Functional Programming by Examples

A Little Taste of Haskell

Ihar I Hancharenka

EPAM Systems

July-Sept, 2011

Traditional programming languages

In traditional (imperative) programming languages (like C, C++, Java, ...):

```
x = x + 1;
```

- what is **x** mathematically ? it is not a (constant) function !
- fundamental notion: state $\Gamma \in \{\textit{variables}\} \rightarrow \mathbb{N} \cup \mathbb{R} \cup \ldots$
- every statement is a transformation of environmennts $\Gamma \mapsto \Gamma')$

This makes it difficult to reason about a program, i.e. prove its correctness.

Functional programming languages

In a pure functional programming language (Haskell), every function is also a function in the mathematical sense:

- no pointers (and UGLY CrashDump, ACCESS_VIOLATION, NullPointerException and so on...)
- no variables - but constant function (sometimes, imperative PL variables are convertedd to SSA form at compilation phase)
- no loops - recursion as a magic bullet

We can prove properties of programs by the usual mathematical means! (induction, case analysis)
Partial functions are allowed !

The Quicksort algorithm has been invented by SIR C.A.R. Hoare.
At that time - just a Tony Hoare.

Here's a typical C++ implementation:

Example

```cpp
void quickSort(int arr[], int left, int right)
{
        int i = left, j = right;
        int pivot = arr[(left + right) / 2];
        while (i <= j)
        {
                while (arr[i] < pivot) ++i;
                while (arr[j] > pivot) ++j;
                if (i <= j)
                        std::swap(arr[i++], arr[j--]);
        }
        if (left < j) quickSort(arr, left, j);
        if (i < right) quickSort(arr, i, right);
}
```

Let's summarize

Disclaimer #1

The total number of lines is 14

Here's appropriate Haskell implementation

Example

```haskell
qsort          :: Ord a => [a] -> [a]
qsort []       = []
qsort (x:xs) = (qsort lth) ++ [x] ++ (qsort gth) where
         lth = filter (<  x) xs
         gth = filter (>= x) xs
```

Or (the same thing) using list comprehensions

Example

```haskell
qsort          :: Ord a => [a] -> [a]
qsort []       =  []
qsort (x:xs) =  qsort [y | y <- xs, y <  x] ++
                [x                        ] ++
                qsort [y | y <- xs, y >= x]
```

Let's summarize

Disclaimer #2

The total number of lines is 5
The difference between C++ and Haskell version is 14 - 5 = 9 lines !!!

Now let's describe the Haskell implementation of quicksort.
But first let's see the result of using quicksort at some sample input data.

Example

```
> qsort [2,5,3,2,1]
[1,2,2,3,5]
```

The numbers in square brackets [] is a regular haskell list (of numbers).
But we can sort list of Char-s, Float-s and other types also.

But what is a List from Haskell point of view?

Definition

```
data [a] = [] | a:[a]
```

This is an example of recursively-defined data structure (has no analogs in C++).
Here we define a base case - [] for an empty list and a recursive one - item (:) list. And here's some examples with syntactic sugar:

Example

```
[1..3] == [1,2,3] == 1:[2,3] == 1:2:3:[]
```

But the Haskell itself always treats is like a degenerated tree

```
(1:(2:(3:[])))
```

Now it's time to look at the first line of our quicksort:

Definition

```
qsort          :: Ord a => [a] -> [a]
```

This is just a type declaration of qsort function.

- `a`

  is a type variable (can be Integer, Float, Char, ...).
- `Ord a =>`

  is a constraint for the type a (it shold support total ordering).
- `[a] -> [a]`

  means that qsort takes a list of some time and return the [sorted] list of the same type.

It worth to be mentioned:

FYI

- It's not needed to put type declaration for all the functions.
- In most cases it will be automatically deduced.
- According to the Hindley-Milner type inference algorithm.

