The conceptual and formal analysis of the Haskell programming language

Adonis Bodea 931
Carmen Pintea 936
Cosmin Pletosu 936

Why Haskell? - purely functional, statically typed, lazy programming language (Carmen Pintea)

Haskell is **functional**. Functional programming is a computer programming paradigm that relies on functions modeled on mathematical functions. The essence of a functional program is the *expression*. Expressions include concrete values, variables and also functions. In Haskell, and in functional programming more generally, functions are *first-class citizens* (which means that they can be used as values and passed as arguments to yet more functions).

Because it is functional, Haskell is based on the lambda calculus. Some functional languages incorporate features that aren't translatable into lambda expressions. Haskell is a **pure** functional language, because it has no exception. The word *purity* is sometimes also used to refer to *referential transparency*, which means that the same function with the same input will always return the same result, regardless of any global or local state. The referential transparency is ensured by the impossibility of the existence of side effects during the execution of a function. How? Haskell programs are constrained by the **SAF** (single assignment form) rule, which means that the variables in Haskell are *immutable* and they can be assigned only once.

In imperative languages the programs are a sequence of tasks to execute. In functional programming the focus is on what stuff *is* instead of defining steps that change some state and the problem-solving approach is radically changed.

Haskell is **statically typed.** This means that the type checking occurs at compile time which brings the advantage of quickly identifying errors from the compilation phase. Being a functional language, the code is not a series of commands to be executed, but expressions to be evaluated. In haskell every expression has a type. Haskell uses a type system capable of *type inference*. This means that the programmer does not have to say the type of a variable explicitly because the system can intelligently deduce the type.

```
getText number name = if number > 0
                      then "Hello, " ++ name
                      else "Bye, " ++ name
```

In the code snippet above, the compiler will deduce that *number* is a Num (from number >0) and *name* is a String (from concatenation ".." ++ name).

Haskell is **lazy**. When we talk about evaluating an expression, we're talking about reducing the terms until the expression reaches its simplest form (it is irreducible). Haskell uses a *nonstrict evaluation*, also called lazy evaluation, which defers evaluation of terms until they're forced by other terms referring to them. This feature helps the programmer to view the program as a series of data transformations and allows the use of infinite data structures

that can have an important applicability in the field of mathematics. A very good example which clearly explains the laziness concept is the following Fibonacci numbers definition:

```
fibs :: [Int]
fibs = 0 : 1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

First, let's explain it step by step. *(a, b)* is a tuple, meaning two values stuck together. The tuple is a convenient way to store and pass multiple values in Haskell. The code adds 'a' and 'b' together to get a number in the Fibonacci sequence, so clearly, *(a, b)* is of type '(Int, Int)'. *'zip'* is a function that takes two lists L1={l_{11},l_{12}, …}, L2={l_{21},l_{22}, …} and returns a list of tuples of the form L={(l_{11},l_{21}), (l_{12},l_{22}) …}. The *tail* function, as its name says returns the argument list without the first element. So the call *zip fibs (tail fibs)* assigns to *(a, b)* the first two elements of the *fibs* which are then added and appended to the list. The first parameter to *zip* is *'fibs'*, which is the list defined by the expression!

The definition of fibs is evaluating itself while it is computing itself and fibs seems to be infinite. This works due to **Haskell's laziness**: until you read some values from *fibs* and print them, there's only the 0 and the 1, plus the function to generate more of the list. When you read an element from *fibs* list, *fibs* will be evaluated to that point, and no further. Also, because *fibs* is defined globally, it will remain defined in memory, making reading further values very quick.

`fibs !! 30` - will return the 30th element and the first 30 elements will be kept in memory

`fibs !! 35` - will return the 35th element much faster

Haskell installation (Pletosu Cosmin)

One can try Haskell on repl.it. To compile the code on your own machine, there are three widely used ways to install the Haskell toolchain on supported platforms:

- *Minimal installers:* just GHC (compiler) and build tools (Stack or Cabal)
- *Stack*: a project-centric build tool to automatically download and manage Haskell dependencies on a project-by-project basis
- *Haskell Platform*: GHC, Cabal and a starter set of libraries in a global location on the system

Haskell offers two primary ways of working with code: either by using the REPL / GHCi environment, or by writing the program in source files and compiling them.

REPL is short for read-eval-print loop. It is an interactive environment where one can input code, have it evaluated, and see the result. Files can be imported using the :load command.

```
Prelude> 5 * 2
10
Prelude> 2 ^ 65
36893488147419103232
Prelude> :t 8
8 :: Num p => p
Prelude>
```

Prelude is a library of standard functions that is contained in Haskell's base package.

As nice as REPLs are, usually one wants to store the code in a text file (.hs extension) so he or she can build it incrementally. To compile the source codes, the programmer must use the "ghc" command in the command line interface:

```
D:\fac\plf\lab6>ghc file1.hs file2.hs
[1 of 2] Compiling Shop             ( file1.hs, file1.o )
[2 of 2] Compiling Main             ( file2.hs, file2.o )
Linking file2.exe ...
```

Because the Haskell language permits both interpretation and compilation, it has the best of both worlds: during the development phase using an interpreter can enhance the productivity and the final product can be compiled to benefit from more speed and not being dependent on an environment.

(print "Hello world!") and other basic syntax rules (Pletosu Cosmin)

In the next section the syntax of the Haskell programming language is going to be presented using practical examples that will be briefly explained.

A function to analyze

```
addNumbers :: [Int] -> Int
addNumbers [] = 0
addNumbers (head:tail) = head + addNumbers tail
```

The name of the function written above is **addNumbers**. It **must** begin with a lowercase letter. Here, :: is a way to write down a type signature and can be interpreted as "has the type" (so addNumbers has the type `[Int] -> Int`). What it actually means is that the function has to take a **list** of **Int**s as input and return an **Int** as result. Between the function's name and the = symbol are placed the formal parameters. The = is used to define the body of the function.

In this function definition one can observe the similarities with the "facts and rules" concept: the first body definition contains a fact (when the input is the empty list, return 0) and the second one is a rule (divide the list in (head:tail) and apply a rule afterwards). This is called **"pattern matching"** and is the most Haskell-ish way to express a condition, which can be informally seen as "if the arguments look like this, the result is…"

Haskell takes advantage of "tail call optimisation" to ensure that excessively used recursion does not consume too much memory.

Prefix and Infix notation

Functions in Haskell default to prefix syntax, meaning that the function being applied is at the beginning of the expression. While this is the default syntax, not all functions are "prefixed". For instance, operators (that are functions) can be used in either prefix style (have to be warped in parentheses) or infix style (default).

