Programming Paradigms
Functional Programming

Pedro Santos

May 19, 2016

Contents

1 Functional Language

Functional languages are languages that support and encourage the functional style of programming. This is style is characterized by the use of functions and arguments. Functional languages approach computation on its basic level.

1.1 Summing 1 to 10 in java

```
int total = 0;
for (int i = 1; i =< 10; i++)
  total = total + i;
```

1.2 Summing 1 to 10 in Haskell

```
sum[1..10]
```

1.3 Example:quicksort() in haskell

```
f [] = []
f (x:xs) = f ys ++ [x] ++ f zs
 where
    ys = [a | a <- xs, a <= x]
    zs = [b | b <- xs, b > x]
```

2 Introduction to Haskell

2.1 The Prelude

Haskell as a high-level language include in its prelude a set of functions from the basic math operators such as + and * to more advanced list functions such as head, tail and take. As haskell is a functional language it avoid using variables so working with those high level functions are essential to manipulate lists or complex inputs.

2.1.1 Examples with list manipulators

Examples of:

- tail
- index selector
- take

```
tail [1..5]
[2,3,4,5]
```

```
[1,2,3,4,5] !! 2
3
```

```
take 3 [1,2,3,4,5]
[1,2,3]
```

2.2 Functions with lists

Many functions on the haskell prelude are designed to work with functions such as product which returns the product of all the elements in a list, (++) which appends two lists together, or reverse which reverses a list. Those high level functions can reduce the amount of code necessary to write a program making Haskell programs usually much smaller in comparison to Java or C programs.

2.3 Function application

In maths, a function is applied by using '()', in haskell we use spaces to separate the function from its parameters. For example:

```
1 --where function is the name of the function and 'a b' are the
    input parameters
2 function a b
```

2.4 Haskell scripts

Haskell allows the creation of scripts, those are usually .hs files which contain a series of function definitions which can be called once in the interpreter. To open those scripts its possible to call ghci filename.hs and assuming no errors are encountered the defined function will be able to be called in the same way as prelude functions.

2.4.1 Example factorial and average function in haskell

```
1 factorial n = product [1..n]
2 average ns = sum ns 'div' length ns
```

1

3 Types and classes

One of the main characteristics that distinguish Haskell to OO languages is type. A type can be compared to variable types in languages like C or Java as most times they assume values such as booleans, integers or chars. One of the main

[1]Variables in haskell can be named at will. However it is useful to add an s after a variable when representing lists as it makes reading functions more intuitive

differences between types in haskell and variable types in other languages is that the Haskell interpreter will return errors if it encounters a value that does not fit the type expected for example in Haskell 1 + false will return an error as the parameters are different types, meanwhile a few languages will try to sum those values and convert the false to an integer normally 0 for false and return 1 as a result, this example is true in languages such as Javascript.

In Haskell the operator '::' is used to define types for functions, and the command :type can be used in GHCi to return the type of a function. This is useful when using default prelude functions.

```
1 >not False
2 True
3
4 > :type not False
5 not False :: Bool
```

This characteristic of Haskell and a few other functional languages is often called Strong typing.

3.1 Basic Haskell Types

The basic types used in Haskell are:

Bool Logic values

Char Single characters

String String of characters

Int fixed-precision integers

Integer Arbitrary-precision integers

Float floating-point numbers

All Haskell types can be used in the creation of tuples and lists which are more complex types. In the case of a list the type name must be surrounded by '[]' thus defining the type as a list of whatever type is in the brackets. Types can also be defined as tuples which are a conjunction of two or more types in a sequence such as (Bool, Bool) or (Int, Bool) Both tuples and lists can be composed of each other as we can have lists of tuples [(Bool,Bool)] or tuples with lists are their components ([Int],Bool).

3.2 Function Types

Functions in Haskell must have their typed defined by both its inputs and its returns, for example a function 'not' must be defined as a Bool input to a Bool output, this is done using the :: operator and the ->operator.

```
1 not :: Bool -> Bool
2 --or in the case of different types for input and output
3 even :: Int -> Bool
```

3.3 Curried Functions

Curried function is the name given to functions that take multiple inputs in Haskell. Those receive their own type as well.

```
1 add    :: Int -> (Int -> Int)
2 add x y = x+y
```

Curried functions are define as a function to a function as it takes an integer to another integer function. Curried functions can have n amount of arguments as long as they are done in the same manner by using '()'. For example a function with 3 inputs would be defined as:

```
1 mult       :: Int -> (Int -> (Int -> Int))
2 mult x y z = x*y*z
```

Curried functions can be avoided by using tuples as arguments such as:

```
1 add       :: (Int,Int) -> Int
2 add (x,y) = x + y
```

However curried functions are useful as curried functions can be partially applied to generate other functions.