Let's look at the Ord type class (corresponds to something like interface in Java/C/C++):

Definition

```
data Ordering = LT | EQ | GT

-- Minimal complete definition: either 'compare' or '<='.
-- Using 'compare' can be more efficient for complex types.
--
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT
```

Let's continue:

Definition

```
x <  y = case compare x y of { LT -> True;  _ -> False }
x <= y = case compare x y of { GT -> False; _ -> True }
x >  y = case compare x y of { GT -> True;  _ -> False }
x >= y = case compare x y of { LT -> False; _ -> True }

max x y = if x <= y then y else x
min x y = if x <= y then x else y
```

Before moving to the second line of our quicksort we need to tell about recursive functions definition. In mathematics we define recursive functions in the following way:

$$f(n) = \begin{cases} 1, & n = 1, \\ n * f(n-1), & n \geqslant 1 \end{cases}$$

In Haskell we define it by:

Definition

```
factorial 1 = 1
factorial n = n*factorial(n - 1)
```

Or, using syntax sugar:

Definition

```
factorial n = |n == 1        = 1
              |otherwise     = n*factorial(n - 1)
```

Now it's time to look at the second line of our quicksort:

Definition

```
qsort []     = []
```

This is just a base-case of a recursively-defined function.
If we have an empty list - we don't need to sort it - so let's just return it as a result.

The third line of our quicksort is:

Definition

```
qsort (x:xs) = (qsort lth) ++ [x] ++ (qsort gth) where
```

This is a recursive case. (x:xs) is a pattern matching of the input list. Remember the second case of list definition (... a:[a]). It makes possible to do such pattern-matching around a colon (:) operator. This is a common trick in FP - split the list to it's head (element) and tail (a smaller list).

Let's continue with the third line of our quicksort:

Definition

```
qsort (x:xs) = (qsort lth) ++ [x] ++ (qsort gth) where
```

Here [x] is a list of one element - x (for example - [3] or [71]).
$++$ - is a list concatenation operator (sometimes denoted in a literature as $+\!\!\!+$).

Example

```
> [1,3] ++ [6,4,5]
[1,3,6,4,5]
```

(qsort lth) and (qsort gth) is a recursive call with a lists of smaller length.

Let's continue with the fourth line of our quicksort:

Definition

```
        lth = filter (<  x) xs
```

Here filter is one of the standard Haskell well-known higher-order functions (HOF).
In other words - this is a function that takes another functions as it's [first] argument.

Example

```
> filter (<  3) [1,2,3,4,5]
[1,2]
> filter (>= 3) [1,2,3,4,5]
[3,4,5]
> filter (\v -> (v > 1 && v <5)) [1,2,3,4,5]
[2,3,4]
```

(< x) is a section - partially applied (curried) function for the appropriate operator (x - is the bounded (fixed) second argument).
Let's remember Ord type class:

Definition

```
    (<), (<=), (>), (>=) :: a -> a -> Bool
```

All the mentioned functions (operators) takes 2 arguments.
But if we fix (partially apply) one of the argument - the remaining function will be of one argument which returns Bool (True | False).

Example

```
> :t (< 28)
(< 28) :: (Ord a, Num a) => a -> Bool
> (< 28) 27
True
> (< 28) 28
False
```

Here's a definition of standard (Prelude) version of filter higer-order (HOF) function:

Definition

```
-- 'filter', applied to a predicate and a list,
-- returns the list of those elements that
-- satisfy the predicate, i.e.
-- > filter p xs = [ x | x <- xs, p x]
filter :: (a -> Bool) -> [a] -> [a]
filter _pred []     = []
filter pred  (x:xs)
  | pred x          = x : filter pred xs
  | otherwise       = filter pred xs
```

The type declaration of filter tells us that it's a HOF by the bracket around the first argument:

```
filter :: (a -> Bool) -> ...
```

The second argument is a list of type **a** ([a])
The result is also the list of typa **a** which contains the same or fewer elements than the second argument. The next line of filter definition:

```
filter _pred []     = []
```

Is a partial definition for the case then the second argument is an empty list.
The first argument (predicate **_pred**) does not matter in this case.
It is usual to user the underscore symbol - _ for such cases.

