if` operator, on the other hand (like the equivalent operator of all functional languages), must *always* be evaluable to a value. Thus, neither `then` or `else` expression can be empty. In this sense, a Haskell "`if` *predicate* `then` *expression1* `else` *expression2*" expression is equivalent to the if arithmetic "*predicate* `?` *expression1* : *expression2*" of the C or C++ language.

An example of using `if` is

```
if a > b then a else b
```

which implements an expression evaluated to the maximum between `a` and `b`.

Finally, an important feature of the Haskell type system should be noted: the existence of *classes of types* (typeclasses), which group together various types having certain properties (that is, for which some specific operations are defined).

For example, the class of types "`Eq`" contains all types on which the comparison operator "==" is defined, the class "`Show`" contains all types displayable on screen via "`print`", class "`Ord`" contains all types on which comparison operators are defined ("`<`" and "`>`"), the class "`Num`" contains all types expressing numeric values ("`Int`", "`Float`' ', "`Double`"), the class "`Fractional`" contains all types representing non-integer numbers ("`Float`" and "`Double` ") and so on.

Haskell's type inference mechanism will not associate a value with a type, but with a class of types; so, for example, the value "`5`" is not associated with the type "`Int`", but with a generic type of the class "`Num`"; if you want to specify the type of a value in a precise and unambiguous way, this must be done explicitly.

3.3 Associating Names to Values

The expressions composing a Haskell program can directly use values (irreducible expressions) as operands or they can use *identifiers* defined in a *environment* to represent values (technically, the environment is said to contain *bindings* between names and values). An environment can be seen as a set of pairs (identifier, value) which associate names (or identifiers) to values[5].

While there is no concept of a mutable variable, the various "bindings" that bind names and values in the environment can vary over time. The environment can in fact be modified (actually, extended) by associating a value (of any type) with a name (identifier) using the "`=`" operator:

```
<name> = <value>
<name> :: <type>; <name> = <value>
```

The "`=`" operator (called *definition* in Haskell) adds a binding between the "`<name>`" identifier and the "`<value>`" value to the environment, while "`::`" (called *declaration*) allows you to specify the type of the value. The value can also be the result of an expression evaluation; in this case, the definition takes the form

```
<name> = <expression>
```

[5]Note that the environment described here is a global environment and each function will have a then its local environment.

for example, `v = 10 / 2;` binds the name "v" to the value "5".

An interesting peculiarity of Haskell is that the type of the value associated with a name is not determined at the moment of definition (moment in which a binding between the name and the value is added to the environment), but when the value is actually used. This implies that a "a = 5" definition associates the name "`a`" to a generic numerical value "5", not to an integer, floating point or other. More precisely, the type of "`a`" is "`Num p => p`", indicating a generic type "`p`" belonging to the class of types "`Num`". Technically, the symbol "`p`" representing a type is called *type variable* and the expression "`Num p =>`" represents a constraint on that variable (in this case, "`p`" must be a numeric type). Note that the "`a/2.5`" operation is possible (unlike in other languages), because the "`/`" operation is defined on arguments whose type belongs to class "`Fractional`" and "`a`" has a type belonging to class "`Num`"; since "`Fractional`" is a subset of "`Num`", the type of "`a`" may be of class " `Fractional`" (like the type of the value "`2.5`") and is therefore type-compatible with "`/`". The type of the result is obviously "`Fractional a => a`".

If you want the type of the value associated with a name not to be polymorphic (but to be a well-specified type), you have to declare it explicitly as in

```
a = 5
to :: Int
```

In this case, the "`a/2.5`" operation will not be possible and the compiler will throw an error. It is important to note the difference between Haskell's polymorphic inference mechanism and the data type promotion (or automatic value conversion) mechanism used by other languages: in Haskell, "`5 / 2.5`" is possible because the value "`5`" can have a type belonging to the class "`Fractional`", while in C, C++, Java or similar " `5`" is an integer value that is automatically converted to a floating point to perform division.

Returning to the creation of new bindings in the environment, it is interesting to note that in Haskell the binding of a name to a value can be seen as a *variable declaration*: for example, `pi = 3.14` creates a variable identified by the name `pi` and binds it to the real value 3.14. But it should still be noted that these variables are simply names for values, they are not containers of mutable values. In other words, a variable is immutable and always has a constant, unchangeable value. A subsequent declaration `pi = 3.1415` does not modify the value of the variable `pi` but creates a new value 3.1415 of type `Float` or `Double` and associates it with the "`pi`" name, "masking" the previous binding. Haskell always uses the last value that was mapped to a name. This means that the keyword "`=`" **modifies the environment, not the value of variables**: "`=`" always defines a new variable (initialized with the specified value) and creates a new binding (between the specified name and the created variable) in the environment.

Finally, remember that in a functional programming language functions are expressible, denotable, and storable values. Hence, a Haskell variable can denote function values too. As an example, a "`sum2`" function can be defined as:

```
        sum2 x y = x * x + y * y
```

3.4 Functions

A particular data type that has not been previously mentioned but is related to a fundamental feature of functional programming languages is the *function* data type. As suggested by the name, a value of this type is a function, in the mathematical sense of the term: a relation that maps each element of a domain set into one and only one element of a codomain set. In a functional language, the domain and codomain sets are defined by the types of the parameter and the returned value. Looking at things from a different point of view, a function can be considered as a *parameterized expression*, i.e. an expression whose value depends on the value of a parameter.

Remember that considering functions with only one parameter is not reductive: the parameter can be a n-tuple of values (therefore, the domain set is the Cartesian product of n sets), or the *currying* mechanism (see Section 3.7) can be used to reduce functions with multiple parameters to functions with only one parameter. According to the definition given above the only effect of a function is the computation of a value (result, or return value) based on the value of the parameter. Functions cannot therefore have *side effects* of any kind (that is, they cannot have effects that are not in the return value of the function).

As in all functional languages, function values are *expressible*, that is, they can be generated as the results of expressions. In particular, Haskell uses the symbol "\" to represent λ-calculus abstractions

("\" is an approximation of the Greek letter λ) and hence generate values of type function. The syntax is:

```
\ <param> -> <expression>
```

where `<param>` is the name of the formal parameter while `<expression>` is a valid expression, which can use the names present in the global environment plus the name `<param>`. The expression \x –> **exp** when evaluated has therefore as a result a value of type function. Each time the function will be applied to a value (actual parameter), this value will be linked to the name `x` (formal parameter) in the local environment in which the expression `exp` is evaluated (the "\" construct thus creates a local environment for the function!).

For instance,

```
\n -> n + 1
```

is a function that increments a number (in this case, the formal parameter is `n` and the expression to evaluate when applying the function is `n + 1`). A function can be applied to a value by writing the function followed by the actual parameter. For instance,

```
(\n -> n + 1) 5
```

applies the function \`n -> n + 1` to the value 5 (the parentheses are necessary to indicate the order of precedence of the operations: first the function is defined and then it is applied to the value 5). This means that the value 5 (actual parameter) is bound to the name `n` (formal parameter) and then the expression `n + 1` is evaluated, which gives the function return value. The result of this expression is obviously 6.

Like all other values, a function-type value can also be mapped to a name using Haskell's "`=`" definition mechanism. For example, the following code will define a variable (immutable, remember!) `increment` whose value is a function that adds 1 to the value passed as a parameter (in other words, it will associate the name `increment` to this function):

```
        increment = \n -> n + 1
```

The type of this variable is `Num a => a ->a` (which is equivalent to the mathematical notation "$f : \mathcal{X} \rightarrow \mathcal{X}$", where "$\mathcal{X}$" is a numerical set: $\mathcal{N}$, $\mathcal{Z}$, $\mathcal{R}$, ...). It is now possible to apply this function to an actual parameter by using its name: "`increment 5`" will obviously result in "6". Associating symbolic names to functions (that is, creating variables of function type) is useful when the function must be used/referenced multiple times... For example, consider

```
        (\x -> x+1) ((\x -> x+1) 2)
```

versus

```
        increment (increment 2)
```

Haskell also provides a simplified syntax for defining functions and associate names to them:

```
        <name> <param> = <expression>
```

is (almost) equivalent to

```
        <name> = \ <param> -> <expression>
```

As an example, the `increment` function can be defined as

```
        incrementa n = n + 1;
```

which looks more readable than the definition based on the lambda expression and a variable definition (which is more similar to the λ-calculus).

This syntax does not introduce new functionalities (and does not increase the language's expressive power), but is just *syntactic shugar* used to simplify the "`...  = \...`" definitions[6].

[6] Actually, things are a little bit more complex, for multi-parameters functions.

3.5 Definitions by Cases

Besides being able to be defined through an arithmetic expression that allows to calculate its value (as just seen), a function can also be defined "by cases", explicitly specifying the value of the result corresponding to each value of the parameter (or to specific values, then using a generic expression for the remaining cases). This is generally useful for functions defined by induction (or for recursive functions), where we distinguish values for inductive bases and for inductive steps.

The case definition can be easily implemented using the `case` operator:

```
\x -> case x of
        <pattern_1> -> <expression_1>
        <pattern_2> -> <expression_2>
       ...
       ...
        <pattern_n> -> <expression_n>
```

where the expression **case** x **of** p1 —> e1; p2 —> e2; ... first evaluates the expression `x`, then compares the obtained value with the specified *patterns* (`p1`, `p2`, etc...). As soon as the comparison is successful (a *match* occurs), the matching expression is evaluated by assigning the resulting value to the `case` expression.

An easy way to define a function "by cases" is then to use constant values as a patterns to enumerate the possible values of the parameter. An example of a function defined by case (or by enumeration) is:

```
day = \n -> case n of
                1 -> "Monday"
                2 -> "Tuesday"
                3 -> "Wednesday"
                4 -> "Thursday"
                5 -> "Friday"
                6 -> "Saturday"
                7 -> "Sunday"
                _ => "Invalid day"
```

note that the strange "_" pattern is used to "catch" all cases not previously enumerated (integers smaller than 1 or greater than 7).

Hence, the `case` operator seems to be the expression-level equivalent of the `case` or `switch` command (multiple selection command) that exists in many imperative languages. However, there is an important difference: Haskell's `case` expression (like the equivalent of many functional languages) allows you to use not only constant patterns (as seen in the previous example), but also more complex patterns containing names or constructs such as tuples or similar. When comparing the value of an expression `x` against these "more complex" patterns, Haskell employs a *pattern matching* mechanism. For example, if a pattern contains variable identifiers, when Haskell compares the expression against that pattern it can create bindings between these identifiers and values so that the match is successful. For example, in the following code

```
f = \a -> case a of
            0 -> 1000.0
            x -> 1.0 / x
```

if the parameter of the function is not `0` (the first pattern does not match) a binding between `x` and the value associated with `a` is created (so that the second pattern can match).

More complex patterns based on tuples can be used (even without using the "`case`" keyword) to define multi-variable functions:

```
sum = \ (a, b) -> a + b
```

In this case, when the "`sum`" function is invoked the pattern matching mechanism creates a binding between "`a`" and the first value of the pair passed as an actual argument (similarly, a binding between "`b`" and the second value of the pair is created).