```
(div 6 2)  * (8 `mod` 3) + ((*) 100 100)
```

Due to the interaction of parentheses, currying and infix syntax, negative numbers get special treatment in Haskell (cannot mix two infix notations). To work with negative numbers, the programmer must wrap the constants in parentheses.

```
123 + (-6)
```

Basic functions

```
-- inline comment
{- inline block comment -}
6 + 6 -- 12
2 * 2 -- 4
6 - 2 -- 4
11 / 8 -- 1.375
11 `div` 8 -- 1
not False -- True
1 == 1 -- True
2 /= 2 -- False
True && False -- False
False || True -- True
4 < 2 -- False
11 >= 5 -- True
```

A more obfuscated function to analyze

```
multiplyListBy :: (Num a) => a -> [a] -> [a]
multiplyListBy =
  \x -> \y ->
    case y of
      [] -> []
      (h:t) -> [(*) h x] ++ multiplyListBy x t
```

This function takes a number **y** and a list $\mathbf{x_1}$, $\mathbf{x_2}$, …, $\mathbf{x_n}$ and returns a list **y** * $\mathbf{x_1}$, **y** * $\mathbf{x_2}$, …, **y** * $\mathbf{x_3}$.

Compared to the first line, this one looks a bit stranger. This one is read as "multiplyListBy is of type 'a' to '[a]' to '[a]', where 'a' is a member of class Num". **[a]** defines a list that contains variables of type **a**. "`a -> [a] -> [a]`" can be clarified if it is seen as "`a -> ([a] -> [a])`", which can be informally translated to "a number will be used to transform a list into list".

Here, the function arguments seem to be missing, but they are actually passed using lambda abstraction (`\x -> \y ->`). Furthermore, the pattern matching in function definitions has been replaced by **case** expression for choosing what value should be returned.

Both Let and Where are used to introduce components of expressions. The contrast is that **let** introduces an *expression*, so it can be used wherever you can have an expression, but **where** is a *declaration* and is bound to a surrounding syntactic construct. Declarations are top-level bindings which allows us to name expressions.

It is a matter of taste on using either declaration style or expression style. In the declaration style, you formulate an algorithm in terms of several equations that shall be satisfied. In the expression style, you compose big expressions from small ones.

As illustration for the two styles, a function "filter" can be implemented as following:

Declaration style

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
   | p x = x : rest
   | otherwise = rest
   where
     rest = filter p xs
```

Expression style

```
filter :: (a -> Bool) -> [a] -> [a]
filter =
   \p -> \ xs ->
      case xs of
         [] -> []
         (x:xs) ->
            let rest = filter p xs
            in  if p x
                  then x : rest
                  else rest
```

These are the characteristic elements of both styles:

Declaration style	Expression style

where clause		**let** expression	
function arguments on left-hand side	`f x = x * x`	lambda abstraction	`f = \x -> x * x`
pattern matching in function definitions	`f [] = 0`	case expression	`f x = case x of` `  [] -> 0`
Guards on function definitions	`f [x] | x > 0 = 'a'`	if expression	`f [x] = if x > 0 then 'a' else ...`

Haskell types (Carmen Pintea)

Types play an important role in the readability, safety, and maintainability of Haskell code as they allow us to classify and delimit data, thus restricting the forms of data our programs can process. In the next sections the relevant details about the way of categorizing values are going to be described.

It is easy to find out the type of a value, expression, or function in GHCi. We do this with the *:type* command (or just :t in some compilers).

```
Prelude> :type 'a'
'a' :: Char
```

Basic Data Types

Int: This type is a fixed-precision integer.
Integer: This type is also for integers, but this one supports arbitrarily large (or small) numbers (sort of a "lazy" type).
Float: This is the type used for single-precision floating point numbers.
Double: This is a double-precision floating point number type. It has twice as many bits with which to describe numbers as the Float type.
Rational: This is a fractional number that represents a ratio of two integers. The value *1 / 2 :: Rational* will be a value carrying two Integer values, the numerator 1 and the denominator 2, and represents a ratio of 1 to 2.

Tuples
Tuple is a type that allows you to store and pass around multiple values as if it were just one. Tuples have a distinctive, built-in syntax that is used at both type and term levels, and each tuple has a fixed number of constituents. We refer to tuples by the number of values in each tuple: the two-tuple or pair, for example, has two values inside it, (x, y); the three-tuple or triplet has three, (x, y, z), and so on.

Lists

Lists are another type used to contain multiple values within a single value. However, they differ from tuples in three important ways: First, all elements of a list must be of the same type. Second, lists have their own distinct [] syntax. Like the tuple syntax, it is used for both the type constructor in type signatures and at the term level to express list values. Third, the number of values that will be in the list isn't specified in the type — unlike tuples where the arity is set in the type and immutable.

```
Prelude> let p = "This"
Prelude> let sentence = [p, "is", "a list"]
Prelude> sentence
["This","is","a list"]
Prelude> let sentenceWithPunctuation = sentence ++ ["!"]
Prelude> sentenceWithPunctuation
["This","is","a list","!"]
```

Haskell typeclasses

A **typeclass** is a set of operations defined with respect to a polymorphic type. Typeclasses and types in Haskell are, in a sense, opposites. Where a declaration of a type defines how that type in particular is created, a declaration of a typeclass defines how a set of types are consumed or used in computations. Typeclasses allow us to generalize over a set of types in order to define and execute a standard set of features for those types.