The other lines of filter definition is a recursive case:

```
filter pred  (x:xs)
  | pred x          = x : filter pred xs
  | otherwise       = filter pred xs
```

It consists of two parts.
The first one is for the case if predicate **pred** is **True** for the element **x**. In this case we keep the element **x**.

```
x : filter pred xs
```

The second one (**otherwise**) is for for the opposite case (**False**). In this case we skeep(omit) **x**.

```
filter pred xs
```

And process a list of the smaller size (**xs**) recursively.

We've just finished with one of the most famous HOF: filter. But there are also other two ones, **greately** used in functional programming.
But first let's not break a tradition and start from example.
Let's summarize the squares of all the natural numbers starting from one up to **n**.

$$1^2 + 2^2 + ...n^2$$

For the C++ this would be:

Example

```
unsigned sumnats(unsigned n)
{
        unsigned result = 0;
        for (unsigned i = 0; i < n; ++i)
                result += i*i;
        return result;
}
```

For the Haskell this function will be much simpler (**AGAIN!**):

Example

```
sumnats :: Int -> Int
sumnats n = foldr (+) 0 (map (\x -> x^2) [1..n])
```

The comparison is:

Disclaimer #2

The total number of lines for the C++ case is 5
The total number of lines for the Haskell case is 2
The difference between C++ and Haskell version is 7 - 2 = 5 lines !!!

The list comprenehsion [1..n] is just a syntax sugar for the list of successive numbers from 1 to n.
Consider the case n = 5:

Example

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

Now let's describe the following sub-expression:

```
map (\x -> x^2) [1..n]
```

The map HOF - is for applying it's first argument (function) to the list.

Example

```
> map (\x -> x*x) [1..5]
[1,4,9,16,25]
```

Here's a more general description of map:

Definition

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

And this is the formal (source code) definition of map:

Definition

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

We pattern-match on a list constructor (:), convert matching element using f-function and process the rest of the list (xs) recursively.

The foldr (as well as foldl) is the most famous HOF - for traversing elements of the list from right (to left) starting from a specified element (0 in our case):

Example

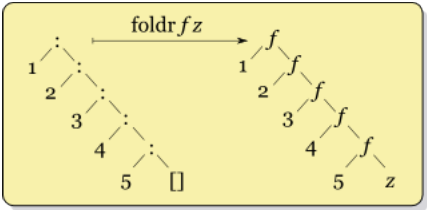
```
> foldr (+) 0 [1,2,3,4,5]
15
> foldr (+) 0 [1,4,9,16,25]
55
```

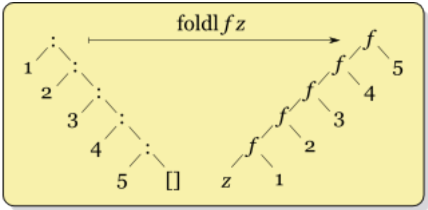
Here's a more general description of foldr:

Definition

```
foldr (op) x0 [x1, x2, ..., xn] ==
  (x1 'op' (x2 'op' ... 'op' (xn 'op' x0)))
-- or, in case of associative operation 'op':
foldr (op) x0 [x1, x2, ..., xn] ==
  x1 'op' x2 'op' ... 'op' xn 'op' x0
```

In graphical form, foldr and foldl can be represented as:





In other words - we replace the empty list ([]) by start element and list constructor (:) by our operation.
And this is the formal (source code) definition of foldr:

Definition

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []      = z
foldr f z (x:xs) = f x (foldr f z xs)
-- foldr op x0 (x:xs) = x 'op' (foldr op x0 xs)
```

Another sample - let's calculate a maximum of list elements. Traditional (C++) approach is:

Example

```
int my_max(size_t cnt, int values[])
{
        int result = values[0];
        for(size_t i = 1; i < cnt; ++i)
                result = values[i] > result ? values[i] : r
        return result;
}
```

The haskell version is:

Example