It also interesting to notice how this expression

```
        f = \x -> case x of
            <pattern_1> => <expression_1>
            <pattern_2> => <expression_2>
             ...
             ...
            <pattern_n> => <expression_n>
```

can be rewritten without using the "`case`" keyword. For example, as

```
        f x = case x of
            <pattern_1> => <expression_1>
            <pattern_2> => <expression_2>
             ...
             ...
            <pattern_n> => <expression_n>
```

which can be further simplified to

```
        f <pattern_1> -> <expression_1>
        f <pattern_2> -> <expression_2>
         ...
         ...
        f <pattern_n> -> <expression_n >;
```

The latter syntax is sometimes more readable, as it allows you to specify even more explicitly the result value of the function for each value of the argument. More formally, Haskell compares the value of the actual parameter with which the function is invoked against the various patterns specified in the definition `<pattern_1>`...`<pattern_n>`. If the current parameter matches `<pattern_1>`, the first definition is considered and the function is evaluated as `<expression_1>`; if the current parameter matches `<pattern_2>`, the function is evaluated as `<expression_2>` and so on. In other words, *the creation of the link between formal parameter (name) and actual parameter (value) in the local environment of the function takes place by pattern matching.* Note that the "standard" definition f x = <expression> is a special case of this form.

Using this syntax, the definition of the function `day` presented above becomes:

```
day 1 = "Monday"
day 2 = "Tuesday"
day 3 = "Wednesday"
day 4 = "Thursday"
day 5 = "Friday"
day 6 = "Saturday"
day 7 = "Sunday"
day _ = "Invalid day"
```

Again, it is important that the patterns `<expression_1>`, ... `<expression_n>` cover all possible values that the function can take as actual parameters. Hence, the special symbol "(_)" allows to match all the values not covered by the previous clauses.

Pattern matching is a very generic mechanism, used in many other contexts and not only to define functions for cases. In general, it can be said that the pattern matching mechanism is used whenever a binding between a name and a value is created (even independently from the definition of functions): for example, the expression

```
  x = 2.5
```

creates a link between the floating point value 2.5 and the name `x` to match the pattern "x" with the value 2.5. More complex patterns can be used to create multiple links at the same time; for instance

```
(x, y) = (4, 5)
```

will bind the name `x` to the value `4` and the name `y` to the value `5` in order to create a match between the pattern `(x, y )` and the pair value $(4, 5)$.

To define pattern matching more precisely, we can say that a pattern can be:

- a constant value, which matches only that specific value;
- a *variable pattern* `<var>`, which matches any value, after creating a binding between the name `<var>` and the value;
- a *tuple pattern* `(<pattern_1>, <pattern_2>, ..., <pattern_n>)`, which composes n simpler patterns in one tuple. In this case we have a match with a tuple of n values if and only if each of the values of the tuple matches the corresponding pattern of the tuple pattern;
- the wildcard pattern _, which matches any value.

Note that the value `()` can be seen as an example of a pattern tuple (null tuple).

As we all know, a program coded according to the functional programming paradigm uses the recursion mechanism wherever an imperative program uses iteration. It is hence fundamental to understand how to implement recursion using Haskell.

A recursive function is a function similar to the other ones, and there is no need to define it in any particular way: the function body can simply invoke the function itself, without issues. An example is

```
fact = \ n -> if n == 0 then 1 else n * fact (n - 1)
```

which could also be defined as

```
fact n = if n == 0 then 1 else n * fact (n - 1)
```

or even

```
fact 0 = 1
fact n = n * fact (n - 1)
```

3.6 Controlling the Environment

Like many modern languages, Haskell uses static scoping: in a function, *non-local* symbols are resolved (ie: associated with a value) by referencing the environment of the code block in which the function is defined (and not to the caller's environment).

Note that the lambda construct (or the "beautified syntax" for defining functions) creates a static nesting block (like the { and } symbols in C). Inside this block a new binding between the symbol that identifies the formal parameter and the value of the current parameter with which the function will be invoked is added. This new binding may introduce a new symbol or mask an existing binding. For example, a "v = 1" definition creates a link between the symbol "`v`" and the value "`1`" in the global environment. A "f = \v −> 2 ∗ v" definition creates a nesting block (containing the expression "2 ∗ v") where the symbol "v" is no longer associated with the value "1", but with the value of the current parameter with which "`f` " will be invoked. So, "`f 3`" will return 6, not 2.

Non-local symbols[7] are looked up in the active environment *when the definition of the function is evaluated* (and not when the function is called) and can be resolved at that time. For example, a statementf = \ x =>x + y; will result in an error if when this declaration is processed the symbol `y` is not bound to any value.

Haskell also provides two mechanisms for creating static nesting blocks and changing the environment inside them (without changing the environment outside the block): one for creating nesting blocks containing expressions (**let** <definitions> **in** <expression>) ed one for modifying the environment inside a definition (<definition> **where** <definition>).

In other words, "**let** <definitions> **in** <expression>" is an expression evaluated to the value of `<expression>` after the environment is changed by adding the bindings defined in `<definitions>`. These bindings are used to evaluate the expression and are then removed from the environment immediately after evaluation. For example, the `let` construct can be used to implement a *tail recursive* version of the factorial function. Recall that a function is tail recursive if it uses only *tail* recursive calls: the traditional function fact = \n −> **if** n == 0 **then** 1 **else** fact (n − 1) ∗ n It is not tail recursive because the result of `fact(n - 1)` is not immediately returned, but must be multiplied by `n`. A tail recursive version of the factorial function uses a second parameter to store the partial result: fact_tr = \n =>\res =>**if** n == 0 **then** res **else** fact_tr (n − 1) (n ∗ res). This function therefore receives two arguments, unlike the original `fact` function. Hence, a wrapper is needed to invoke `fact_tr`

[7]According to the previous descriptions the local symbols are just the formal parameters of the function.

```
fact = \ n ->
        let
            fact_tr = \ n -> \ res ->
                if n == 0 then
                    res
                else
                    fact_tr (n - 1) (n * res)
        in
            fact_tr n 1
```

Figure 3.1: Implementation of the tail-recursive factiorial function in Haskell, hiding the `fact_tr` function through a `let` block.

```
fact = \n -> fact_tr n 1
    where fact_tr = \n -> \res -> if n == 0
            then
                res
            else
                fact_tr (n - 1) (n * res)
```

Figure 3.2: Implementation of the tail-recursive factiorial function in Haskell, hiding the `fact_tr` function through the `where` construct.

with the right parameters: fact = \n =>fact_tr n 1. However, such a solution has the problem that `fact_tr` is visible not only to `fact` (as it should be), but in the whole global environment. The `let` construct allows you to solve this problem, as shown in Figure 3.1.

The "<definition1> **where** <definition2>" construct instead allows you to use the bindings defined by `<definition2>` in `<definition1>`, then restoring the original environment. The usefulness of the `where` construct can be better understood by considering the following example: suppose you want to implement in Haskell a function $f : \mathcal{N} \to \mathcal{N}$, even if Haskell does not support the type `unsigned int` (corresponding to $\mathcal{N}$) but only the type `Int` (corresponding to $\mathcal{Z}$). To overcome this limitation, one can define a function `integer_f` which implements $f()$ using `Int` as domain and codomain, calling it from a function `f` which checks whether the value of the parameter is positive or not, and returning -1 in case of negative argument:

```
    f = \n -> if n < 0 then -1 else integer_f n
  where
    integer_f = \n -> ...
```

This solution can be used to avoid making the "`integer_f`" function (which accepts negative arguments without any check) visible to everyone. Similarly, `where` can be used to "hide" the two-arguments function `fact_tr` in the tail recursive definition of the factorial (see previous example in Figure 3.1), as shown in Figure 3.2.

Comparing the two examples in Figure 3.1 and 3.2, it is easy to understand how there is a close relationship between the `let` construct and the `where` construct and how it can always be possible to use `let` instead of `where` (moving the definitions from the `in` block outside the construct).

3.7 Functions Working on Functions

Since a functional language provides the function data type and the ability to view functions as denotable values (handled similarly to values of other more "traditional" types such as integers and floating points), it is possible to define functions that take other functions as parameters and thus work on functions. Similarly, the return value of a function can itself be a function. From this point of view, functions that take functions as parameters and return functions are similar to functions that take and return (for example) integer values. However, there are some details that deserve to be considered more carefully and that motivate the existence of this section.

To analyze some interesting peculiarities of functions working on functions (often referred to as "high-order functions" in the literature) let us consider a simple example based on the computation of the derivative (or better, of an approximation of it) of a function $f : \mathcal{R} \to \mathcal{R}$. Starting by defining a function `computederivative` which accepts as parameter the function f to calculate the derivative of and a number $x \in \mathcal{R}$. The function `computederivative` returns an approximation of the derivative of f computed in x. This function, which has a function parameter and a floating point parameter and returns a floating point value, can be implemented, for example, as:

```
computederivative = \ (f, x) -> (f(x) - f(x - 0.001)) / 0.001
```

Notice that Haskell is able to infer the type of the "`x`" parameter (which turns out to be a type "`a2`" belonging to the "`Fractional`" class, therefore a floating point number, due to the expression `x - 0.001`) and `f` (which turns out to be a function from "`a1`" belonging to the "`Fractional`" class to "`a2`" — also belonging to the "`Fractional`" class — since `f` is applied to `x` and its return value is divided by `0.001`). The type of `computederivative` will therefore be "`(Fractional a1, Fractional a2) => (a2 -> a1, a2) -> a1`", where `a1` and `a2` are type variables with the constraint of representing types of the "`Fractional`" class (floating point numbers).

However, the example presented is not surprising, because something similar to the "`computederivative`" function presented above can be easily implemented even using an imperative language (for example, using the C language one can use a function pointer as the first parameter, instead of f). On the other hand, something considerably more difficult to implement with non-functional languages is a "`derivative`" function which receives only the function f as input (and not the point `x` in which to compute the derivative) and returns a function (and not a real number) that approximates the derivative of f. This `derivative` function has a single parameter, of type `(Fractional a1, Fractional a2) => a1 -> a2` (indicating that `a1` and `a2` are floating point types) and returns a value of type `a1 -> a2`. The type of this function will therefore be `(Fractional a1, Fractional a2) => (a2 -> a1) -> a2 -> a1` and its possible implementation in Haskell is the following:

```
derivative = \f -> (\x -> (f x - f (x - 0.001)) / 0.001)
```

Let's try to better understand this definition of `derivative`: derivative = \f -> basically says that the name "`derivative`" is associated with a function whose parameter is "`f`". The expression that defines how to compute this function is \x -> (f x − f (x − 0.001)) / 0.001 (some parentheses have been added to the above example for readability), which indicates a function of variable `x` computed as $\frac{fx - f(x-0.001)}{0.001}$. Thus, the return value of the function `derivative` is a function (of the "`x`" parameter), which Haskell can identify as a floating point due to the expression $x - 0.001$. This function is computed based on "`f`", which is a parameter of `derivative`. By evaluating the definition, Haskell can infer the type of `f` (floating point to floating point function).

As a further consideration, it should be noted that the `derivative` function can be seen as a "curryfied" version of the `computederivative` function. Basically, instead of 2 arguments (the function `f` to calculate the derivative of and the point `x` in which to calculate the derivative) the function receives only the first argument `f`... When invoked with 2 arguments, the "`derivative`" function returns a real number (the value of the derivative of `f` at the point `x`); therefore, if applied to a single parameter `f` the function cannot return a real, but will return "something" which can become a real number when applied to a further parameter `x` (which is a real number)... This "something" is therefore a function $f' : \mathcal{R} \to \mathcal{R}$.