3.4 Polymorphic functions

Polymorphic functions are functions that can take different forms depending on their variables. In the case of haskell polymorphic functions are functions defined with a relative type, one of the best examples of this feature is the length function which will take a list of any type and return an integer regardless of the type of the list.

```
length :: [a] -> Int
--the a on the list represents any type and this will be valid as
    long as the input is a list
```

3.5 Overloaded Functions

A polymorphic function can be considered a overloaded function if its type contains one or more class constrains. For example the prelude (+) function is overloaded as it takes two numeric types 'a' and returns a value of type 'a'

```
(+) :: Num a => a -> a -> a
```

This means that a can be Int, Integer or Float and all it will return a value with the same type as its input.

3.5.1 Haskell classes

Haskell has a series of class types which can be used to create polymorphic types. Some of those types are:

- Num for numeric types
- Eq for equality types
- Ord for ordered types

Some examples of those types include:

```
(+)  :: Num a => a -> a -> a
(==) :: Eq  a => a -> a -> Bool
(<)  :: Ord a => a -> a -> Bool
```

Class types constrains are important as Haskell is a strongly typed language, this way wrong types of input will generate an error instead of the interpreter trying to generate an output that will most likely be wrong.

4 Defining Functions

In Haskell functions are the main component of programming and therefore Haskell counts with a number of different ways to define a function, from conditional expressions to pattern matching.

4.1 Conditional Expressions

As in most programming languages the most simple way of defining a function in Haskell is through conditional expressions such as if then else statements.

```
abs    :: Int -> Int
abs n   = if n >= 0 then n else -n
```

Haskell as most programming languages also allow for nested if statements, for example:

```
signum  :: Int -> Int
sugnum n = if n < 0 then -1 else
                 if n == 0 then 0 else 1
```

In Haskell all if statements must have an else branch, this way it avoids ambiguity problems on nested ifs.

4.2 Guarded Equations

One alternative to conditionals is the use of guarded equations, those closely resemble maths conditional functions on its declaration.

```
abs n | n >= 0     = n
      | otherwise = -n
```

The main reason of using guarded equations is the fact that it makes nested conditionals much simpler to read as all the conditions and results can be aligned in multiple lines.

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

One thing to notice while making guarded equations is that by default all otherwise clauses will return true if not defined as something else.

4.3 Pattern Matching

Many functions are quite simple and have clear definitions and because of that Haskell implements pattern matching which is a simple way of defining simple functions such as logic operators. This method works by simply relating a set of inputs to a set of outputs.

```
not       :: Bool -> Bool
not False = true
not True  = false
```

Pattern matching is viable in very few scenarios but when it is viable it allows programmers to save time on simple functions. Haskell also allows for the use of '_' this will allow the function to be simplified by assuming that any inputs not defined fit the category. One example where this would simplify a function is on and && function where only one scenario out of four will give a different output.

```
True && True = True
_    && _    = False
```

One even more efficient way of defining this function is to avoid checking the second argument if the first way fails to meet the requirements.

```
True  && b = b
False && _ = False
```

Pattern matching expressions work in order what means that the interpreter will go in order looking for a match what means that the otherwise operator must always come last or all inputs will result the same.

4.3.1 List patterns

In Haskell lists are of major importance as it cant really store values in variables lists are usually the way most information and stored. Internally Haskell sees every list as a head and a tail, and this tail is a list as well so this will also have its own tail and so on. The operator when it comes to pattern matching to split the head and the tail is ':' this means that if defined (x:xs) x will be the first element of the list and xs the rest of the list. Those can be used to make pattern matching functions that do specific tasks with those parts of the list.

```
head      :: [a] -> a
head (x:_) = x

tail      :: [a] -> [a]
tail (_:xs)= xs
-- note the use of _ as we discard those values to make the
   function more efficient.
```

This method will only work on non-empty list, otherwise it will return an error.

4.4 Lambda Expressions

Haskell has a particular type of function called Lambda Expressions which are functions that do not require a name and can be defined on GHCi command line.

```
1 (\ x -> x + x) 10
2 --this will take a number x and return x + x in this particular
    case it is taking 10 as an input and therefore will return 20
```

5 List Comprehensions

In maths set comprehension is a notation used to construct sets based on a rule, such as odd numbers, even numbers, or powers of x for example. In Haskell this is translated to what is called Lists Comprehensions.

```
1 --This example will create a list with the squares from 1 to 5 as
    x is the list 1 to 5 and the definition of the comprehension is
     the square of every x.
2 [x^2 | x <- [1..5]]
```

The comprehension in this example is composed by two parts, the first one is only the declaration, while the second one 'x ¡- [1..5]' is called a generator as it generates the initial list. Comprehensions can have multiple generators as well as multiple variables for example:

```
1 [(x,y) | x <- [1,2,3], y <- [4,5]]
```

The Generators are executed in order so if they are changed the result will probably change. Multiple generators can be thought about as nested loops as the second will be executed once per every execution of the first.