```
my_max :: Ord a => [a] -> a
my_max = foldr1 max
```

Here foldr1 is a standard Haskell derivation of foldr (using last element of a list as a start one):

Definition

```
-- | 'foldr1' is a variant of 'foldr' that
-- has no starting value argument,
-- and thus must be applied to non-empty lists.
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ [x]    =  x
foldr1 f (x:xs) =  f x (foldr1 f xs)
foldr1 _ []     =  errorEmptyList "foldr1"
```

Fold Universality

Fold is a very universal. Map can be expressed through it:

```
map' f xs = foldr ((:) . f) []) xs
```

Haskell allows to omit all the dupclicated arguments if they appear at the both (left and right) side of definition:

```
map' f = foldr ((:) . f) [])
```

This is called point-free style.
((:) . f) is a composition of list constructor (:) and argument-function f. Function composition (.) is defined as:

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

The last line can be also written as:

```
f . g = \x -> f (g x)
```

Filter can be represented as:

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc)
```

Note: If always have the else-clause in haskell since it's pure functional language (without side-effects).

Many list functions can be specified by a single foldr:

Example

$$
\begin{array}{lcl}
\mathit{sum} & = & \mathit{foldr}\ (+)\ 0 \\
\mathit{product} & = & \mathit{foldr}\ (*)\ 1 \\
\mathit{and} & = & \mathit{foldr}\ (\wedge)\ \mathit{True} \\
\mathit{or} & = & \mathit{foldr}\ (\vee)\ \mathit{False} \\
\mathit{maximum} & = & \mathit{foldr}\ \mathit{max}\ (-\infty) \\
\mathit{minimum} & = & \mathit{foldr}\ \mathit{min}\ \infty \\
\mathit{length} & = & \mathit{foldr}\ (\lambda\,\_\,n \mapsto 1 + n)\ 0 \\
\mathit{concat} & = & \mathit{foldr}\ (+\!\!\!+)\ []
\end{array}
$$

The Bird-Meertens Formalism, devised by Richard Bird and Lambert Meertens:

- The BMF (also called Squiggol) is a calculus for deriving programs from specifications (in a functional program setting).
- Proposes algebra of List-based data structures with map/fold/filter HOFs, composition and primitive operations.
- A big class of regular loop/for calculations in imperative langugles can be easily described by BMF.
- List-functions, calculated this way (homomorphisms), can be easily parallelized (Map/Reduce).

The great introduction to Lists, Monoids, Homomorphisms (and it's 3 theorems) is given in russian at Folds in Intel Click Plus article.

Selected Publications

Richard Bird

Lambert Meertens

Jeremy Gibbons

- R.S. Bird, An Introduction to the theory of lists
- Lambert Meertens. Algorithmics Towards programming as a mathematical activity
- Jeremy Gibbons. The Third Homomorphism Theorem

Safety

Consider the case - we need to describe a flock of sheep. Every sheep has a mother and father. In C++ we usually write this:

```
class Sheep {
public:
  std::string m_Name;
  Sheep       *m_pFather;
  Sheep       *m_pMother;

  Sheep(const std::string &name)
  : m_Name(name), m_pFather(NULL), m_pMother(NULL)
  {}
}
```

And if we need to get a name of the fater we write:

```
std::string fater_name = aSheep.pFater->name
```

And this is sometimes JUST CRASH with ACCESS VIOLATION since aSheep can be a "Dolly-clone" without mother and/or father.

Safety

"Null References" are also invented by C.A.R. Hoare.

Citation

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W).
My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.
This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references.
This is the solution, which I rejected in 1965.

Nulls at Haskell

In haskell there is no such notion as null. It uses Maybe algebraic data type (ADT) in order to represent calculation which may provide no any result:

Example

```
data Maybe a = Nothing | Just a
```

Here "Maybe" is a type-constructor function (it gives a type "a" as an input and returns a new type "Maybe a").
"Nothing" and "Just a" are value constructors.