We recall that (informally speaking) the fundamental idea of currying is to transform a function of two parameters x and y into an equivalent function of the x parameter which returns a function of the y parameter. Thus, the currying mechanism allows us to express a function in several variables as a function in one variable which returns a function of the other variables. For example, a function $f : \mathcal{R}x\mathcal{R} \to \mathcal{R}$ which takes two real parameters and returns a real number can be rewritten as $f_c : \mathcal{R} \to (\mathcal{R} \to \mathcal{R})$.

As another example, the "`sum`" function that sums two numbers

```
sum = \ (x, y) -> x + y
```

can be rewritten as

```
sum_c = \x -> (\y -> x + y)
```

To better understand the currying mechanism, consider $f(x, y) = x^2 + y^2$ and its curryified version $f_c(x) = f_x(y) = x^2 + y^2$. Notice that de domain of $f()$ is $\mathcal{R}^2$ and the codomain of $f()$ is $\mathcal{R}$, while $f_c()$ has $\mathcal{R}$ as a domain and $\mathcal{R} \to \mathcal{R}$ as a codomain. In Haskell these functions are defined as

```
f  = \ (x, y)  -> x * x + y * y
fc = \x -> (\y -> x * x + y * y)
```

`fc` allows partial applications, such as `val g = f 5`, which defines $g(y) = 25 + y^2$, while `f` does not allow anything similar.

The Haskell syntax directly supports currying (thanks to the fact that function application is left-associative); moreover, it is possible to define multi-parameters functions with something like

```
  f a b = exp;
```

which is equivalent to

```
  f = \a -> \b -> exp;
```

Again, this is just syntactic sugar that simplifies the usage of currying for imperative programmers.

Thanks to this simplified syntax, the "`f`" and "`fc`" functions from the previous example can be written as

```
f (x, y) = x * x + y * y;
fc x y   = x * x + y * y;
```

while the definition of "`derivative`" becomes

```
derivative f x = (f(x) - f(x - 0.001)) / 0.001;
```

So, a two-parameters function "f (p1, p2) = ..." can be curryified by changing "`(p1, p2)`" into "`p1 p2`": "f p1 p2 = ...".

3.8 Haskell and Data Types

So far we've been working with Haskell's predefined data types: `()`, `Bool`, `Int`, `Float`, `Double`, `Char` and `String` (although the `()` type hasn't been used much... In practice, it is mainly useful for modeling "strange" functions that take no arguments or that return only effects and not values).

All types recognized by Haskell ("simple types" such as the primitive types just mentioned, or more complex user-defined types) can be aggregated in various ways to generate more complex types. The simplest way to compose data types is through tuples (defined on the cartesian product of the types that compose them). For example, we have already seen how a multi-argument function could be defined as a function whose only argument is a tuple composed of the function's arguments:

```
    sum2 = \ (a,b) -> a * a + b * b
```

This function has type "**Num** a =>(a, a) -> a" which means "function from a pair of elements of the type `a` to an element of the type `a`, where `a` is a type representing a number (`Int`, `Float`, `Double`)"; in other words, the only parameter of the function is of type "(a,a)", which represents the cartesian product of a numerical set by itself (for example, $\mathcal{N} \times \mathcal{N}$).

More formally speaking, a tuple is an ordered set of multiple values, each of which has a type recognized by Haskell. Again, notice how the type `()` (or unit, having a single value `()`) can be seen as a tuple of 0 items, and how tuples of 1 items match the item itself. For example, "`(6)`" is equivalent to "`6`" and has type `Num p => p`; thus, the expression "(6)" is evaluated as "6" and the expression "(6) == 6" evaluates to `True`.

As seen, the pattern matching mechanism applied to tuples allows you to create bindings between several names and symbols at the same time: "pair=("pi", 3.14)" creates a link between the symbol "`pair`" and the pair "("pi", 3.14)" (having type "`Fractional b => ([Char], b)`"). Instead, "(**pi**_name, **pi**_value) = pair" binds the symbol "`pi_name`" to the string "`"pi"`" and binds the symbol "`pi_value`" and the number "`3.14`" (of a "`Fractional`" type). This same mechanism is used to pass multiple parameters to a function using a tuple as the only argument.

Haskell also allows you to define new data types, using the "`data`" keyword, which allows you to make the compiler recognize data whose values are not previously known. The simplest way to use `data` is to define the equivalent of an enumeration type; for instance

```
    data Currency = Eur | Usd | Ounce_gold
```

```
data Currency = Eur | Usd | Ounce_gold
data Money = Euros Float | Usdollars Float | Ounces_gold Float

convert (amount, to) =
    let toeur (Euros x)       = x
        toeur (Usdollars x)   = x / 1.05
        toeur (Ounces_gold x) = x * 1113.0
    in
        (case to of
            Eur        -> toeur amount
            Usd        -> toeur amount * 1.05
            Ounce_gold -> toeur amount / 1113.0
        , to)
```

Figure 3.3: Converting amounts of "`Money`" to different currencies. Notice the pattern matching on the "`Money`" and "`Currency`" types.

```
data Currency = Eur | Usd | Ounce_gold
data Money = Euros Float | Usdollars Float | Ounces_gold Float

convert (amount, to) =
    let toeur (Euros x)       = x
        toeur (Usdollars x)   = x / 1.05
        toeur (Ounces_gold) x = x * 1113.0
    in
        case to of
            Eur        -> Euros       (toeur amount)
            Usd        -> Usdollars   (toeur amount * 1.05)
            Ounce_gold -> Ounces_gold (toeur amount / 1113.0)
```

Figure 3.4: Converting amounts of "`Money`", final version.

defines three new values ("`Eur`", "`Usd`" and "`Ounce_gold`") that were not previously recognized: before this definition, trying to associating the value "`Eur`" to a name (with "c = Eur") results in an error. But after the definition, "c = Eur" succeeds and "`c`" is associated with a value of type "`Currency`".

In this kind of definition, the symbol "`|`" represents an "or" meaning that a variable of type "`Currency`" can have value " `Eur`" or "`Usd`" or "`Ounce_gold`". Note that names separated by "`|`", which represent the constant values we are defining, must start with an uppercase letter, as the type name. Technically, they are *value constructors*, ie functions that return (construct) values of the new type we are defining. In this case (definition of a type equivalent to an enumerated type), the constructors have no arguments and are therefore constant constructors; however, there is the possibility of defining constructors that generate a value starting from an argument.

For example, consider `Money` come segue:

```
data Money = Euros Float | Usdollars Float | Ounces_gold Float
```

(note that "`Euros`, "`Usdollars`" and "`Ounces_gold`" have been used here to make the constructors of the " tt Money" type different from constructors of the "`Currency`" type...).

Like all Haskell functions, these value constructors also have only one argument (and as usual, this limitation can be overcome by using a tuple as an argument).

In this case, "`Euros`" does not represent a (constant) value of the "`Money`" type (as was the case with "`Eur`" for the " `Currency`" type), but it is a function from numbers `Float` to values of the "`Money`" type (in Haskell, "**Float** –> Money "). Values of "`Money`" type are therefore "`Euros 0.5`" or similar.

Figure 3.3 shows a "`convert`" function that converts values between different currencies. The function is implemented using the two new data types just described, doing pattern matching on values of the "`Currency`" type in the "`toeur`" function and doing pattern matching on values of the "`Money`" type in the body of "`convert`". Note however that the use of these new data types is still partial, because this version of "`convert`" receives a pair where the first element is of type "`money`" and the second is of type "`currency`", but returns a "`(Real, Currency)`" pair instead of a value of the "`money`" type. The implementation of Figure 3.4 solves this last problem.

When using "`data`" to define a new data type, it is important to remember that defining a type by simply specifying its values is not enough. We must also specify how to operate on such values, defining functions that operate on the type that we defining (going back to the previous example, define the "`Currency`" and "`Money`" types without defining the "`convert`" function is not very useful...).

3.9 Working with Haskell

An abstract machine for the Haskell language (that is, an abstract machine capable of understanding and executing programs written in Haskell) can be implemented through interpreters or compilers. Although there are several compilers or interpreters for Haskell, here we will focus on the Glasgow Haskell Compiler (ghc) starting with its interactive version, ghci, which implements a read-evaluate-print loop (Real Evaluate Print Loop — REPL) for Haskell. This is because an interactive program like `ghci` is initially more intuitive and easier to use (avoiding us, at least initially, to consider more complex concepts like I/O Monad or similar). This means that operationally a user can interact directly with the REPL via a prompt and test the first commands we will see in a fast and intuitive way. The REPL can be invoked by typing the command "`ghci`", which responds as follows:

```
luca@nowhere   $ ghci
GHCi, version 8.6.5:  http://www.haskell.org/ghc/ :?  for help
Prelude>
```

`ghci` shows its prompt (characterized by the "`Prelude>`" symbol) and waits for expressions to be entered to be evaluated.

As already explained, running a functional program means evaluating the expressions that compose it (see Section 3.1); thus, a Haskell REPL can initially be viewed as an interactive evaluator of expressions.

Expressions entered via the prompt are evaluated by `ghci` as the user enters them; every time `ghci` evaluates an expression, it displays the resulting value on the screen. For example, typing

```
 5
```

the REPL answer is

```
5
```

showing that the expression we enteres has been evaluated to 5. Of course, more complex expressions can also be tested:

```
Prelude> 5 + 2
7
Prelude> 2 * 3 * 4
24
Prelude> (2 + 3) * 4
20
```

and so on.

The "Return" (or "Enter") key works as a *terminator* marking the end of an expression. Thus, an expression is not compiled/evaluated/reduced until a carriage return is encountered; on the other hand, a carriage return results in immediate evaluation of the expression and this could lead to problems with multi-line expressions. For example, the following Haskell code declares and defines an integer variable called "`a`" and initializes it with the value 5:

```
a :: Int
a = 5
```

however, typing these two lines in the `ghci` REPL we get an error after the first line:

```
Prelude> a :: Int

<interactive>:4:1: error: Variable not in scope: a :: Int
```

because `ghci` tries to compile "`a :: Int`" before "`a = 5`" is entered. A possible solution to this issue is to type the variable declaration and definition on the same line, separated by "`;`":

```
Prelude> a :: Int; a = 5
Prelude> a
5
Prelude> :t a
a :: Int
```

as a side note, this example also shows that `ghci` interprets the "`:t`" command as a request to show the type of a variable. If the declaration of the type of "`a`" was not entered, the result would have been

```
a = 5
Prelude> a
5
Prelude> :t a
a :: Num p => p
```

indicating that the value associated to the "`a`" identifier is has a generic type "`p`" of the "`Num`" class (`Int`, `Float`, `Double`).

The previous examples show how given an expression, the compiler may be able to infer the type of the variables (but also of the arguments and the return value of the functions). Unlike other languages like Standard ML, Haskell uses its polymorphic type system to give more flexibility without performing automatic type conversions. For this reason, expressions like

```
 5 + 2.0
```

are perfectly valid:

```
Prelude> :t 5
5 :: Num p => p
Prelude> :t 2.0
2.0 :: Fractional p => p
Prelude> 5 + 2.0
7.0
Prelude> :t 5 + 2.0
5 + 2.0 :: Fractional a => a
```

This indicates that `5` is a generic number, while `2.0` is a floating point number. The result of the sum will therefore be a floating point number (a type belonging to the "`Fractional`" class). The `+` (sum) operator accepts as parameters two values of the same type, provided that this type belongs to the "`Num`" class. Since the "`Fractional`" class is a subset of the "`Num`" class, the sum is possible and its result has a type belonging to the "`Fractional`" class.