5.1 Dependant Generators

Some multiple generators comprehensions can be depended as the second generator might take the results of the first one to produce a result, one example of this would be:

```
1 [(x,y) | x <- [1..3], y <- [x..3]]
```

One function that uses this resource is the concat function which merges multiple lists into one.

```
concat   ::[[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]

>concat[[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

5.2 Guards

A guard is a restriction to a generator, this works as an if statement as it will allow the generator to create numbers if those result true in the statement. The guard is the third element in a comprehension as it comes after the generator.

```
[x | x <- [1..10], even x]
--this will return the whole x generator as long as the values
    generated are even
[2,4,6,8,10]
```

5.3 The zip function

An useful function in the Haskell prelude for working with multiple lists is the function zip, this function takes two lists and combines them together by creating a list of tuples with the elements with the elements of both lists.

```
zip :: [a] -> [b] -> [(a,b)]
--For example
>zip ['a','b','c'][1,2,3,4]
[('a',1),('b',2),('c',3)]
```

One consideration while using this functions is that the end result will have the same amount of elements as the list with fewer elements. One thing to notice is that on the type definition we have two custom types, a and b this way the function is able to merge lists with two different types.

The zip function on its own might not be extremely useful but with it we can create more functions with only a few arguments. One example is creating a function that will return all pairs in a list.

```
pairs   :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
--example
>pairs[1,2,3,4]
[(1,2)(2,3)(3,4)]
```

We can now use the pairs function to make a even more complex and useful sorting function. Using Pairs we can create a sorted function, which will compare tuple by tuple until it finds an unsorted pair and if that happens it will return false otherwise it will return true.

```
sorted    :: Ord a => [a] -> Bool
sorted xs =
 and [x <= y | (x,y) <- pairs xs]
 --example
>sorted [1,2,3,4]
True
>sorted [1,3,2,4]
False
```

Haskell as most functional programming languages lacks a few of the functionalities of Object oriented languages or procedures languages. However it makes up for it by making it easy to making really complex functions out of very simple ones in a really small amount of code.

5.4 String Comprehension

Strings are a sequence of characters enclosed in double quotes, for simplification purposes all strings in Haskell treats ever string as a list of characters, this allows a string to go in any function that a normal list of characters would. For example a string such as "abc" can be used on the length function or even in a zip function.

```
>length "abc"
3
>zip "abc" "cba"
[(a,c),(b,b),(c,a)]
```

This means that in Haskell string comprehension and list comprehension is theoretically the same, as long as we are talking of functions that do not require numerical values. One very simple and useful example is defining a count function, which will count how many times a specific character will appear in a list

```
count      :: Char -> String ->
count x xs = length [x' | x <- xs, x == x']
```

This function uses a generator with a guard to create a list with only the character specified and then simply returns its length.

6 Recursive Functions

As a functional programming language Haskell tries to make the most of functions and thus encourages the use of recursion, to define functions as this way we can make really complex functions in a small repetitive procedure.

Recursive functions are all functions that call themselves in their definitions, one of the classic examples of this is a factorial function that can be seen in this example.

```
1 fac  :: Int -> Int
2 fac 0 = 1
3 fac n = n * fac (n-1)
```

As we can see this function not only uses recursion it also implements pattern matching, this in C for example would implement an if statement for the whole function. However in Haskell we can implement a single line of pattern matching to add the finishing case of the function. A recursive function needs a finish scenario as at some point it need to finish and one of the most effective ways of doing this in Haskell is by using pattern matching and matching the last theoretical input to a certain value instead of another call of the function. In Haskell we are allowed to do this and exclude an if statement which would be in theory better as it can detect out of range values because GHCi will automatically identify any inputs which will result in a infinite loop and return an error by default, making coding faster as we do not need to worry about preventing certain inputs.

6.1 Why recursion?

Recursion is a really useful tool when it comes to programming but is almost never required. However it simplifies some functions so much that it would be counter intuitive to avoid it. When creating a recursive function we only need to focus in solving part of a problem and letting the function repeat itself to solve this small problem n times until the end of the input. In the following example recursion is applied to lists where such functions allow really small amount of code to do complex tasks.

6.2 Recursion on list

6.2.1 Product

One useful way of using recursion is to apply a simple operation to all elements of a list for example multiplication.