When a type has an instance of a typeclass, values of that type can be used in the standard operations defined for that typeclass. In Haskell, typeclasses are **unique** pairings of class and concrete instance. This means that if a given type *a* has an instance of Eq, it has only one instance of Eq.

Eq is the typeclass used to implement the equality in Haskell. The functions its members implement are == and /=. So if there's an Eq class constraint for a type variable in a function, it uses == or /= somewhere inside its definition. All the types we mentioned previously except for functions are part of Eq, so they can be tested for equality.

```
Prelude> :info Eq
class Eq a where
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool\
```

First, it tells us we have a typeclass called Eq where there are two basic functions, equality

and inequality, and gives their type signatures.

Ord is for types that have an ordering.

```
Prelude> :info Ord
class Eq a => Ord a where
compare :: a -> a -> Ordering
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
max :: a -> a -> a
min :: a -> a -> a
```

Num is a numeric typeclass. Its members have the property of being able to act like numbers.

```
Prelude> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Other important typeclasses are: **Show**, **Read**, **Enum**, **Integral, Bounded**.

Polymorphism in Haskell means being able to write code in terms of values which may be one of several, or any, type. Polymorphism in Haskell is either parametric or constrained. The identity function, id, is an example of a **parametrically** polymorphic function:

```
id :: a -> a
id x = x
```

Here id works for a value of any type because it doesn't use any information specific to a given type or set of types. Whereas, the following function *isEqual*:

```
isEqual :: Eq a => a -> a -> Bool
isEqual x y = x == y
```

Is polymorphic, but **constrained** or bounded to the set of types which have an instance of the Eq typeclass.

Understanding type signatures

When we query the types of numeric values, we see *typeclass* information instead of a concrete type, because the compiler doesn't know which specific numeric type a value is until the type is either declared or the compiler is forced to infer a specific type based on the function. For example, 14 may look like an integer, but that would only allow us to use it in computations that take integers (and not, say, in fractional division). For that reason, the compiler gives it the type with the **broadest applicability (most polymorphic)** and says it's a constrained polymorphic Num a => a value:

```
Prelude> :type 14
14 :: Num a => a
```

Next, let's look at the types of some arithmetic functions. The act of wrapping an infix operator in parentheses allows us to use the function just like a normal prefix function, including being able to query the type:

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

The arrow, (->), is the type constructor for functions in Haskell. This means that the addition takes one numeric argument, adds it to a second numeric argument of the same type, and returns a numeric value of the same type as a result

In Haskell we can define **type synonyms**. A type synonym is a new name for an existing type. Values of different synonyms of the same type are entirely compatible. In Haskell you can define a type synonym using *type*:

```
type MyChar = Char
```

Consider the case when you are constantly dealing with lists of three-dimensional points. For instance, you might have a function with type [(Double,Double,Double)] -> Double -> [(Double,Double,Double)]. However, typing [(Double,Double,Double)] all the time gets very tedious. To get around this, you can define a type synonym:

```
type List3D = [(Double,Double,Double)]
```

Haskell variables (Pletosu Cosmin)

In Haskell, a variable is a name for some valid expression. The word “variable” used to describe Haskell variables is somehow misleading, because all variables are actually immutable. Instead, the concept of a Haskel variable is much closer to the mathematical sense of the word, a symbol used to represent an arbitrary element of a set. (e.g. one might use *pi* to stand for 3.1415… or X^3 to represent the formula of some computation). These values do not change in the middle of performing a mathematical calculation or proof.

The operator = is used to assign a value to a variable (pi = 3.1415). The scope of these variables can be controlled using either let or where clauses.

Moreover, in Haskell the term “variable” can be also used to denote formal parameters to functions. This corresponds to the notion of “variable” defined in lambda calculus.

Scopes and static nesting

To define functions inside other functions (static nesting), the keyword "where" can be used to include procedures and variables. A function / variable must be used before it is defined in the "where" clause. (“let” can also be used, it is only a matter of style, as described above in **“Where and Let”**, **Syntax** section)

Variables and functions from lower levels can be used from upper ones. These variables are called “free variables”. Moreover, Haskell allows us to redefine both functions and variables in higher-level functions. Because functions and variables can be redefined, here we meet the concept of "**hole-in-scope**".

The distinction between bond variables and free variables is important when talking about how expressions are evaluated because we have to be careful not to shadow free variables. A rule of thumb is that the value of a free variable affects the value of the expression, so a “hole-in-scope” may alter the expected value, leading to evil logical errors.

Memory allocation

Almost everything in Haskell is placed on heap, except when the compiler decides that it can use the stack for some optimizations. The programmer cannot manipulate the memory by any means and a Garbage Collector manages the allocated memory. The memory implementation details are hidden from the programmer as it is not he’s duty to take care of where variables have been allocated.

Functions in Haskell (Pletosu Cosmin)

Partially applied functions

Consider the type declaration for a "two-argument" function:

```
doStuff :: Int -> [Int] -> Int
```

This is actually parsed as:

```
doStuff :: Int -> ([Int] -> Int)
```

i.e. doStuff is actually a function that takes a String and returns a function [String] -> String. When a function has values applied to some (but not all) of its arguments, it is referred to as a partial function.

When you need to pass a function as an argument, a partially-applied function can be used.