As previously mentioned, Haskell provides several base types:

```
Prelude> :t 2
2 :: Num p => p
Prelude> :t 2.0
2.0 :: Fractional p => p
Prelude> :t 2 > 1
2 > 1 :: Bool
Prelude> :t "abc"
"abc" :: [Char]
Prelude> :t 'a'
'a' :: Char
```

We are now ready to do the first thing that every programmer does when facing with a new programming language:

```
> "Hello, " ++ "world"
"Hello, world"
```

`"Hello, "` and `"world"` are two values of the "`String`" type (which is a list of characters "`[Char]`"), while "++" is the string concatenation operator, which receives two string as parameters and evaluates to a string representing their concatenation.

So far we've seen how to use `ghci` to evaluate expressions where all values are explicitly expressed, but Haskell also allows assigning **names** to "entities" recognized by a language. So let's see how to associate names to entities in Haskell and what are the denotable entities in Haskell (in an imperative language, denotable entities are variables, functions, data types, ...) . Haskell provides the concept of *environment* (a function which associates names with denotable values) but there is no concept of memory (function which associates each variable the value it contains)[8]; thus, it is possible to map names to values, but it is not possible create mutable variables:

```
Prelude> n = 5
Prelude> n
5
Prelude> :t n
n :: Num p => p
```

These commands bind the "`n`" identifier to "5", print the value bound to "`n`", and print the type of "`n`". Of course, it is possible to bind a variable name to values computed using more complex expressions:

```
Prelude> x = 5.0 + 2.0
Prelude> n = 2 * 3 * 4
```

After associating a name to a value, it is possible to use such a name (in place of the value) in the following expressions:

```
Prelude> x = 5.0 + 2.0
Prelude> y = x * 2.0
Prelude> x > y
False
```

notice that "y = x * 2" would not have resulted in any error... Why?

Haskell also allows to define *functions*, which associate names to blocks of code. While in imperative languages variables and functions are defined using different constructs, in functional languages (and not only) there is a "function" data type, generated by lambda expressions similar to $\lambda x.e$ (abstraction of the λ-calculus). In particular, Haskell uses the "\" symbol instead of the "λ" symbol and "`->`" (an arrow) instead of ".":

```
Prelude>:t \x -> x + 1
\x -> x + 1 :: Num a => a -> a
```

in this case the value resulting from the evaluation of the expression is a function (in the mathematical sense of the term) from a numeric type "`a`" to itself. Since no definition has been used, this function has not been given a name (so, it is named *anonymous function*). However, a function can be applied to data without giving it a name:

```
Prelude> (\x -> x + 1) 5
6
```

`ghci` is generally able to infer the type of a function, as for all the other data types:

```
Prelude> :t \x -> x + 1
\x -> x + 1 :: Num a => a -> a
Prelude> :t \x -> x + 1.0
\x -> x + 1.0 :: Fractional a => a -> a
```

At this point, it should be clear how to associate a name to a function, through what in other languages would be called a function definition and which in Haskell corresponds to a simple variable definition. Technically, the following code defines a variable "`mul2`" which has type "function from numbers to numbers":

[8]Theoretically, the concept of memory / modifiable variable also exists in haskell, but we won't deal with it.

```
fact_tr = \n -> \res -> if n == 0 then res else fact_tr (n - 1) (n * res)
fact = \n -> fact_tr n 1

fact_tr1 0 res = res
fact_tr1 n res = fact_tr1 (n - 1) (res * n)
fact1 n = fact_tr1 n 1
```

Figure 3.5: Fattoriale con ricorsione in coda.

```
Prelude> mul2 = \n -> 2 * n
Prelude> :t mul2
doppio :: Num a => a -> a
Prelude> mul2 9
18
Prelude> mul2 4.0
8.0
```

Haskell also provides a simplified syntax for function definition:

```
Prelude> mul2 n = 2 * n
Prelude> mul2 9
18
```

The simplified "`name a = ...`" syntax looks more intuitive than the explicit definition "`name = \a -> ...`", but is equivalent to it.

Combining what we have seen so far with the "`if`" conditional expression, it is possible to define even complex functions (equivalent to iterative algorithms implemented with an imperative language) via recursion. For instance,

```
Prelude> fact = \n -> if n == 0 then 1 else n * fact (n - 1)
Prelude> fact 5;
120
```

Using the simplified syntax, it is possible to define recursive functions based on the inductive base and the inductive step:

```
Prelude> :{
Prelude| fact 0 = 1
Prelude| fact n = n * fact (n - 1)
Prelude| :}
Prelude> fact 4
24
```

Based on what has been discussed so far, it is possible to implement the following recursive functions in Haskell:

- Compute the factorial of a number *using tail recursion* (Figure 3.5)
- Compute the greatest common divisor between two numbers (Figure 3.6)
- Solution to the Tower of Hanoi problem (Figure 3.7 and 3.8)

As explained earlier in this document, in addition to allowing you to define and evaluate expressions (possibly associating expressions or values with names, through the environment concept), Haskell allows you to define and use new *data types*. In particular, the `data` construct allows you to define new data types by specifying the constructors that generate their variants. For example, one could define a type `Color`, which represents a component of red, green, or blue (with the intensity expressed as a real number):

```
Prelude>:t Red

<interactive>:1:1: error: Data constructor not in scope: Red
```

```
gcd = \a -> \b -> if b == 0 then a else gcd b (a `mod` b)

gcd1 a b = if b == 0 then a else gcd1 b (a `mod` b)

gcd2 a 0 = a
gcd2 a b = gcd2 b (a `mod` b)
```

Figure 3.6: Massimo Comun Divisore.

```
move n from to via =
    if n == 0
      then
        "\n"
      else (move (n - 1) from via to) ++
        "Move disk from " ++
        from ++ " to " ++ to ++
        (move (n - 1) via to from)
```

Figure 3.7: Torre di Hanoi.

```
Prelude> :t Red 0.5

<interactive>:1:1: error:
    Data constructor not in scope: Red :: Double -> t
Prelude> data Color = Red Float | Blue Float | Green Float
Prelude> :t Red
Red :: Float -> Colore
Prelude> :t Red 0.5
Red 0.5 :: Color
```

Before defining "**data** Color = Red **Float** Blue Float — Green Float—", the "`Red`" identifier is not understood by `ghci`, which complains about using the "`Red`" constructor without defining it (see the first two errors); after the definition, the "`Red`" identifier is correctly recognised as a data constructor for the "`Color`" type (in this case, the constructor is a function from `Float` to `Color`).

Finally, it should be noted that `ghci` may seem non very user-friendly, because entering multi-line programs is not that easy, even using "`:{`' ' and "`:}`". This issue can be addressed by editing complex Haskell programs in text files (usually with the extension `.hs`), taking advantage of the features of advanced editors such as `emacs`, tt vi, `gedit` or similar. A program contained in a text file can then be loaded in `ghci` using the "`:l`" directive: "**Prelude**> :l < file>.hs"

As an example of using `:l`, you can insert the program of Figure 3.8 into the file `hanoi.hs`, using your favorite text editor (for example `vi`). The program can then be loaded into `ghci` with

```
Prelude> :l hanoi.hs
[1 of 1] Compiling Main             ( hanoi.hs, interpreted )
Ok, one module loaded.
```

```
move n from to via =
    if n == 1
      then
        "Move disk from " ++ from ++ " to " ++ to ++ "\n"
      else
        (move (n - 1) from via to) ++
        (move 1 from to via) ++
        (move (n - 1) via to from)
```

Figure 3.8: Torre di Hanoi, versione alternativa.

```
*Main> putStrLn  (move 3 "Left" "Right" "Center")
Move disk from Left to Right
Move disk from Left to Center
Move disk from Right to Center
Move disk from Left to Right
Move disk from Center to Left
Move disk from Center to Right
Move disk from Left to Right
```

As shown in the previous example, the "`move`" function is now defined as if it had been entered directly from the keyboard (about the example, notice how the "`putStrLn`" function has been used to display the string, in order to correctly handle newline characters).

Using `:l` is also useful for debugging, because it reports syntax errors more accurately, indicating the line of the program where the error occurred.

Chapter 4

Recursive Data Types

4.1 Data Types

Various programming languages allow users to define new data types by defining the set of valid values and the operations that can be performed on these values.

An example of user-defined data types (perhaps the simplest case of a user-defined type) is represented by enumerative data types. For example, you can define a "`color`" type with values "`red`", "`blue`", and "`green`" and then define the various operations you can perform on those values. This way of defining new data types is clearly not practical when the number of values is very large (and does not allow defining new data types when the number of values is not finite). To address this issue, the concept of enumerative type can be generalized by using *data constructors* instead of simple constant values like "`red`", "`blue`", and "`green`". The new data type that is going to be defined therefore has values generated by one or more constructors, which operate on one or more arguments. It is clear that a constructor having an argument of integer type can generate a large number of values. Also note that the "`red`", "`blue`", and "`green`" values mentioned above are nothing more than special cases of data constructors[1].

The values of the new data type are then partitioned into various subsets, called *variants*; in other words, the set of possible values (definition set of the data type) is the disjoint union of the variants of the data type. Each variant is associated with a constructor, which generates all the values composing the variant; in other words, a variant is the set of values generated by a single data constructor. Note that if we assume that two different constructors can never generate the same value, we can say that the definition set of the data type is the union (or sum between sets) of the variants (instead of "disjoint union"). It will become clear later on why the concept of "sum" is interesting in this context.

Using the Standard ML syntax, it is possible to define a new data type in a very generic way with

```
datatype <name> = <cons1> of [type1] | <cons2> of [type2] | ... ;
```

where the "`<cons1>`", ... "`<consn>`" constructors represent the type variants. A similar definition in Haskell would be

```
data <name> = <cons1> [type1] | <cons2> [type2] | ... | <consn> [typen]
```

(notice that for Haskell the data type names and the constructor names must start with a capital letter). Note that constructors are real functions that take an argument and map it into a value of the new data type. In some languages like Standard ML, if you need multiple arguments for a constructor you can use a tuple argument; in other languages like Haskell a constructor can return a function that returns the value of the new data type (or a function again...), using the currying technique. Haskell provides a simplified syntax for these situations, however, which makes valid definitions like

```
data Test = Constructor1 Int Double | Constructor2 Char String
```

or similar. In this case, "`Constructor1`" is not a function with two parameters (as a naive reader might think), but a function accepting one single parameter (of type "`Int`") and returning a function from "`Double`" to "`Test`". "`Constructor1`" has hence type "`Int -> Double -> Test`" (which means "`Int -> (Double -> Test)`").

The example in Figure 4.1 shows how to define a "`Num`" data type which can have integer or floating point values.

[1] In particular, they are constructors that take no input arguments - or have 0 arguments. Thus, each constructor generates a unique value, which is identified with the name of the constructor.

```
    data Num = Integernum Int | Realnum Double
    sumnumbers (Integernum a, Integernum b) = Integernum (a + b)
    sumnumbers (Integernum a, Realnum    b) = Realnum
(fromIntegral a + b)
    sumnumbers (Realnum    a, Integernum b) = Realnum
(a + fromIntegral b)
    sumnumbers (Realnum    a, Realnum    b) = Realnum    (a + b);
      ...
```

Figure 4.1: Example of data type with two constructors (receiving arguments of different types), in Haskell.

As previously mentioned, a constructor receiving more than one argument as input can use a tuple to group its arguments, thus realizing a *Cartesian product* among the argument definition sets[2]. New data types can then be created from existing data types using (disjoint) unions and Cartesian products. More simply, we can speak of *sums or products* between existing data types.