```
1 --Notice the use of the Num class to only allow numerical values
    in the function
2 product        :: Num a=> [a] -> a
3 --We add the end scenario before the function as Haskell takes an
    ordered approach to pattern matching
4 product []     :: 1
5 product (n:ns) = n * product ns
```

With only a few lines of code we created a function which will scale infinitely and will also return errors accordingly.

6.2.2 length

One of the most useful functions in most programming languages length can also be defined recursively as it simply goes through the list adding one per every element.

```
1 length        :: [a] -> Int
2 length         = 0
3 length (_:xs) = 1 + length xs
```

Another benefit of recursion we can see in this example is the fact that we avoid creating of temporary variables and simple return the value of the function.

6.3 Quicksort

Quick sort is a simple algorithm that sorts lists. Its not the most optimal sorting algorithm when it comes to a massive sample size. However is quick to code and fairly quick in reasonable sample sizes.

```
1 qsort        :: Ord a => [a] -> [a]
2 qsort []      = []
3 qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
4                where
5                  smaller = [a | a <- xs, a <= x ]
6                  larger  = [b | b <- xs, b >  x ]
```

This implementation uses generators to create two unsorted lists around the first element of the list, and puts those around the head of the list as now this element is sorted. This is repeated, until the qsorts receive empty lists and one argument lists which will result in simple results. For comparison purposes take a look at the same algorithm in C

```
void quicksort(int x[10],int first,int last){
    int pivot,j,temp,i;
     if(first<last){
         pivot=first;
         i=first;
         j=last;

         while(i<j){
             while(x[i]<=x[pivot]&&i<last)
                 i++;
             while(x[j]>x[pivot])
                 j--;
             if(i<j){
                 temp=x[i];
                  x[i]=x[j];
                  x[j]=temp;
             }
         }
         temp=x[pivot];
         x[pivot]=x[j];
         x[j]=temp;
         quicksort(x,first,j-1);
         quicksort(x,j+1,last);
    }
}
```

This example also uses recursion, but its much longer, more complicated to read, and requires a considerable amount of temporary variables.

7 Higher-Order Functions

Higher-order functions are functions that take functions as inputs or returns functions as a result. This means we can create utility functions that can use other function. The most obvious application of higher-order functions is to create functions that will apply a function multiple time to one or more values, for example a twice function which applies the same function twice to the same value.

```
-- as we can see in the type definition the first argument is a
    function as it takes a type and returns the same type
```

```
twice    :: (a->a) -> a -> a
twice f x = f (f x)
```

7.1 The map function

One of the most useful high-order functions in the Haskell prelude is the Map function, this takes a function as a input and applies that function to every element in a list.

```
map    :: (a -> b) -> [a] -> [b]
>map (1+) [1,3,5,7]
[2,4,6,8]
```

The map function can be defined really simply using function comprehension

```
map f xs = [f x | x <- xs]
```

It can also be defined using recursion but in this particular case list comprehension allow for a simpler definition but for demonstration this is a valid definition of map.

```
map f []     = []
map f (x:xs) = f x : map f xs
```

7.2 The filter function

The filter function filters all elements that result true in a function from a list. This is particularly useful as it allows quick and intuitive creation of new lists without the need of the use of generators and list comprehensions.

```
-- as we can see in this example the filter function will take
   only functions that result in a Bool result as it needs a true
   or false value to decide if the elements stays or not in the
   list.
filter :: (a -> Bool) -> [a] -> [a]
>filter even [1..10]
[2,4,6,8,10]
```

7.3 The foldr function

The foldr function allows the creation of functions that follow a simple recursive pattern, This function takes a function as a input and also returns a function as a result so it is used to define functions.

The pattern used by foldr returns the following:

```
f []     = v
f (x:xs) = x f2 f xs
```

foldr takes a f2 and v, where f2 is another function and v a default value for the empty list. A few examples where this function can be used is to define the following functions for lists.

```
sum     = foldr (+) 0
product = foldr (*) 1
or      = foldr (||) False
and     = foldr (&&) True
```

This function is never required. However is a tool that can be really useful to simplify code as it allows to the quick creation of recursive functions that follow that pattern.

7.4 Dot notation (.)

The library function (.) also known as dot notation allows for the composition of new functions based on two functions.

```
(.)   :: (b -> c) (a->b) (a->c)
f . g  = \x -> f(g x)
```

This function allows the creation of new function by simply merging two existing functions into one. A simple example of this would be implementing 'not .' to existing functions to create new reverse functions.

