Haskell for Dummies

Luca Abeni

December 12, 2022

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 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

[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.

[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

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).

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"`.

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>

[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.

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 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>
```

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

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

"`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
```

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 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 "$f : \mathcal{X} \to \mathcal{X}$", 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].

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:

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

```
\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.

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

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)
```

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:

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

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

Figure 2: Implementation of the tail-recursive factiorial function in Haskell, hiding the `fact_tr` function through the `where` construct.

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` 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 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 1), as shown in Figure 2.

Comparing the two examples in Figure 1 and 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).

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 = ...".

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
```

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).

```
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: 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 4: Converting amounts of "`Money`", final version.

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 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 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...).

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 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:

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

```
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":

```
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:

```
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 5: Fattoriale con ricorsione in coda.

```
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 6: Massimo Comun Divisore.

```
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 5)
- Compute the greatest common divisor between two numbers (Figure 6)
- Solution to the Tower of Hanoi problem (Figure 7 and 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
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

```
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 7: Torre di Hanoi.

```
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 8: Torre di Hanoi, versione alternativa.

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 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.
*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.