In this way, an algebra, or algebraic structure, of data types is defined, consisting of a support set - the set of data types - with a sum operation and a product operation defined on it. The name "*Algebraic Data Types*" (ADT) is hence often used to indicate the data types defined in this way.

4.2 Recursion

The *recursion* technique (closely related to the mathematical concept of *induction*) is used in computer science to define some kind of "entity"[3] based on itself. The typical example is represented by recursive functions: a function $f()$ can be defined by expressing the value of $f(n)$ as a function of other values computed by $f()$ (typically, $f(n-1)$). But a similar idea can also be used to define a set by describing its elements based on other elements contained in the set (example: if the element n belongs to the set, then $f(n)$ also belongs to the set).

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 elements starting from existing elements. The basis is a clause of the recursive definition that does not refer to the "entity" being defined (for example: "the factorial of 0 is 1", or "0 is a natural number", or "1 is a prime number", etc...) and has the task of ending the recursion. Without an inductive basis, an endless recursion happens (and this is not useful from a practical point of view).

It is known how it is possible to use recursion to define functions (and how recursion is the only way to implement loops in functional programming languages - in other words, recursion can be considered as a sort of "functional equivalent" of iteration). In essence, 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 details, 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 $\mathcal{X}$. As an inductive basis, we define the value of the function for the smallest value belonging to the domain (for example, $f(0) = a$, with $a \in \mathcal{X}$) and as an inductive step the value of $f(n+1)$ is defined based on $f(n)$. As mentioned above, this can be done by defining $f(n+1) = g(n, f(n)))$. Note that the domain of $g()$ is the set of pairs of elements taken from the domain and codomain of $f()$, while the codomain of $g()$ is equal to the codomain of $f()$.

It is important to note that the mathematical concept of induction can be used to define not only functions but also sets, properties of numbers, etc... For example, a set can be defined by induction by indicating one or more elements that belong to it (inductive basis) and by defining new elements of the set starting from elements belonging to it (inductive step). Using this technique the set of natural numbers can be defined according to the *Peano axioms*:

- Inductive base: 0 is a natural number ($0 \in \mathcal{N}$)

[2]In some languages, such as Standard ML, a data constructor is restricted to having at most one argument, so the use of the Cartesian product is mandatory. Other languages, such as Haskell, provide a syntax to create multi-argument constructors. In this second case the Cartesian product is hidden by the syntax of the language, but conceptually it is always present.

[3]The term "entity" here is used informally to generically indicate functions, sets, values... But, as we will see shortly, also data types!

```
i2n 0 = Zero
i2n x = Succ (i2n (x - 1))

n2i Zero     = 0
n2i (Succ n) = 1 + n2i n

nsum Zero     n = n
nsum (Succ m) n = Succ (nsum m n)
```

Figure 4.2: Various functions working on Peano encoding of the natural numbers (defined using a recursive data type).

- Inductive step: $n \in \mathcal{N} \Rightarrow n + 1 \in \mathcal{N}$ (that is, it is possible to define a function $s : \mathcal{N} \to \mathcal{N} - \{0\}$ computing the successor of a natural number)

4.3 Recursive Data Types

As mentioned, to define a data type it is necessary to define its definition set (set of values belonging to the data), plus the operations that can be applied to the values of the type. Since a set can be defined by induction (as seen), one can think of applying recursion to define new data types.

Recall that the values of a data type can be generated by constructors that take one or more arguments as input (actually, a n-tuple of arguments), it becomes clear how to apply recursion to types data: a constructor for a data type `T` can have an argument of type `T` (or, a tuple containing an element of type `T`). Clearly, an inductive basis is needed to avoid infinite recursion; this means that one of the variants of the type must have a constructor that has no arguments of type `T`. As an inductive step, there may be other variants with constructors that have arguments of type `T`.

For example, it is possible to define the type `natural` based on the recursive definition of the set of naturals given by Peano: the type will consist of two variants, one of which (the inductive basis) will have a constant constructor `zero` (representing the number 0), while the other will have a `succ` constructor, which generates a natural starting from a natural. Using Haskell's `data` construct, this would be

```
data Natural = Zero | Succ Natural
```

To complete the definition of the "`Natural`" data type, some simple operations of the values of this type must be defined. For example, Figure 4.2 shows how to implement a conversion from "`Int`" to "`Natural`" ("`i2n`" function), a conversion from "`Natural`" to "`Int`" ("`n2i`" function), and the sum of two natural numbers ("`nsum`" function).

Curious readers can check a possible C++ implementation of the Peano coding of natural numbers, shown in Figure 4.3. From this simple example (the implementation of the "`i2n`" function and the "`+`" operator are missing, but they are quite simple) one can immediately observe an interesting thing: since in C++ the data constructor associated with a class has the same name as the class, to have a "`zero()`" constructor and a "`succ()`" constructor it was necessary to derive two "`zero`" and "`succ`" classes from the "`natural`" abstract class. This suggests that a data type with multiple variants can be implemented in C++ by using an abstract base class representing the data type, from which a class for each variant / constructor is derived.

Another interesting thing to note is the usage of *references* to account for the recursive nature of the "`natural`" data type. This begins to suggest that there is a relationship (which will become clear as we look at lists) between recursive datatypes and references / pointers: languages that do not explicitly support recursion on datatypes are forced to introduce the pointer or reference data type, while languages that don't support pointers / references use recursive data types to avoid losing expressive power.

Finally, the "`const`" keyword is used to guarantee the immutability of the data structures encoding the natural numbers.

Although interesting from a mathematical point of view, the definition of this "`Natural`" data type is not too useful... However, there are a number of data structures that can be defined as recursive types and they are very useful: they are of lists, trees, and similar dynamic data structures. For example, a list of integers is recursively definable as follows: a list is empty or it is the concatenation of an integer and a list. The "empty list" constructor represents the inductive basis, while the "integer and list concatenation" constructor represents the inductive step. Note that a list is a recursive data structure

```
class natural {
  public:
    virtual unsigned int n2i(void) const = 0;
};

class zero: public natural {
  public:
    zero(void) {
    };
    virtual unsigned int n2i(void) const
    {
      return 0;
    };
};

class succ: public natural {
    const natural &prev ;
  public:
    succ(const natural &p) : prev(p)
    {
    };
    virtual unsigned int n2i(void) const
    {
      return prev.n2i() + 1;
    };
};
```

Figure 4.3: Example showing a possible C++ implementation of the Peano encoding of natural numbers.

because the "integer and list concatenation" constructor takes values of type list as its second argument. Using Haskell syntax, a list of integers can therefore be defined as

```
data List = Empty | Cons Int List
```

where `Empty` is the constructor of an empty list, while `Cons` generates the variant containing all non-empty lists. Note that the `Cons` constructor has an argument of type `Int` and returns a function `List -> List`, because a Haskell function can't have more than one argument (thus, currying is used to support multiple arguments). However, "`Cons Int List`" could also be seen as a function that takes two arguments: an integer and a list.

4.4 Immutable and Mutable Lists

The "`List`" recursive data type introduced in the previous section represents a very special type of list, called an "immutable list". The reason for this name can be understood by considering the operations implemented on this type. Since immutable lists (and immutable data structures in general) are mainly used in functional languages, the Haskell language will be used in the next examples (but the examples are easily translatable into other functional languages, such as those of the ML family).

The first two operations on the "`List`" data type are the two constructors `Empty` and `Cons`, which respectively generate empty lists and non-empty lists (that is, concatenations of integers and lists). To operate on immutable lists in a generic way, two other operations traditionally called "`car`" and "`cdr`" are needed. Given a non-empty list, "`car`" returns the first integer of the list (the so-called "head of the list"), while "`cdr`" returns the list obtained by removing the first element. Both "`car`" and "`cdr`" are undefined for empty lists. A simple implementation of these two functions is given by:

```
car (Cons v _) = v
cdr (Cons _ l) = l
```

Note that these definitions are not exhaustive: both "`car`" and "`cdr`" are in fact defined using pattern matching on a value of the "`List`" type, but the only pattern present in the definition matches non-empty

```
isempty Empty = True
isempty _     = False

list_len Empty      = 0
list_len (Cons _ l) = 1 + list_len l

concat Empty b      = b
concat (Cons e l) b = Cons e (concat l b)
```

Figure 4.4: Example of Haskell functions working o immutable lists, using only `Empty`, `Cons`, `car`, and `cdr`.

```
ins = \n -> \l ->
     if ((l == Empty) || (n < car l))
       then
         Cons n l
       else
         Cons (car l) (ins n (cdr l))
```

Figure 4.5: Ordered insert into a list, implemented in Haskell.

lists ("`Cons`" constructor)... There is no pattern that matches empty lists ("`Empty`" constructor). This means that if "`chr`" or "`cdr`" is applied to an empty list, an exception is thrown (not really "purely functional"). This all reflects the fact that "`car`" and "`cdr`" are not defined for empty lists.

Every operation on lists can be implemented based on `Empty`, `Cons`, `car` and `cdr` only. For instance, Figure 4.4 shows how to check if a list is empty, compute its length, or concatenate two lists using only the primitives just mentioned. Note that many of the functions that act on lists are recursive (since lists are defined as a recursive data type, this shouldn't be too surprising).

The "`isempty`" function is very simple and returns a boolean value based on pattern matching: if the list passed as an argument matches a list generated by the `Empty` constructor (that is, if it is a empty list), returns `True` otherwise (wildcard pattern) returns `False`.

The "`list_len`" function works recursively, using the "`Empty`" pattern as an inductive basis (the length of an empty list is 0) and saying that if the length of a list `l` is n, then the length of the list obtained by inserting any integer in front of `l` is $n + 1$ (inductive step).

Finally, the `concat` function uses the concatenation of an empty list with a generic list "`b`" as an inductive basis (by concatenating an empty list with a list "`b`", we get `b`). The inductive step is based on the fact that concatenating a list composed of an integer `n` and a list `l` with a list `b` is equivalent to creating a list composed of the integer `n` followed by the concatenation of `l` and `b`.

Finally, a function to insert an integer into a sorted list can be implemented as shown in Figure 4.5. When we want to insert a new element in an empty list (`l = Empty`), the resulting list is created (via `Cons`) by adding the new element to the head of the list. This is also done if the element to be inserted is smaller than the first element of the list (`n < car l`). Otherwise, a new list is created (again via `Cons`) by concatenating the head of `l` (`car l`) with the list created by inserting the number into the remainder of `l` (`insert n (cdr l)`). In any case, a list containing `n` inserted in the right position is created by one or more invocations of `Cons`, and not by modifying the list `l`.

For example, consider what happens when you invoke

```
ins 5 (Cons 2 (Cons 4 (Cons 7 Empty)))
```

since $5 > 2$, this "`ins`" expression evaluates to

```
Cons 2 (ins 5 (Cons 4 (Cons 7 Empty)))
```

and

```
Cons 2 (Cons 4 (ins 5 (Cons 7 Empty)))
```

which finally stops the recursion evaluating to "Cons 2 (Cons 4 (Cons 5 (Cons 7 Empty)))". As a result, 3 new values of type "`List`" have been created.

```
struct list {
  int val;
  struct list *next;
};

struct list *empty(void)
{
  return NULL;
}

struct list *cons(int v, struct list *l)
{
  struct list *res;

  res = malloc(sizeof(struct list));
  res->val = v;
  res->next = l;

  return res;
}

int car(struct list *l)
{
  return l->val;
}

struct list *cdr(struct list *l)
{
  return l->next;
}
```

Figure 4.6: Example of immutable lists in C.