Nulls at Haskell

Datatype for our sheep is:

Example

```haskell
data Sheep = Sheep {
  name    :: String
, mother :: Maybe Sheep
, father :: Maybe Sheep
  ..
}
-- this will auto-generate functions:
name :: Sheep -> String
name = ...
father :: Sheep -> Maybe Sheep
father = ...
mother :: Sheep -> Maybe Sheep
mother = ...
```

Nulls at Haskell

The following function will find mothernal grandfather:

Example

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

Let's try to forget about "Nothing" case:

Example

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

Nulls at Haskell

Let's compile the mentioned version:

Example

```
> ghc --make -Wall ... <file_name>.hs
...
Warning: Pattern match(es) are non-exhaustive
         In a case alternative:
           Patterns not matched: Nothing
...
```

We can see that the compiler warns us about incomplete case alternatives.
Note: this works only if we are interesting in warning (-Wall option).

Nulls at Haskell

But what if we need to know a more distant ancestor? The typical solution is to use a nested case operators (or if-then-else):

Example

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

How does Haskell deal with this issue?
The answer is - MONADS !!!

Do you see similarity?

Let's look on the following very similar functions:

Example

```haskell
-- father, mother :: Sheep -> Maybe Sheep
matGrandf :: Sheep -> Maybe Sheep
matGrandf s = case (mother s) of
                Nothing -> Nothing
                Just m  -> father m
matGrandm :: Sheep -> Maybe Sheep
matGrandm s = case (mother s) of
                Nothing -> Nothing
                Just m  -> mother m
patGrandm :: Sheep -> Maybe Sheep
patGrandm s = case (father s) of
                Nothing -> Nothing
                Just m  -> mother m
```

Unusual functional composition

They all are ALMOST the same except having **mother** and **father** at distinct places.
This means that we can ABSTRACT from the function specific (**mother** or **father** which are of the same type) and think about function composition in an UNUSUAL way.

Example (unusual functional composition)

```
-- father, mother :: Sheep -> Maybe Sheep
type SheepFun = Sheep -> Maybe Sheep
s_comp :: SheepFun -> SheepFun -> SheepFun
s_comp fun1 fun2 = \s ->
  case (fun1 s) of
    Nothing -> Nothing
    Just m  -> fun2 m
```

Great simplification

Using our unusual composition **s_comp** we can GREATLY simplify our functios:

Example (great simplification)

```
-- father, mother :: Sheep -> Maybe Sheep
matGrandf :: Sheep -> Maybe Sheep
matGrandf = mother 's_comp' father
matGrandm :: Sheep -> Maybe Sheep
matGrandm = mother 's_comp' mother
patGrandm :: Sheep -> Maybe Sheep
patGrandm = father 's_comp' mother
motPatGrand :: Sheep -> Maybe Sheep
motPatGrand = mother 's_comp' father 's_comp' father
```

Why is this unusual?

Why is this kind of composition is UNUSUAL?
Let's remember the usual function composition:

Example (usual functional composition)

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

In a usual way (using point in Haskell) we can combine two functions such that the input argument type (**domain**) of the first is equal to the output (**domain**) of the other (type-variable "b" at the example).
Returning to our SHEEPs, we need to compose functions, domen of the first one (Sheep) is NOT-EQUAL to the codomain of the second (Maybe Sheep).
That's why our sheep-function-composition is UNUSUAL.

Monadic functions

So, our sheep-functions (of type "Sheep − > Maybe Sheep") are called **monadic-functions**.
For 's_comp' Haskel uses special fish-operator (>=>).