```
divisors n = filter (divides n) [2..(n `div` 2)]
  where divides a b = (a `mod` b) == 0
```

Curried functions

In lambda calculus functions use only a single input. An ordinary function that requires two inputs can be reworked into an equivalent function that accepts a single input, and as output returns another function, that in turn accepts a single input. For example

```
(x, y) -> x + y
```

can be rewritten as

```
x -> y -> x + y
```

Function application of the first function above to the arguments (5, 2), yields at once

```
((x, y) -> x + y)(5, 2) = 7
```

whereas evaluation of the curried version requires one more step

```
((x -> (y -> x + y))(5))(2) = (y -> 5 + y)(2) = 7
```

Manipulating functions

Once you start to think of functions as "values", there are many useful ways to manipulate them (two of them were actually explained above).

The function **.** (called composition function) is similar to the definition in mathematics ("fog" notation). It returns a function that takes in an input, passes it to the function **g**, and then pipes the result of **g** into **f**.

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

The **$** function is a function used to reduce the amount of parentheses needed (sometimes the number of parentheses used in a functional programming language can give headaches).

```
not $ 3 < 5
```

is the exact same as

```
not (3 < 5)
```

Note that $ is not a special operator - it's just a simple function.

Lambda expressions

Lambda expressions or anonymous functions are functions without a name. They are Lambda abstractions (from Lambda Calculus) and might look like this: \x -> x + 1. (That backslash is Haskell's way of expressing a λ and is supposed to look like a Lambda.)

Sometimes it is more convenient to use a lambda expression rather than giving a function a name. For instance, if I wanted to add one to each element of a list, here's one way to do it (without anonymous functions):

```
addOneList lst = map addOne' lst
  where addOne' x = x + 1
```

But here's another way, where we pass an anonymous function to map rather than a named one.

```
addOneList' lst = map (\x -> x + 1) lst
```

Higher order functions

Haskell functions can take functions as parameters and return functions as return values. A function that does either of those is called a higher order function. Higher order functions aren't just a part of the Haskell experience, they pretty much are the Haskell experience. It turns out that if you want to define computations by defining what stuff is instead of defining steps that change some state and maybe looping them, higher order functions are indispensable. They're a really powerful way of solving problems and thinking about programs.

Next I will present some useful Haskell higher order functions that are implemented in the standard library.

zipWith

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

zipWith' applies a function to two zipped lists. The first parameters is a **function** that takes two variables and produces a result. The second and the third parameters are lists, and a list is returned.

flip

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x
```

flip' takes a function and returns a function that is like our original function, but the first two arguments are flipped.

Example:

```
flip' zip [1, 2, 3, 4] "abcd"
```

will produce [('a', 1), ('b', 2), ('c', 3), ('d', 4)]

map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

map takes a function, a list and applies that function to every element in the list, producing a new list.

Example:

```
map (\x -> x + 10) [1, 2, 3, 4]
```

will produce [11, 12, 13, 14]

filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

filter takes a predicate (a predicate is a function that tells whether something is true or not) and a list and then returns the list of elements that satisfy the predicate.

Haskell and monads (Adonis Bodea)

What is a monad?

A monad is a design pattern which aims to structure computations in terms of values and sequences of computations using those values. Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required

Intro in Haskell monads

In Haskell a monad is represented as a type constructor (call it m), a function that builds values of that type (a -> m a), and a function that combines values of that type with computations that produce values of that type to produce a new computation for values of that type (m a -> (a -> m b) -> m b). It is customary to call the monad type constructor "m" when discussing monads in general. The function that builds values of that type is traditionally called "return" and the third function is known as "bind" but is written ">>=".

Using a container analogy, the type constructor "m" can be seen as a container that can hold different values, while "m a" is a container holding a value of type a. The "return" function puts a value into a monad container. The bind or " >>=" function takes the value from a monad container and passes it to a function to produce a monad container containing a new value, possibly of a different type. By adding logic to the binding function, a monad can implement a specific strategy for combining computations in the monad.

To understand the concept better, let's walk through an example. We will firstly tackle the problem without monads and then with their help. Suppose that we are writing a program to keep track of sheep clonation. We would certainly want to know the genetic history of all of our sheep, so we would need "mother" and "father" functions. But since these are cloned sheep, they may not always have both a mother and a father!

We would represent the possibility of not having a mother or father using the "Maybe" type constructor in our Haskell code:

```
type Sheep = ...

father :: Sheep -> Maybe Sheep
father = ...

mother :: Sheep -> Maybe Sheep
mother = ...
```

Then, defining functions to find grandparents is a little more complicated, because we have to handle the possibility of not having a parent:

```
maternalGrandfather :: Sheep -> Maybe Sheep
```

```
maternalGrandfather s = case (mother s) of
                          Nothing -> Nothing
                          Just m  -> father m
```

For the other grandparent combinations, the functions look similar. However, it gets even worse if we want to find great grandparents:

```
mothersPaternalGrandfather :: Sheep → Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
                                 Nothing -> Nothing
                                 Just m  -> case (father m) of
                                              Nothing -> Nothing
                                              Just gf -> father gf
```

Aside from being ugly, unclear, and difficult to maintain, this is just too much work. It is clear that a "Nothing" value at any point in the computation will cause "Nothing" to be the final result, and it would be much nicer to implement this notion once in a single place and remove all of the explicit "case" testing scattered all over the code. This will make the code easier to write, easier to read and easier to change. So good programming style would have us create a "combinator" that captures the behavior we want:

```
-- comb is a combinator for sequencing operations that return Maybe
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing  _ = Nothing
comb (Just x) f = f x

-- now we can use `comb` to build complicated sequences
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = (Just s) `comb` mother `comb` father `comb` father
```

Much better. Notice also that the "comb" function is entirely polymorphic, i.e. it is not specialized for "Sheep" in any way. In fact, *the combinator captures a general strategy for combining computations that may fail to return a value.* Thus, we can apply the same combinator to other computations that may fail to return a value, such as database queries or dictionary lookups.