To better understand the behavior of an immutable list and the reason for its name, it is useful to see how it can be defined in a language that does not directly support recursive data types, such as the C language. In this case, pointers are used to connect the various elements of the list: in particular, each element of the list is represented by a structure composed of an integer field (the value of that element) and a pointer to the next element of the list. So, instead of having a "`list`" data type defined based on itself we have a "`list`" data type defined using a pointer to "`list`". The two "`empty()`" and "`cons()`" constructors are implemented as functions that return a pointer to "`list`" (these functions therefore dynamically allocate the memory necessary to contain a "`list`" structure) and the "`car()`" and "`cdr`" functions simply return the values of the fields of the "`list`" structure. A list is terminated by an element which has a pointer to the next element equal to `NULL` (this can be considered the equivalent of the inductive basis). In summary, a simple C implementation might look like Figure 4.6. Note that to simplify the implementation the result of `malloc()` is not checked (a more "robust" implementation should instead check that `malloc()` does not return `NULL`).

The "`insert()`" function can then be defined in the same way as in Haskell, using only the "`empty()`", "`cons()`", "`car()`" and "`cdr()`" functions as shown in Figure 4.7. This definition uses the *arithmetic if* operator, which is equivalent to Haskell's `if...then...else...end` expression but is perhaps less readable compared to a "traditional" selection construct. The implementation of Figure 4.8 is equivalent (albeit unstructured and "less purely functional").

Now, consider what happens when invoking `l = ins(5, l);` if `l` is a pointer to the list in Figure 4.9, containing the values 2, 4, and 7. Since 5 > 2, `ins()` recursively invokes `ins(5, cdr(l));`, with `cdr(l)` being a pointer to the second element of the "`l`" list (it is therefore a pointer to a list containing the values 4 and 7). The result of this recursive invocation of `ins()` will then be passed to `cons()` (which will dynamically allocate a new structure of type "`list`"). Since `car(cdr(l))` returns 4 and 5 > 4, `ins(5, cdr(l));` will again recursively invoke `ins()`, with parameters 5 and `cdr(cdr(l))`. Now, since

```
struct list *ins(int n, struct list *l)
{
  return ((l == empty()) || (n < car(l))) ?
                  cons(n, l) : cons(car(l), ins (n, cdr(l)));
}
```

Figure 4.7: Ordered insert into an immutable list, implemented in C.

```
struct list *ins(int n, struct list *l)
{
  if ((l == empty()) || (n < car(l))) {
    return cons(n, l);
  } else {
    return cons(car(l), ins(n, cdr(l)));
  }
}
```

Figure 4.8: Ordered insert into an immutable list, implemented in C without arithmetic if.

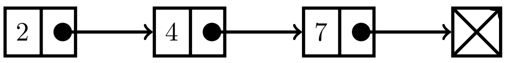

Figure 4.9: Example of list containing the integers 2, 4 e 7.

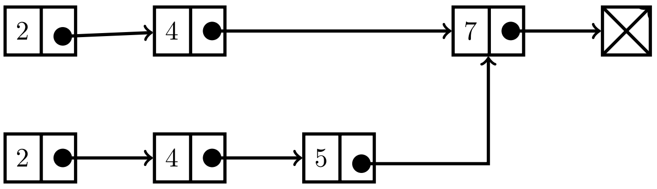

Figure 4.10: List of Figure 4.9 after ins(5, l);.

`car(cdr(cdr(l))))` returns 7 and 5 < 7, ins(5, cdr(cdr(l))); will return `cons(5, cdr(cdr(l)))`, dynamically allocating a structure of type "`list`". `cons()` will then be invoked 2 more times, and as a result `ins(5, l);` will return a list consisting of 3 new dynamically allocated elements (`cons()` has been called 3 times), without modifying any element of the list of Figure 4.9. This is why this kind of list is called "immutable": the `val` and `next` fields of the `list` structure are set when the structure is allocated (from `cons()`) and are never modified. The final result is visible in Figure 4.10, which shows below the 3 new dynamically allocated elements (note that now the pointer `l` points to the element of value 2 on the bottom).

It is also important to note that in the previous example any reference to the first element of `l` may have been "lost", leaving memory areas that are dynamically allocated but no longer reachable: if a program invokes

```
l = vuota();
l = ins(4, l);
l = ins(2, l);
l = ins(7, l);
l = ins(5, l);
```

the data structures that have been dynamically allocated by the second invocation of `ins()` (that is, `ins(2, l);`) no longer have any pointers "reaching" them.

Returning to the previous example, when `l = ins(5, l)` is invoked, the structures containing the first two elements of the "old list" may therefore no longer be reachable, but the memory in which they are stored has not been released by any call to `free()`. This clearly indicates that this implementation of immutable lists is likely to generate *memory leaks*, so some kind of garbage collection mechanism is needed.

```
struct list {
  int val;
  struct list *next;
};

struct list *ins(int n, struct list *l)
{
  struct list *res, *prev, *new;

  new = malloc(sizeof(struct list));
  new->val = n;

  res = l;
  prev = NULL;
  while ((l != NULL) && (l->val < n)) {
    prev = l;
    l = l->next;
  }
  new->next = l;
  if (prev) {
    prev->next = new;
  } else {
    res = new;
  }

  return res;
}
```

Figure 4.11: Traditional implementation of linked lists in C, without using immutable data structures.

The previous implementation of immutable lists in C can be compared with a "more traditional" implementation based on mutable data structures, shown in Figure 4.11.

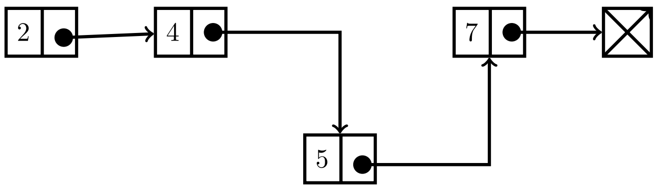

Figure 4.12: List of Figure 4.9 after ins(5, l); using "traditional" mutable lists.

Note that this implementation of the "`ins()`" function modifies the `next` field of the previous element to the one being inserted, so the list is not immutable. On the other hand, it doesn't suffer from the memory leak problems highlighted for the previous implementation (essentially, this tells us that garbage collection techniques are only necessary if immutable data structures are used). The result of l = insert(5, l); using this implementation of lists is visible in Figure 4.12 (note that the arrow going out of element "4" has changed).

Finally, for the sake of a comparison Figure 4.13 shows an implementation of an immutable list in Java (remember that a Java VM implements a garbage collector by default, so memory leaks due to the use of immutable data structures are not a problem).

It is interesting to note that in Java the `final` qualifier allows you to explicitly specify that the fields of the class will never be modified (the data structure is immutable). Also, note that Java (like many other object-oriented languages) forces you to use the same name for constructors and for the class. Therefore, there will not be two "`Empty`" and "`Cons`" constructors for the two variants, but two constructors both called "`List`" which are distinguished by the number and type of their arguments (one constructor - that of empty list - has no arguments, while the other - the one corresponding to `Cons` - has one argument of type `int` and one of type `List`).

```java
public class List {
    private final Integer val;
    private final List next;

    public List(){
        val=null;
        next=null;
    }
    public List(int v, List l) {
        val = v;
        next = l;
    }
    public int car() {
        return val;
    }
    public List cdr() {
        return next;
    }


    public void printList() {
        if (next!=null) {
            System.out.println(val);
            next.printList();
        }
    }

    public List ins(int n){
        if (next==null || n < val)
        return new List(n,this);
        else return new List(car(),cdr().ins(n));
    }

    public static void main(String a[]) {
        List l = new List();
        l = l.ins(1);
        l = l.ins(5);
        l = l.ins(3);
        l = l.ins(9);
        l = l.ins(7);
        l.printList();
    }
}
```

Figure 4.13: Java implementation of immutable lists.

Chapter 5

Fixpoint and Similar Amenities

5.1 Fixed Point Combinators

In general, a combinator is a higher-order function (function that takes other functions as arguments and/or returns functions as a result) with no free variables (that is, all variables used in the combinator are bound in its local environment: they are therefore local variables or parameters)[1].

The *fixed point combinators* (fixpoint combinators) are particularly important in the context of functional programming (and its theoretical foundation, λ calculus). A fixed point combinator is a combinator that computes the *fixed point* of its argument. In other words, if g is a function, a fixpoint combinator is a function Fix with no free variables such that $Fix(g) = g(Fix(g))$.

Note that defining a generic higher order function Fix that computes the fixed point of the function passed as an argument is quite easy: by definition

$$Fix(g) = g(Fix(g))$$

and this expression can also be seen as a definition of Fix if we consider a small extension of the λ calculus that allows us to associate names with λ expressions

$$Fix(g) = g(Fix(g)) \Rightarrow Fix = \lambda g.g(Fix(g)). \tag{5.1}$$

As a first consideration, it is interesting to note that if we try to evaluate the equation 5.1 using an *eager* strategy (evaluation by value), we get

$$Fix(g) = (\lambda g.g(Fix(g)))g \rightarrow_\beta g(Fix(g)) = g((\lambda g.g(Fix(g)))g) \rightarrow_\beta g(g(Fix(g))) = ...$$

and the reduction diverges. This happens because an eager evaluation strategy will always evaluate the innermost expression "$Fix(g)$" by expanding it to "$g(Fix(g))$" and so on... Using instead a *lazy* (evaluation by name) strategy, "$Fix(g)$" is evaluated only when g actually invokes it recursively (hence, it is not evaluated when arriving at the inductive basis... This guarantees that if if the various recursive invocations lead to the inductive basis then the evaluation of $Fix(g)$ does not diverge). This observation is important when trying to implement a fixed point combinator in eager languages Standard ML.

Another important observation is that the "Fix" function defined in Equation 5.1 allows (by construction) to compute the fixed point of its argument g, but it is not a combinator: in particular, the definition of Fix uses the "Fix" variable which is free, not being bound by any λ (while the definition of a combinator should contain only bound variables). This is especially relevant because "Fix" is just the name of the function being defined, so the definition of "Fix" is recursive. In other words, we just moved the recursion from the definition of g to the definition of Fix.

There are many different fixed point combinators that can be used to compute the fixed point of a function without using explicit recursion either in the definition of the function or in the definition of the combinator. The existence of fixed point combinators has a remarkable theoretical importance (in practice, it shows that "pure" λ-calculus - without environment or extensions that allow to associate names to expressions - can implement the recursion and is Turing complete). From a practical point of view, it means instead that a functional language that does not implement "`val rec`" (or the equivalent "`fun`"), "`let rec`' ' or similar...) may still allow the implementation of recursive functions!

[1]Remember that in the particular case of λ calculus a combinator is defined as a λ expression which does not contain free variables, consistently with this more generic definition.

The most famous of the fixed-point combinators is the **Y combinator**, developed by Haskell Curry (yes, the names are always the same at the end...), whose definition (using the λ-calculus) is:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)). \tag{5.2}$$

It is very important (especially when searching for information on the internet!) to notice that in the literature there are some small terminological inconsistencies: while in general the Y combinator is *a particular* fixed point combinator, some people tend to use the term "Y combinator" to identify a generic fixed point combinator (and thus write that there is an infinity of Y combinators).