```
odd :: Int -> Bool
odd  = not . even
```

7.5 More High-order functions

Click on the functions to go to its entry on the function appendix.[2]

- all
- any
- takeWhile
- dropWhile

[2]Functions in this list are mentioned but not deeply explained in lectures.

8 Type declarations

Haskell allows the creation of custom types by what is called a type declaration. As mention previously a string is a list of chars so we can define a string with the following

```
type String = [Char]
```

Type declarations are useful as they allow code to be easier to read, for example when working with coordinates you can define a new type Pos which will hold x and y coordinates as a tuple.

```
type Pos = (Int, Int)

origin     :: Pos
origin      = (0,0)

left       :: Pos -> Pos
left (x,y) = (x-1,y)
```

Although not required for this previous example defining this new type can save time while programming. Like functions types can also recieve parameters, this way you can make one definition for multiple types. In this example we set a new type called Pair which will create a tuple with a pair of any type given.

```
type Pair a = (a,a)
mult        :: Pair Int -> Int
mult (m,n)   = m*n
```

Type declarations can be nested by creating a new type defined by another user generator type. However type declarations cannot be recursive.

8.1 Data declarations

If you required to create a completely new type from scratch and not from already predefined types you can use a data declaration.

```
--This declaration creates a new type called Bool which can hold
    either true or false.
data Bool = False | True
```

The values False and True in this example are called Constructors, types and constructors must begin with upper-case letterers. One useful example of creating a new data type would be an answer type with three potential values

```
data Answer = Yes | No | Unknown
```

This is a more complex type of Bool which would be useful for a marking program for example where not all questions are actually answered.

Data constructors are also allowed to have their own parameters allowing the creation of complex types and the creation of polymorphic functions that would fit this new type.

```
data Shape      :: Circle Float | Rect Float Float
square n         = Rect n n

area            :: Shape -> float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

In this last example we can consider Circle and Rect as constructor functions, similar to constructors in a OO language.

8.2 Recursive Types

Although new types cannot be recursive new data types can, this allows for the creation of really complex structures and most importantly infinite lists.The most simple example to show this would be all the natural numbers

```
data Nat = Zero | Succ Nat
```

This will create a type that can be either Zero or the number next Nat itself, looping this indefinitely we will get a type that can hold zero or any natural number.

9 Function Appendix

List of Haskell Prelude functions mentioned in lectures

Sums all elements of a list

```
sum [1,2,3]
6
```

9.0.1 sqrt

Square root function

```
sqrt (3^2 + 4^2)
5.0
```

9.0.2 product

returns the product of all elements in a list

```
product [1,2,3,4,5]
120
```

9.1 List functions

9.1.1 tail

Takes a list as an input and returns the tail of the list

```
tail [1,2,3,4]
[2,3,4]
```

9.1.2 head

Takes a list as an input and returns the first element of this list

```
head [1,2,3,4]
1
```

9.1.3 take

Takes a number and a list and returns the first n elements of that list

```
take 3 [1,2,3,4,5]
[1,2,3]
```

9.1.4 drop

remove the first n elements from a list

```
drop 3 [1,2,3,4,5]
[4,5]
```

9.1.5 length

Returns the length of a list

```
length [1,2,3,4,5]
5
```

9.1.6 concat

merges all lists in a list

```
>concat[[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

9.1.7 zip

Merges two lists into one list of tuples

```
zim :: [a] -> [b] -> [(a,b)]
--For example
>zip ['a','b','c'][1,2,3,4]
[('a',1),('b',2),('c',3)]
```

9.2 High-Order Functions

9.2.1 Map

Applies the input function to all elements of a list

```
map    :: (a -> b) -> [a] -> [b]
>map (1+) [1,3,5,7]
[2,4,6,8]
```

9.2.2 Filter

Filters out all elements that do not return true to the function given.

```
>filter even [1..10]
[2,4,6,8,10]
```

9.2.3 (.)

The . function merges two functions to create a new function.

```
odd :: Int -> Bool
odd  = not . even
```

9.2.4 all

Checks if all elements in a list return true to a function

```
> all even [2,4,6,8,10]
True
```

9.2.5 any

Checks if at least one element in a list return true to a function

```
> any (== ' ') "abc def"
True
```

9.2.6 takeWhile

List all elements until a function returns false.

```
> takeWhile (/= ' ') "abc def"
"abc"
```

9.2.7 dropWhile

Removes all elements until the condition is met

```
> dropWhile (== ' ') " abc"
"abc"
```