"Fully Haskell" Monads

Nevertheless, what we described here is not a "fully Haskell" monad. To obtain one, we have to make it conform to the monad framework built into the language, which provides a standard "Monad" class (not the OOP sense of a class, as mentioned earlier) that defines the names and signatures of the two monad functions "return" and ">>=". It is not strictly necessary to make your monads instances of the Monad class, but it is a good idea since Haskell has special support for Monad instances built into the language and making your monads instances of the Monad class will allow you to use these features to write cleaner and more elegant code (i.e. to write "fully Haskell" monads).

The standard "Monad" class definition in Haskell looks something like this:

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
```

```
    return :: a -> m a
```

Continuing with the previous example, we will now see how the Maybe type constructor fits into the Haskell monad framework as an instance of the Monad class.

Recall that our "Maybe" monad used the "Just" data constructor to fill the role of the monad "return" function and we built a simple combinator to fill the role of the monad ">>=" binding function. We can make its role as a monad explicit by declaring "Maybe" as an instance of the "Monad" class (even though "Maybe" is already defined as an instance of the "Monad" class in the standard prelude, so you don't need to do it yourself):

```
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

Once we have defined Maybe as an instance of the Monad class, we can use the standard monad operators to build the complex computations:

```
mothersMaternalGrandfather :: Sheep -> Maybe Sheep
mothersMaternalGrandfather s = (return s) >>= mother >>= father
```

```
mothersMaternalGrandfather :: Sheep -> Maybe Sheep
mothersMaternalGrandfather s = (return s) >>= father >>= mother >>= mother
```

Another advantage of membership in the "Monad" class is the Haskell support for "do" notation. "Do" notation is an expressive shorthand for building up monadic computations, similar to the way that list comprehensions are an expressive shorthand for building computations on lists. Any instance of the "Monad" class can be used in a do-block in Haskell.

In short, the do notation allows you to write monadic computations using a pseudo-imperative style with named variables. The result of a monadic computation can be "assigned" to a variable using a left arrow <- operator. Then using that variable in a subsequent monadic computation automatically performs the binding. The type of the expression to the right of the arrow is a monadic type "m a". The expression to the left of the arrow is a pattern to be matched against the value *inside* the monad. (x:xs) would match against Maybe [1,2,3], for example.

Nevertheless, the do notation is simply syntactic sugar. There is nothing that can be done using do notation that cannot be done using only the standard monadic operators. However, do notation is cleaner and more convenient in some cases, especially when the sequence of monadic computations is lon

Here is a sample of do notation using the Maybe monad and our sheep example:

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = do m  <- mother s
                                  gf <- father m
                                  father gf
```

Monad laws and other types of monads

The concept of a monad comes from a branch of mathematics called category theory. While it is not necessary to know category theory to create and use monads, we do need to obey a small bit of mathematical formalism. So, to create a proper monad, it is not enough just to declare a Haskell instance of the "Monad" class with the correct type signatures. To be a proper monad, the "return" and ">>=" functions must work together according to three laws:

1. (return x) >>= f ==== f x
2. m >>= return ==== m
3. (m >>= f) >>= g ==== m >>= (\x -> f x >>= g)

The first law requires that "return" is a left-identity with respect to ">>=". The second law requires that "return" is a right-identity with respect to ">>=". The third law is a kind of associativity law for >>=. Obeying the three laws ensures that the semantics of the do-notation using the monad will be consistent.

Any type constructor with "return" and "bind" operators that satisfy the three monad laws is a monad. In Haskell, the compiler does not check that the laws hold for every instance of the Monad class. It is up to the programmer to ensure that any Monad instance they create satisfies the monad laws.

When we gave the definition for the monad class earlier we have chosen to omit something and only present the minimal one. The full definition of the "Monad" class actually includes two additional functions: "fail" and ">>".

The default implementation of the "fail" function is:

```
fail s = error s
```

You do not need to change this for your monad unless you want to provide different behavior for failure or to incorporate failure into the computational strategy of your monad. The "Maybe" monad, for instance, defines "fail" as:

```
fail _ = Nothing
```

so that "fail" returns an instance of the "Maybe" monad with meaningful behavior when it is bound with other functions in the "Maybe" monad.

The "fail" function is not a required part of the mathematical definition of a monad, but it is included in the standard "Monad" class definition because of the role it plays in Haskell's do notation. The "fail" function is called whenever a pattern matching failure occurs in a do block. Let's consider the following function:

```
fn :: Int -> Maybe String
fn idx = do let l = [Just "abc", Nothing, Just [], Just ['a','x','y']]
            (x:xs) <- l!!idx   -- a pattern match failure will call "fail"
            return xs
```

So in the code above, "fn 0" has the value "bc", but "fn 1" and "fn 2" both have the value Nothing, which is due to pattern match failing and causing the "fail" function to trigger.

The “>>” function is a convenience operator that is used to bind a monadic computation that does not require input from the previous computation in the sequence. It is defined in terms of “>>=”:

```
(>>) :: m a -> m b -> m b
m >> k = m >>= (\_ -> k) -- a lambda function used for overcoming the lack of input
```

As the standard “Monad” class is defined, there is no way to get values out of a monad. That is not an accident. Nothing prevents the monad author from allowing it using functions specific to the monad. For instance, values can be extracted from the Maybe monad by pattern matching on “Just x” or using the “fromJust” function.

By not requiring such a function, the Haskell “Monad” class allows the creation of one-way monads. One-way monads allow values to enter the monad through the “return” function (and sometimes the “fail” function) and they allow computations to be performed within the monad using the bind functions “>>=” and “>>”, but they do not allow values back out of the monad.

The “IO” monad is a familiar example of a one-way monad in Haskell. Because you can't escape from the “IO” monad, it is impossible to write a function that does a computation in the IO monad but whose result type does not include the “IO” type constructor. This means that *any* function whose result type does not contain the “IO” type constructor is guaranteed not to use the “IO” monad. Other monads, such as Maybe, do allow values out of the monad. So it is possible to write functions which use these monads internally but return non-monadic values.

The wonderful feature of a one-way monad is that **it can support side-effects in its monadic operations but prevent them from destroying the functional properties of the non-monadic portions** of the program.

Consider the simple issue of reading a character from the user. We cannot simply have a function “readChar :: Char”, because it needs to return a different character each time it is called, depending on the input from the user. It is an essential property of Haskell as a pure functional language that all functions return the same value when called twice with the same arguments. But it *is* ok to have an I/O function “getChar :: IO Char” in the “IO” monad, because it can only be used in a sequence within the one-way monad. There is no way to get rid of the “IO” type constructor in the signature of any function that uses it, so the “IO” type constructor acts as a kind of tag that identifies all functions that do I/O. Furthermore, such functions are only useful within the “IO” monad. So a one-way monad effectively creates an isolated computational domain in which the rules of a pure functional language can be

relaxed. **Functional computations can move into the domain, but dangerous side-effects and non-referentially-transparent functions** cannot escape from it.

Beyond the three monad "axioms" stated above, some monads obey additional laws. These monads have a special value "mzero" and an operator "mplus" that obey four additional laws:

1. mzero >>= f == mzero
2. m >>= (\x -> mzero) == mzero
3. mzero `mplus` m == m
4. m `mplus` mzero == m

It is easy to remember the laws for "mzero" and "mplus" if you associate "mzero" with 0, "mplus" with +, and ">>=" with × in ordinary arithmetic.

Monads which have a "zero" and a "plus" can be declared as instances of the "MonadPlus" class in Haskell:

```
class (Monad m) ==> MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Continuing to use the "Maybe" monad as an example, we see that the "Maybe" monad is an instance of MonadPlus:

```
instance MonadPlus Maybe where
  mzero           = Nothing
  Nothing `mplus` x = x
  x `mplus` _       = x
```

This identifies "Nothing" as the zero value and says that adding two "Maybe" values together gives the first value that is not "Nothing". If both input values are "Nothing", then the result of "mplus" is also "Nothing".

The "mplus" operator is used to combine monadic values from separate computations into a single monadic value. Within the context of our sheep-cloning example, we could use

"Maybe"'s "mplus" to define a function, "parent s = (mother s) `mplus` (father s)", which would return a parent if there is one, and "Nothing" is the sheep has no parents at all. For a sheep with both parents, the function would return one or the other, depending on the exact definition of "mplus" in the "Maybe" monad.

Useful monads in Haskell

Monad	Type of computation	Combination strategy for >>=
Identity	***N/A — Used with monad transformers***	**The bound function is applied to the input value.**

Maybe	**Computations which may not return a result**	**“Nothing” input gives “Nothing” output “Just x” input uses “x” as input to the bound function.**
Error	**Computations which can fail or throw exceptions**	**Failure records information describing the failure. Binding passes failure information on without executing the bound function, or uses successful values as input to the bound function.**
[] (List)	**Non-deterministic computations which can return multiple possible results**	**Maps the bound function onto the input list and concatenates the resulting lists to get a list of all possible results from all possible inputs.**
IO	**Computations which perform I/O**	**Sequential execution of I/O actions in the order of binding.**
State	**Computations which maintain state**	**The bound function is applied to the input value to produce a state transition function which is applied to the input state.**
Reader	**Computations which read from a shared environment**	**The bound function is applied to the value of the input using the same environment.**
Writer	**Computations which write data in addition to computing values**	**Written data is maintained separately from values. The bound function is applied to the input value and anything it writes is appended to the write data stream.**
Cont	**Computations which can be interrupted and restarted**	**The bound function is inserted into the continuation chain.**

Separate compilation in Haskell (Adonis Bodea)

As mentioned earlier, the usual way in which a Haskell program is structured is by modules. Each module is placed in a file with the same name and the extension .hs (or .lhs). Each module is made up of symbols, some of which are imported, some of which the module chooses which to expose to other modules and some of which are local. Symbols are exposed by specifying them between parentheses immediately after the module name (ex. “module Algorithm(g)” - the Algorithm module exposes only the symbol g), and they are imported by declaring in which module it imports (ex. “import Sort (quicksort)” - a module specifies that it needs the quicksort symbol from the module Sort; if all the symbols from the module Sort would have been needed, the syntax “import Sort” was enough to achieve that). If the content of a file is not inside any module, it is assumed to be implicitly part of the main module (which in particular is the one which contains the main function).

Similar to C++, the compiler of Haskell generates as a result an object file (usually with the extension .o). However, besides this, it also generates an interface file (usually with the extension .hi). The interface file contains the information that the compiler needs in order to compile further modules that depend on this module and it contains things like the types of exported functions, definitions of data types, and so on.

Analogous with the Java compiler, if the current module depends on symbols from other modules, it would force their compilation too, if their .hi files are not already present. However, this is achieved only if the compilation command is called with the option "--make" .In the case this option is not specified, the compiler assumes that the .hi files for the module exist and concludes that the symbols can not be found if it can not find them . The search path for the compiler is the following: a list of directories, which by default contains only the current directory but can be modified with certain commands, in which the compiler looks for files with the extension .hi (if the --make option is not passed), and for files with the extension .hs or .lhs (if the option is passed).