5.2 Haskell Implementation

The Y Combinator is defined as

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Trying a simple conversion from the λ-calculus syntax to the Haskell syntax ("λ" becomes "\" and "." becomes "`->`") we obtain

```
    y = \f -> (\x -> f (x x))(\x -> f (x x))
```

However, this expression is not accepted by a Haskell compiler or interpreter. For example, `ghci` generates the following error:

```
Prelude> y = \f -> (\x -> f (x x))(\x -> f (x x))

<interactive>:1:23: error:
    o Occurs check: cannot construct the infinite type: t0 ~ t0 -> t
      Expected type: t0 -> t
        Actual type: (t0 -> t) -> t
    o In the first argument of 'x', namely 'x'
      In the first argument of 'f', namely '(x x)'
      In the expression: f (x x)
    o Relevant bindings include
        x :: (t0 -> t) -> t (bound at <interactive>:1:13)
        f :: t -> t (bound at <interactive>:1:6)
        y :: (t -> t) -> t (bound at <interactive>:1:1)
...
```

The meaning of this error becomes clear when trying to figure out the type of "`x`". Assuming that "`x`" has type "`t0`", we have:

- Since "`(xx)`" indicates that "`x`" is *a function* applied to an argument, the type "`t0` " of "`x`" must be a function. Let us consider the more generic case: a function $\alpha \to \beta$, therefore $\texttt{t0} = \alpha \to \beta$
- On the other hand, the function "`x`" is applied to the argument "`x`". So the type α of the function argument "`x`" must be equal to the type of "`x`" ("`t0`"): $\texttt{t0} = \alpha$.

Putting it all together you get

$$\texttt{t0} = \alpha \to \beta \land \texttt{t0} = \alpha \Rightarrow \alpha = \alpha \to \beta$$

which defines the type "`t0`" recursively (as indicated by the error message, "`t0`" must be equivalent to "`t0 -> t`"). Since Haskell supports recursive data types, one would think that it should be able to support the "`t0`" type we are talking about. However, a recursion like $\alpha = \alpha \to \beta$ doesn't make much sense in a language performing strict type checks like Haskell. To understand why, it is necessary to go into a little more details about recursive data types.

When faced with a definition such as $\alpha = \alpha \to \beta$, we speak of *equi-recursive data types* (equi-recursive data types) if α and $\alpha \to \beta$ represent the same data type (same values, same operations on values, etc...). We speak instead of *iso-recursive data types* (iso-recursive data types) if the type α and the type $\alpha \to \beta$ are not equal, but there is an isomorphism (function $1 \to 1$, invertible, which for each value of α associates a value of $\alpha \to \beta$ and vice versa) from one to the other. This isomorphism (which establishes that the two types are essentially equivalent) is the "constructor" function of the algebraic data types.

Understanding that Haskell supports iso-recursive types (a value of α can be generated using a special constructor starting from a value of $\alpha \to \beta$), but not equi-recursive (α and $\alpha \to \beta$ **cannot** be the same type!) tt then becomes clear why the definition of `Y` given above generates a syntax error (in particular, a type error).

This error does not mean that the definition of the Y combinator (Equation 5.2) is "wrong", but simply that it is not directly implementable in Haskell (or in any other language that uses strong typing). The Y combinator is defined using the λ-calculus, where each identifier is bound to a function, whose type is not important. Using languages with "less strict" type checking than Haskell (for example, Lisp, Scheme, Python, Javascript, etc...) or languages that support equi-recursive types (for example, OCaml with appropriate options), Equation 5.2 can be implemented without issues.

To better understand how to solve this problem, let us consider the "problematic part" of the previous expression (the application of function "`(xx)`"), focusing on the function "`\x -> (xx)`".

Since in Haskell a recursive data type `t0` can be defined using the `data` construct and (at least) a constructor mapping values of a type that depends on `t0` into values of `t0` (iso-recursive type definition), one can define something like $T = F(T \to \beta)$ to "simulate" the (equi-recursive) type $\alpha = \alpha \to \beta$ of function `x`. The resulting type `T` will then be a function of the type β and in Haskell this is denoted by "`T b`". Remebering the syntax of the "`data`" keyword, we can write

```
data T b = F (T b -> b)
```

where (as mentioned) "`T b`" is the name of the type and "`F`" is the name of the constructor that maps values of `T b -> b` into values of `T b`.

At this point, it is possible to use "`T b`" to correctly type "`x`" by writing "`x`" as a function `T b -> b` from `T b` to `b`. The argument (actual parameter) of this function must therefore be of type `T b`, which can be generated from "`x`" using the constructor `F`. Instead of "\x —> x x" we can then write:

```
\x -> (x (F x))
```

so that Haskell is able to understand and compile this definition.

Note how the impossibility of using equi-recursive types forced us to use the `F` constructor. The type of the function defined above is `(T b -> b) -> b` (function that maps values "x" of type "`T b-> b` " to values of type "`b`").

Now that we have seen how to solve the "x" type issue, it is possible to go back to the original Y combinator expression, which still has a similar problem: "`\x - > f(x (F x))`" applies to itself, so its type is recursive like the type of "`x`"! Again the problem can be solved by using the previously defined "T b" datatype: "\f —> (\x —> f (x (F x)))" has type "`(b -> c) -> (T b -> b) -> c`", so assuming "`b -> c` " as type for "`f`" we have that "`\x -> f (x (F x))`" has type "`(T b - > b) -> c`"... The value "`\x -> f (x (F x))`" to which it is applied must therefore have type "`Tb -> b`". Hence, "`(T b -> b)`" (type of "`x`") must be replaced with "`T b`" (obtainable from "`x`" by applying the constructor "`F`"). The argument will then be "\f —> (\ (F x) —> f (x (F x)))".

As a result, the expression of the Y combinator should be:

```
y = \f -> (\x -> f(x (F x)))(\ (F x) -> f(x (F x)))
```

and in theory this expression should be accepted by Haskell! Unfortunately, however, some `ghc` (and therefore `ghci`) versions have problems correctly inferring data types[2]. Other programs, such as the Hugs interpreter (`https://www.haskell.org/hugs`) are able to parse and evaluate this definition without problems.

The problem encountered by some versions of `ghc` can be overcome by somehow "helping" the compiler to correctly infer the types of the various subexpressions. For example, you could replace "`F x`" with a variable "`z`" gaving the right type (`T b`). To do this, however, you need a function that allows you to extract "`x`" from "`F x`"; in practice, the inverse function of the "`F`" constructor:

```
invF (F x) = x
```

At this point it is possible to replace "`\(F x)`" with "`\z`" and the following "`x`" with "`invF z`", obtaining

```
y = \f -> (\x -> f(x (F x)))(\z -> f((invF z) z))
```

[2]This is probably related to a bug described in `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/bugs.html`.

and this expression is accepted by all versions of `ghci`!

Summing up, the first part of the expression ("`\x -> f(x (F x))`") has type "`(T b -> b) -> c`' ', while the second part ("`\z -> f((invF z) z)`") has type "`T b -> c`" and is therefore usable as an argument to the former simply by setting $\beta = \gamma$.

The type of the resulting function is "`(b -> b) -> b`", where "`b`" is clearly a function type (for the factorial, for example , is a function "`Int -> Int`").

The above discussion, which shows how it is possible to implement the fixed point combinator Y in strictly typed languages (like Haskell) that do not support equi-recursive data types, allows us to intuit one important thing: Y "eliminates recursion " by the fixed point operator (Equation 5.1 uses explicit recursion) by moving the recursion to the data type (indeed it requires using recursive data types of some kind). This is not immediately visible in Equation 5.2 or scheme (or similar) implementations of Y, but it becomes clearer trying to implement Y in (for example) Haskell.

Once Haskell has "accepted and understood" our definition of Y, one can, for example, use this Y combinator to define the factorial function. First of all, let's define a "closed" version `fact_closed`[3] of the factorial function as

```
    fact_closed = \f -> \n -> if n == 0 then 1 else n * f (n - 1)
```

At this point the factorial function can be defined as

```
    fact = y fact_closed
```

Note that the type of "`fact_closed`" is "`(Eq p, Num p) => (p -> p) -> p -> p`", therefore, since Y has type "`(b -> b) -> b`" "`y fact_closed`" has type "`(Eq p, Num p) => p -> p`".

So, a possible definition of the Y combinator in Haskell (there are many others) can be:

```
data T b = F (T b -> b)
invF (F x) = x
y = \f -> (\x -> f(x (F x)))(\y -> f((invF y) y))
```

Based on this definition it is possible to use `ghci` to do:

```
Prelude> fact_closed = \f -> \n -> if n == 0 then 1 else n * f (n - 1)
Prelude> :t fact_closed
fact_closed :: (Eq p, Num p) => (p -> p) -> p -> p
Prelude> fact = y fact_closed
Prelude> :t y
y :: (b -> b) -> b
Prelude> :t fact
fact :: (Eq p, Num p) => p -> p
Prelude> fact 3
6
Prelude> fact 4
24
...
```

[3]This function is called "closed" because it has no free variables. The traditional recursive definition of the factorial function instead uses a free variable (its name) to call itself recursively.

Appendix A

Some Definitions

A.1 Identifiers, Bindings, and Environment

Almost all of the programming languages allow to associate symbolic names to the various “entities” composing the programs (values, memory locations, variabiles, functions, etc...).

More formally, a programming language is composed of *denotable entities* that can be referred through symbolic names (identifiers). This mechanism is implemented by a function named *environment*, having the set of possible identifiers as a domain and the set of denotable entities as a codomain. This function can change during the program runtime, as it is possible to create new bindings between identifier and denotable entities, or to destroy existing bindings. Moreover, some bindings between identifiers and denotable entities only exist in some parts of the code[1]. Hence, the following definitions hold:

Definition 1 (Binding) *A binding between an identifier I and a denotable entity E is a pair (I, E) associating name and entity.*

Definition 2 (Environment) *The environment is the set $\{(I, E)\}$ of bindings existing in a specific moment of the program execution, while the code of a specific program block is being executed.*

this definition is not surprising, since the environment is a function $env : \mathcal{I} \to \mathcal{E}$ (where $\mathcal{I}$ is the set of possible identifiers, and $\mathcal{E}$ is the set of denotable entities) and from the mathematical point of view a function is a set of (I, E) pairs, so $env \subset \mathcal{I} \times \mathcal{E}$.

The set of denotable entities is clearly language-dependent; for example:

- in the Assembly language, identifiers can be associated only to memory locations (so, the only possible denotable entities are memory addresses)
- in higher-level imperative languages, identifiers can be associated to values, variables, functions, or data types
- in λ-calculus, identifiers can only be associated to functions (so, functions are the only possible denotable entities)
- in higher-level functional programming languages, identifiers can be associated to values (mutable variables do not exist, and functions are values!)

In programming languages having the “code block” concept, it is possible to make a distinction between *local environment* (the environment valid inside a code block), *non-local environment*, and *global environment.*

Definition 3 (Local Environent) *The local environment of a code block is the subset of the environment composed of the bindings that have been created inside the code block (and are hence valid only inside this block)*

Definition 4 (Non-Local Environment) *The non-local environment of a code block is the subset of the environment which is not part of the local environment (and hence contains all the bindings that have been created* outside *of this block and will remain valid when the execution exits the block)*

Definition 5 (Global Environment) *The global environment is the subset of the environment containing bindings that are not created inside any code block*

[1]As an example, the environment is modified by a declaration, or by the first usage of a denotable entity.

A.2 Mutable Variables

According to the imperative approach, the execution of programs happens by modifying the values contained in some memory locations, which are called *variables* in high-level languages.