Haskell syntax

In Haskell operator is a function named by spec-symbols only.
Regular function names are made of of alpha-numeric symbols only (first is alpha). Mixing spec-symbols and alpha-numberic symbols is prohibited.
Operators are used in infix form by default, but can be written in prefix form togeather with parenthesis:

```
plus x y = x + y
minus x y = (-) x y
```

Regular functions are in prefix form by default, but can be written in infix form togeather with quotes:

```
mean x y = (plus x y) / 2
mean x y = (x 'plus' y) / 2
```

Monadic fish-operator

Abstracting from "Maybe" (replacing it by "m") we can write the type of our fish operator (for our simple case) as:

```haskell
(>=>) :: Monad m => (a -> m a) -> (a -> m a) -> (a -> m a)
```

Remember, that in our case "a" stands for "Sheep", and "m" stands for "Maybe".
So, it gives two monadic functions, glues them togeather and returns a new monadic function (all are of the same type).
In Haskell in order to be a (**Monad**), fish-operator MUST be associative (for Maybe case):

```haskell
(f >=> g) >=> h  ==  f >=> (g >=> h)
```

Please, try to prove (or find in google) that fish is really associative for the case of "Maybe".

Monadic fish-operator

Proof hints

Glueing maybe–monadic-functions we always propagate "Nothing" case (no matter at what position).
If all the values are in a "Just x" form - we get a regular function composition, which is known to be associative.

Taking associativeness in account we are free to omit parenthesis:

Example

```
motPatGrand = mother >=> father >=> father
```

So, monadic fish-operator (as a first approximation) forms a semigroup at the set of monadic functions.

Algebra goes into play

From this moment we can use ALGEGRAIC methods while operating with monads.

Combining monadic with non-monadic function

But how can we combine a regular and monadic function ?
Suppose we have the following function for making a brother of the Sheep:

Example

```
brother :: Sheep -> Sheep
brother s = Sheep(
  name(s) ++ "_brother"
  mother(s)
  father(s)
)
```

It returns a plain **Sheep** (not a **Maybe Sheep**).
So, we need a kind of CONNECTOR function (to convert from plain **Sheep** to a **Maybe Sheep**).

Monadic return function

In Haskell a function named **return** is used for such convertion.

Example

```
return :: Sheep -> Maybe Sheep
return s = Just s
```

It gives a Sheep s and return a Just value-constructor for it.
Now we can combine our **brother** function with just defined **return** in order to get a monadic function:

Example

```
monadic_brother :: Sheep -> Maybe Sheep
monadic_brother = return . brother
```

We just used a regular composition (.) and Haskell point-free style here.

Monoid

Since return does nothing with its argument except placing it into Maybe monad, we can easily check the following statements:

Example

```
fun >=> return   ==   fun
return >=> fun   ==   fun
```

In other words, return - is the identity element for monadic fish-composition at the set of *simple* monadic functions.
So, our >=> and return form a Monoid (semigroup with identity element OR group without inversibility).

Algebra goes into play

We continue using ALGEGRAIC methods while operating with monads.

Correcting monadic composition

Let's remind our regular function composition:

Example (usual functional composition)

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

Our simplistic-version of monadic composition was:

```
(>=>) :: Monad m => (a -> m a) -> (a -> m a) -> (a -> m a)
```

Actually in Haskell, fish has more sophisticated type:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
```

There is also a flipped-version of fish-operator (similar to regular composition):

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
```

Correcting monadic composition

So, what is a regular function composition?
Let's call F_1 a set of all the one-argument functions:

$$F_1 = \{f : A \mapsto B\}$$

Then, composition can be treated as a partial function from cartesian product of F_1 with itself to F_1.

$$\circ : (F_1 \times F_1) \nrightarrow F_1$$

In the same way, monadic fish composition operator is also a partial function (not all the monadic functions can be combined with it, but only those that match type constraints).