Because of the way certain modules are imported inside others, circular dependencies can appear (for example the module A import functions from module B and module B imports functions from module A). Haskell offers a mechanism for dealing with this kind of problem: .hs-boot file. Every cycle in the module import graph must be broken by a hs-boot file which must be compiled first, before the source is compiled. Each .hs-boot file belongs to a source file (and module by extension) and it must live in the same directory as its parent source file. This kind of file acts as a messenger for the compiler in the sense that it informs him with the bare minimum of information needed to get the bootstrapping process started.

Haskell also offers the possibility to export and import functions from other programming languages through the Foreign Function Interface (FFI) as well. The basic way to import a function is illustrated in the following example:

```
foreign import ccall "exp" c_exp :: Double -> Double
```

The keyword foreign is used to denote the fact that a communication with another programming language is established. After the import keyword (which signals the fact that the function comes from the outside), the function call standard is specified. The name of the outside function and the name of the function inside the Haskell module are then specified. The last thing to be needed is the signature of the function .

A function can be exported from Haskell as shown below, with all the keywords mentioned above maintaining their purpose (export signals that the function is available outside):

```
foreign export ccall triple :: Int -> Int
```

Concurrency and parallelism in Haskell (Carmen Pintea)

GHC implements some major extensions to Haskell to support concurrent and parallel programming. *Parallelism* means running a Haskell program on multiple processors, with the goal of improving performance. Ideally, this should be done invisibly, and with no semantic changes. *Concurrency* means implementing a program by using multiple I/O-performing threads. While a concurrent Haskell program can run on a parallel machine, the primary goal of using concurrency is not to gain performance, but rather because that is the simplest and most direct way to write the program.

Concurrent Haskell

Concurrency in Haskell is mostly done with Haskell threads. Haskell threads are user-space threads that are implemented in the runtime. Haskell threads are much more efficient in terms of both time and space than Operating System threads. Apart from traditional synchronization primitives like semaphores, Haskell offers Software Transactional Memory which greatly simplifies concurrent access to shared memory. To the programmer, Concurrent Haskell introduces no new language constructs; rather, it appears simply as a library, **Control.Concurrent.***.

The functions exported by this library include:

- Forking and killing threads.
- Sleeping.
- Synchronised mutable variables, called MVars
- Support for bound threads; see the paper Extending the FFI with concurrency.

The fundamental action in concurrency is **forking a new thread** of control. In Concurrent Haskell, this is achieved with the *forkIO* operation:

```
forkIO :: IO () -> IO ThreadId
```

The *forkIO* operation takes a computation of type IO () as its argument; that is, a computation in the IO monad that eventually delivers a value of type (). The computation passed to forkIO is executed in a new thread that runs concurrently with the other threads in the system. If the thread has effects, those effects will be interleaved in an indeterminate fashion with the effects from other threads.

Let’s see an example:

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering  -- Put the output Handle into
nonbuffered mode, so that we can see the interleaving more clearly.
  forkIO (replicateM_ 100000 (putChar 'A')) -- Create a thread to print
the character A 100,000 times.
  replicateM_ 100000 (putChar 'B') -- In the main thread, print B 100,000
times.
```

The output will be of the form:
BBBBABABABABABABABABABABABABABABABABABABABABABABABA…..

Thread sleeping is another important functionality, provided through *threadDelay*() which takes an argument representing a number of microseconds and waits for that amount of time before returning:

```
threadDelay :: Int -> IO ()
```

For example, the following program uses *threadDelay* to implement a remainder functionality. The user enters a number of seconds, and after the specified time has elapsed, the program prints a message. Any number of reminders can be active simultaneously:

```
import Control.Concurrent
import Text.Printf
import Control.Monad

main =
  forever $ do
    s <- getLine           -- read the number of seconds
    forkIO $ setReminder s -- start thread

setReminder :: String -> IO ()
setReminder s = do
  let t = read s :: Int
  printf "Ok, I'll remind you in %d seconds\n" t
  threadDelay (10^6 * t)                    -- sleep
  printf "%d seconds is up! BING!\BEL\n" t -- the time expired
```

Is important to know that in Haskell the program terminates when main returns, even if there are other threads still running. The other threads simply stop running and cease to exist after main returns.

Thread communication is realised using **MVars,** the basic communication mechanism provided by Concurrent Haskell. The API for MVar is as follows:

```
data MVar a  -- abstract

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

An MVar can be thought of as a box that is either empty or full. The newEmptyMVar operation creates a new empty box, and newMVar creates a new full box containing the value passed as its argument. The takeMVar operation removes the value from a full MVar and returns it, but waits (or blocks) if the MVar is currently empty. Symmetrically, the putMVar operation puts a value into the MVar but blocks if the MVar is already full.

The following sequence of small examples should help to illustrate how MVars work. First, this program passes a single value from one thread to another:

```
main = do
  m <- newEmptyMVar
  forkIO $ putMVar m 'X'
  r <- takeMVar m
  print r
```

The MVar is empty when it is created, the child thread puts the value x into it, and the main thread takes the value and prints it. If the main thread calls takeMVar before the child thread has put the value, no problem: takeMVar blocks until the value is available.

Parallel Haskell

Parallel Haskell is aimed at providing access to multiple processors in a natural and robust way. The parallel programming models provided by Haskell do succeed in eliminating some error-prone aspects traditionally associated with parallel programming:

- Parallel programming in Haskell is **deterministic**: the parallel program always produces the same answer, regardless of how many processors are used to run it. So parallel programs can be debugged without actually running them in parallel. Furthermore, the programmer can be confident that adding parallelism will not introduce race conditions or deadlocks that would be hard to eliminate with testing.

- Parallel Haskell programs are **high-level and declarative** and do not explicitly deal with concepts like synchronization or communication. The programmer indicates where the parallelism is, and the details of actually running the program in parallel are left to the runtime system.