Definition 6 (Mutable Variable) *A mutable variable is a denotable entity representing some memory locations that can contain* storable entities

The definition above is based on "*storable entities*", which, again, are a language-dependent concept... But in general a storable entity is some value that can be stored in a variable.

From the conceptual point of view, mutable variables imply the existance of a second function, after the environment, named "*store*", associating each variable with the storable entity contained into it.

Definition 7 (Store) *The store is a set of pairs* (V, E) *(where* V *is a variable and* E *is a storable entity) associating each variable with the value contained into it*

The store is hence a function (representing the memory used by the program to store its data, or the mutable state of the program) which has the set of program's variables as a domain, and the set of storable entities used by the program as a codomain.

When, using an imperative programming language, the value stored in variable "`x`" is used, the (abstract) machine applies the store function to the value returned by the environment function applied to identifier "`x`": "`x`": $store(env(\mathtt{x}))$ (where "env" is the environment).

A.3 Denotable, Storable, and Expressible Entities

As discussed, a program is composed of some "entities" (whose definition depends on the programming languages, but can be data types, values, variables, functions, etc...). Such entities can be *denotable*, *storable*, and *expressible*.

(Note: in literature the definitions of "storable", "expressible", and "denotable" are often associated to values)

Definition 8 (Denotable Entities) *A denotable entity is a language entity that can be associated to a symbolic name / identifier*

Definition 9 (Storable Entities) *A storable entity is a language entity that can be stored in a variable*

Definition 10 (Expressible Entities) *An expressible entity is a program entity that can be generated computing an expression*

A denotable entity is hence a generic entity that can be referred through a name (defined by the user or pre-defined in the language); the set of denotable entities is the codomain of the environment function. An expressible entity is a generic entity that "can be computed/allocated" somehow using the language's constructs. Finally, the storable entities (which only exist in imperative programming languages) form the codomain of the store function.

In a functional programming language, all entities are denotable and expressible, whereas in an imperative programming language there can be entities that are denotable but not expressible or storable (for example, functions in the C programming language).

A.4 Functions

Almost all the high-level programming languages allow some form of code modularization by decomposing programs into a set of components (subprograms/subroutines/functions/procedures). Each one of these components implements a specific functionality according to a well-specified interface.

Each subroutine is hence an *entity* implementing a self-contained part of the code which can be invoked exchanging some values (parameters and return values) with the caller. A subroutine which can return a value is generally called "function" (clearly, this is a very different thing respect to a mathematical function!!!).

Definition 11 (Function) *A function is a denotable entity composed of a block of code associated to a name that can be used to invoke its execution. When the execution of a function is invoked, the caller can exchange some data with it through its parameters, its return value and some global state of the program.*

```
void wrong_swap(int a, int b)
{
  int tmp;

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

void correct_swap(int *a, int *b)
{
  int tmp;

  tmp = *a; *a = *b; *b = tmp;
}
```

Figure A.1: Example of C function trying to exchange the values of two variables. Since the C language mandates parameters passing by value, the "`wrong_swap()`" will have no effect (as its name suggests); the "`correct_swap()`" function, instead, receives as parameters some *pointers* to the variables, and can hence work correctly.

```
void reference_swap(int &a, int &b)
{
  int tmp;

  tmp = a; a = b; b = tmp;
}
```

Figure A.2: Example of function swapping the contents of two variables, implemented in C++ passing parameters by reference.

Definition 12 (Foramal Parameter) *A formal parameter of a function (specified in the function's definition) identifies a variable that can be used by the caller to pass data to the function when invoking it*

Definition 13 (Actual Parameter) *An actual parameter is an expression (specified when invoking a function) that will be associated to the corresponding formal parameter during the function's execution*

The fact that a function is composed of a block of code makes it clear that the function is characterised by a local environment (containing bindings between names and local variables, names and function parameters, etc...). Moreover, since a function is defined as a denotable entity this block of code (the function body) can be associated to a name (but this is not always necessary: anonymous functions do exist!).

The way actual parameters are associated to formal parameters depends on the mechanism used to invoke the function, and to the parameters passing style that is used. For example, a formal parameter identifies a variable which can be created when the function is invoked (and is hence a local variable of the function), or can exist before the function is invoked. Different parameters passing styles can be used, and the most important are: parameters passing by value, by reference, and by name.

When parameters are passed *by name*, a new local variable for each formal parameter is allocated when the function is invoked (and is deallocated when the function returns). The simplest way to manage these variables is allocating them on the stack. The local environment of the function then binds the formal parameter name to this variable, and the variable is initialized with the value of the actual parameter (hence the name "by value"). If the function modifies the value stored in this variable, the modifications are discarded when the function returns and the variable is deallocated. In other words, passing parameters by value it is possible to pass data from the caller to the called, but not vice-versa. This mechanism is the only one supported by the C programming language; as a result, the "`wrong_swap()`" function in Figure A.1 does not work correctly (and pointers are needed to code a working "swap" function).

When parameters are passe *by reference*, instead, no new variable is allocated when the function is

```
int f(int v)
{
  int a = 666;

  return a + v;
}
```

Figure A.3: Example of function passing parameters by name.

invoked. Parameters are passed by modifying the local environment of the function so that the formal parameter's name is bound to the actual parameter. Hence, the actual parameter has to be a denotable entity (for example, things like "`x + 1`" are not valid actual parameters when passing parameters by reference). Using the C/C++ jargon, this means that actual parameters have to be L-values. This parameters passing style is not supported by the C language, but is supported by C++; as an example, see the "`reference_swap()`" function in Figure A.2 (comparing this code with Figure A.1 it is possible to better understand the differences between passing parameters by name and by reference).

Finally, when parameters are passed *by name* invoking a function requires to replace each formal parameter with the corresponding actual parameter (this is just text replacement). This mechanism is typically used to evaluate (reduce) functional programs. Although parameters passing by name might look simple to simplement, there are some subtle issues to be addressed. For example, consider the "`f()`" function from Figure A.3: the goal of the function is to sum 666 to the value received as input, and if parameters are passed by name the "`f(n)`" is evaluated as "{ **int** a = 666; **return** a + n;}", which is reduced to "{ **return** 666 + n;}", and the return value is correctly "`666 + n`". But if "`f(a)`" is invoked, things become more complex: a simple replacement of "`v`" with "`a`" would result in "{ **int** a = 666; **return** a + a;}" which reduces to "`666 + 666`", clearly not the expected result... The issue is that when replacing "`v`" with "`a`" it is not possible to make a distinction between two different entities (a local variable and a non local one) which have the same name "`a`".

This issue is generally addressed in functional programming by properly changing the variables' names: if "`f()`" is invoked using as an actual parameter an expression containing "`a`", then the local variable "`int a`" must be renamed (for example to "`a1`") so that the replacement can result in "{ **int** a1 = 666; **return** a1 + a;}" which correctly evaluates to "`666 + a`".

From an implementation point of view, parameter passing by name has been historically implemented by passing "*thunks*", which are (environment,expression) pairs, as parameters. Hence, a function receiving parameters passed by name can be implemented as a function receiving (environment,expression) pairs as parameters[2].

A.5 Closures

```
void->int counter(void)
{
  int n = 0;

  int f(void) {
    return n++;
  }

  return f;
}
```

Figure A.4: Example of a function returning a closure.

In some programming languages, functions are storable entities (there are variables that can store functions) or expressible entities (it is possible to build expressions evaluating to a function). As a result,

[2]the expressions are the actual parameters, and the environments are used to evaluate such expressions; in the example of Figure A.3, the expression is "`a`" and the environment binds this "`a`" name to the global "`a`" variable).

they can be used as actual parameters for other functions, or as return values for functions. In these cases, the values to be stored, returned, or passed as parameters are technically “*closures*”

Definition 14 (Closure) *A closure is a pair composed of a function and its non-local environment*

Basically, a closure is needed to find the entities bound to identifiers for which there are no bindings in the local environment of the function.

```
#include <stdio.h>

int (*counter(void))(void)
{
  int n = 0;

  int f(void) {
    return n++;
  }

  return f;
}

int main()
{
  int (*c1)(void) = counter();
  int (*c2)(void) = counter();
  int (*c3)(void) = counter();

  printf(”␣␣␣%d␣%d\n”, c1(), c1());
  printf(”%d␣%d␣%d\n”, c2(), c2(), c2());
  printf(”␣␣␣%d␣%d\n”, c3(), c3());

  return 0;
}
```

Figure A.5: Example showing the function pointers in C are not closures (the example also uses a non-standard extension provided by the `gcc` compiler).

As an example, look at the function “`void->int contatore(void)`” from Figure A.4, is coded using a pseudo-language with a C-like syntax in which “`void->int`” is the type of the functions withot arguments that return a value of type `int`. The “`counter()`” function receives no arguments and returns a function that generates all the integer numbers starting from 0. It is possible to notice that “`n`;; is a local variable of the “`counter()`” function, and not a variable or an argument of the “`f()`” function. Hence, when “`counter()`” is invoked its local variable contains a binding from “`n`” to a local variable initialized to 0. Such a binding is not in the local environment of “`f()`” (it is in its non-local environment) and when “`counter()`” returns the variable bound to “`n`” is deallocated. So, in order for the program to work 2 things are needed:

1. The variable bound to identifier “`n`” should not be deallocated when “`counter()`” returns. Hence, it cannot be allocated on the stack, but must be allocated on the heap
2. The binding between identifier “`n`” and such a variable has to be somehow associated to the returned function. It is hence copied in a new environment that will be part of a closure returned by “`counter()''`

To better understand the differences between closures and function pointers, look at Figure A.5, that uses a `gcc` extension to implement the function of Figure A.4 in C. In this case, the “`int n`” variable is deallocated when function “`counter()`” returns, hence the program has an undefined behaviour.

A.6 Closures and Classes

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

auto counter(void)
{
  int n = 0;

  return [n](void) mutable {
    return n++;
  };
}

int main()
{
  auto c1 = counter();
  auto c2 = counter();
  auto c3 = counter();

  std::cout << c1() << " " << c1() << std::endl;
  std::cout << c2() << " " << c2() << " " << c2() << std::endl;
  std::cout << c3() << " " << c3() << std::endl;

  return 0;
}
```

Figure A.6: Example showing how to use C++ lambda functions, which implement closures.

```
#include <iostream>

class Counter {
  private:
    int n;
  public:
    Counter(void) : n(0) {
    }
    int operator() (void) {
      return n++;
    }
};

int main()
{
  auto c1 = Counter();
  auto c2 = Counter();
  auto c3 = Counter();

  std::cout << c1() << " " << c1() << std::endl;
  std::cout << c2() << " " << c2() << " " << c2() << std::endl;
  std::cout << c3() << " " << c3() << std::endl;

  return 0;
}
```

Figure A.7: Implementing the "`counter()`" function in C++ using classes.

Closures allow associating a non-local environment to a function, and have been originally introduced to implement high-order functions (functions returning functions as a result, or accepting functions as parameters). But the resulting abstraction is much more powerful than this, and allow associating data (the state contained in the variables bound by the non-local environment) to code (the code implementing the function's body). And this is very similar to what classes do.

To better understand the relationship between closures and classes, look at the C++ implementation of a counter as a closure or as a class, as shown in Figures A.6 and A.7. As it is possible to notice, the "`int n`" variable in the first case is a local variable of the "`counter()`" function (then embedded in the closure) and in the second case is encapsulated in a class as a private member.

‘