Monad type class

Taking all the above into account, the question is - "What is a monad in Haskell"? Let's look at the source code:

Example

```
class  Monad m  where
    -- Sequentially compose two actions, passing any value
    -- produced by the first as an argument to the second.
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b
    -- Inject a value into the monadic type.
    return      :: a -> m a
    -- Sequentially compose two actions, discarding any
    -- value produced by the first, like sequencing operators
    -- (such as the semicolon) in imperative languages.
    (>>)        :: forall a b. m a -> m b -> m b
        -- Explicit for-alls so that we know what order to
        -- give type arguments when desugaring
    m >> k      = m >>= \_ -> k
    ...
```

Fish and Bind operators

Let's compare the monadic bind-operator and a fish-one:

Example

```haskell
-- Just to remind a Monad definition:
(>>=) :: forall a b. m a -> (a -> m b) -> m b
-- Left-to-right Kleisli composition of monads
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \x -> f x >>= g
```

So, fish-operator is a polymorphic higher-order function, defined for ALL the monads through the appropriate SPECIFIC bind operator.

Fish and Bind for Maybe

The Maybe monad in Haskell is defined as:

Example

```haskell
instance  Monad Maybe  where
-- (>>=) :: m a -> (a -> m b) -> m b
  (Just x) >>= k        = k x
  (Just _) >>  k        = k
  Nothing  >>= _        = Nothing

  return                = Just
```

Monad laws - nice version

As we told above, in order to be a monad, a programmer (not a compiler) MUST enforce the followin LAWS for specific monad. The nice version of monad laws uses fish-operator:

Example

```
1. return >=> f       ==  f
2. f >=> return       ==  f
3. (f >=> g) >=> h    ==  f >=> (g >=> h)
```

Compare with the composition LAWS:

Example

```
1. id . f          ==  f
2. f  . id         ==  f
3. (f . g) . h     ==  f . (g . h)
```

What is the difference ???

Monad laws - ugly version

The UGLY version of the same laws is using monadic bind operator:

Example

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

Ensuring the correctness of monad laws for standard/custom monads is CRITICAL for the correctness since Haskell compiler is free to evaluate fish/bind operators in ANY ORDER!!!

Monad folding

What if we need to compose a dynamic list of monadic functions?

Example

```
h = mf1 >=> mf2 >=> ... >=> mfN
```

In this case we need to use a folding function (foldr or foldl):

Example

```
h = foldr (>=>) return [mf1, mf2, ..., mfN]
```

Remember that:

- **return** is an identity element of our monoid
- fish-operator (>=>) - operation of monoid (semigroup)
- monadic functions (**mf1, mf2, ... mfN**) - elements of monoid

S.McLane

Monad is the Monoid at the category of endofunctors.

List comprehensions - Intro

It's time to remember our first example - quicksort function.

Example

```
qsort        :: Ord a => [a] -> [a]
qsort []     =  []
qsort (x:xs) =  qsort [y | y <- xs, y <  x] ++
                [x                        ] ++
                qsort [y | y <- xs, y >= x]
```

The expression in square brackets is a Haskell **list comprehension**.

Pythagorean triples

Let's find all the positive triples such that $x^2 + y^2 = z^2$.
C++0x version (n is the limit) is:

Example

```
void PythTriples(int limit,
    std::function<void(int,int,int)> yield) {
  for (int x = 1; x < limit; ++x)
    for (int y = x + 1; y < limit; ++y)
      for (int z = y + 1; z <= limit; ++z)
        if (x*x + y*y == z*z)
          yield(x, y, z);
}
int main() {
  PythTriples(20, [] (int x, int y, int z) {
    std::cout << x << "," << y << "," << z << "\n";
  });
}
```

Pythagorean triples in Haskell

Haskell version with list comprehensions is:

Example

```
pyth :: (Num a, Enum a) => a -> [(a, a, a)]
pyth n = [(x,y,z) | x <- [1..n], y <- [x..n], z <- [y..n],
          x^2 + y^2 == z^2]
```

Sample session (for n = 20):

Example

```
> pyth 20
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]
```

List comprehensions actually is a very conveniet syntax sugar over **List Monad**.