- By embodying fewer operational details, parallel Haskell programs are **abstract** and are therefore likely to work on a wide range of parallel hardware.

The main functionalities for creating parallelism are provided by the module *Control.Parallel.Strategies*. Parallelism is expressed using the *Eval monad*, which comes with two operations, *rpar* and *rseq*. The *rpar* combinator creates parallelism: it says, "My argument could be evaluated in parallel"; while rseq is used for forcing sequential evaluation: It says, "Evaluate my argument and wait for the result." The Eval monad provides a runEval operation that performs the Eval computation and returns its result.

To see the effects of *rpar* and *rseq*, suppose we have a function *f*, along with two arguments to apply it to, *x* and *y*, and we would like to calculate the results of *f x* and *f y* in parallel. Let's say that *f x* takes longer to evaluate than *f y*.

Var1:

```
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    return (a,b)
```

In this case *f x* and *f y* begin to evaluate in parallel, while the return happens immediately. It doesn't wait for either *f x* or f y to complete. The rest of the program will continue to execute while *f x* and *f y* are being evaluated in parallel. If we expect to be generating more parallelism soon and don't depend on the results of either operation, this approach makes sense.

Var2:

```
runEval $ do
     a <- rpar (f x)
     b <- rseq (f y)
     return (a,b)
```

Here *f x* and *f y* are still evaluated in parallel, but now the final return doesn't happen until *f y* has completed. This is because we used *rseq*, which waits for the evaluation of its argument before returning.

IORefs (and why Haskell is not actually purely functional) (Pleto)

IORef gives you the ability to assign a reference to a variable in the IO monad. This at least decorates your code in such a way that it's obvious to you, the developer and Haskell that there will be side effects. This affects the purity of the extended Haskell language, but these "side effects" can be used only in monads, separating the "core" language and the "unsafe" one. Therefore, Haskell's core purity is not affected!

IORef are usually used in concurrent algorithms to have a shared state between threads in Haskell.

```
data IORef a -- pointer to a value of type a
newIORef :: a -> IO (IORef a) -- allocate new pointer & init.
readIORef :: IORef a -> IO a -- dereference
writeIORef :: IORef a -> a -> IO () -- modify location
modifyIORef :: IORef a -> (a -> a) -> IO () -- update value
atomicModifyIORef :: IORef a -> (a -> (a, b)) -> IO b
```

As a proof of concept, to implement a "Bank account transfer problem" using IORefs, the code would look like this:

```
transfer :: IORef Int -> IORef Int -> Int -> IO ()
transfer fro to n = do
 bal_fro <- readIORef fro
 bal_to <- readIORef to
 writeIORef fro (bal_fro - n)
 writeIORef to (bal_to + n)
```

Team's impressions of Haskell

Adonis Bodea

Studying Haskell was a bit of a challenge for me. Coming with a background of mostly imperative and object oriented languages, the shift to a functional environment was quite drastic (I did not take Lisp that seriously since, in my opinion, hacks to treat programs in an imperative style can still be used). Moreover, Haskell is a pure functional language, so functions like "set", "setq" which are found in Lisp (and I used vigorously) have no place here since everything is immutable. This constraint (or feature) in particular was the most difficult to get accustomed to, since I really relished the assignment (and reassignment) operations of the aforementioned paradigms and up until now I always thought how to reuse my variables and how to get my best out of them. Haskell forced me to toss that aside and think more mathematically, which I enjoyed in the end. Furthermore, I got a chance to study a topic that I postponed way too much given how many recommended me to do it, namely monads (the majority of recommendations came from friends who study at Oxford, where Haskell is taught in the first year of their degree). After going over it, both in mathematics and as a design pattern in computer science, I think understanding monads can improve the skills of any programmer since it truly coerces to squeeze out as much general factorization as possible from your code.

As a general perspective of Haskell, I think it can be a great didactical material for teaching the functional paradigm in schools or universities (undoubtedly better than Lisp). However, when it comes to building software projects I am somewhat reserved. They do not seem to make the best out of the von Neumann architecture which dominates the everyday computational systems available, all the while forcing us to treat every problem we try to solve way too mathematically than it sometimes needs.

Carmen Pintea

By learning the basics of Haskell language, the main qualities which I noticed it promotes are:

- as in any functional language, some solutions are more intuitive, having a form closer to natural expression
- the laziness allows a more elegant and cleaner expression, allowing even the definition of infinite structures that help the programmer to view the program as a series of transformations on data

- bugs are much better controlled by the attributes: pure (no side-effects), strongly typed (no type errors), high-level (no worries about memory management, abstract, portable)

Of course, I also noticed a number of disadvantages. Because it is a high level language with a fairly high level of abstraction, it can take a long time to master the notions, especially without a solid theoretical background. I would definitely not recommend this language to a beginner in programming.

Cosmin Pletosu

Haskell is definitely a hard language to learn, and while it is not the language of choice when developing a new software system, it promotes good practices in current times that can be applied in other, more “permissive” programming languages.

Best practices are obviously not a binary "yes/no" proposition or even a linear scale. There are multiple dimensions to best practices and some languages are better on some dimensions and worse on others. Some dimensions where Haskell performs well are:

- Absence of null
- Immutability
- Strongly typed language
- Generic types
- Treating safe and unsafe code differently

Haskell isn't perfect, though, so in the interests of balance I will also list some areas where Haskell actively rewards worst practices and punishes best practices:

- Poor performance
- Excessive abstraction
- Unchecked exceptions

From my point of view, the modern times require parallel, concurrent and “low level” (system) higher-level languages. Haskell fits the “parallel”, “concurrent” and “higher-level” requirements, but there are better alternatives to choose in production because they are easier to learn (on average, an IT employee works for a company 2 to 3 years) and most programmers are already used to imperative programming languages.