Pythagorean triples desugared

So, desugared version of pythagorean triples uses list monad:

Example

```
pyth n = do
  x <- [1..n]
  y <- [x..n]
  z <- [y..n]
  guard $ x^2 + y^2 == z^2
  return (x,y,z)
```

Do-notation is also just a syntactic sugar with the following rules

Definition

```
do { x }
  = x
do { x ; <stmts> }
  = x >> do { <stmts> }
do { v <- x ; <stmts> }
  = x >>= \v -> do { <stmts> }
```

Desugared List monad

Desugared version using monadic bind operator is:

Example

```
import Control.Monad
pyth n =
  [1..n] >>= (\x ->
  [x..n] >>= (\y ->
  [y..n] >>= (\z ->
  (guard $ x^2 + y^2 == z^2) >>
  return (x,y,z))))
```

Here guard - is a monadic higher-order function:

Example

```
guard :: (MonadPlus m) => Bool -> m ()
guard True  = return ()
guard False = mzero
```

MonadPlus

MonadPlus class extends Monad in order to support choice and failure:

Example

```
class Monad m => MonadPlus m where
   -- the identity of 'mplus'.
   -- It should also satisfy the equations
   --   mzero >>= f   =  mzero
   --   v >> mzero    =  mzero
   mzero :: m a
   -- an associative operation
   mplus :: m a -> m a -> m a

instance MonadPlus [] where
   mzero = []
   mplus = (++)

instance MonadPlus Maybe where
   mzero = Nothing
   Nothing `mplus` ys = ys
   xs      `mplus` _  = xs
```

List Monad

And, finally, list monad is defined as:

Example

```
instance Monad [] where
  return x   = [x]
  xs >>= f   = concat (map f xs)
```

Here return is an identity monadic function (of type a $\mapsto$ [a]).
It just puts an argument to the list (returns a one-elemen-list).

Function f (also of a type a $\mapsto$ [b]) is any monadic function.
It converts an argument to the list of some type (equal or not to the type of argument).

Bind maps **f** over monadic value (xs list) which leads to the list of lists, then - makes a plain version via concat function.

List Monad Bind example

In order to be more clear, consider example:

Example

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

> concat $ map (\x -> [-x,x]) [1,2,3]
[-1,1,-2,2,-3,3]

> [1,2,3] >>= \x -> [-x,x]
[-1,1,-2,2,-3,3]
```

Concat is defined as:

Example

```
-- Concatenate a list of lists.
concat :: [[a]] -> [a]
concat = foldr (++) []
```

Powerset definition

Consider the powerset - set of all subsets of the given set.
For simplicity let original set be all numbers from 0 to N-1.
For N = 4 the set will be 0,1,2,3 and its powerset:

Example

```
{ }
{ 3 }
{ 2 }
{ 2 3 }
{ 1 }
{ 1 3 }
{ 1 2 }
{ 1 2 3 }
{ 0 }
{ 0 3 }
{ 0 2 }
{ 0 2 3 }
{ 0 1 }
{ 0 1 3 }
{ 0 1 2 }
{ 0 1 2 3 }
```

Powerset in c++ - part 1

Another example - all subsets of the given set (powerset):

Example

```
void powerset(size_t s, size_t n, bool b[],
    std::function<void(std::vector<size_t>)> yield){
  for(size_t i = 0; i <= 1; ++i) {
    b[n] = i ? true : false;
    if (n < (s - 1))
      powerset(s, n + 1, b, yield);
    else {
      std::vector<size_t> v;
      for (size_t j = 0; j < s; ++j)
        if (b[j])
          v.push_back(j);
      yield(v);
    }
  }
}
```

Powerset in c++ - part 2

Calling the powerset (in CPS style using C++0x lambdas):

Example

```
int main(void) {
  size_t s = 4;
  boost::scoped_array<bool> pB(new bool [s]);
  powerset(s, 0, pB.get(), [] (std::vector<size_t> v) {
    std::cout << "{";
    std::for_each(v.begin(), v.end(), [] (size_t e) {
      std::cout << " " << e;
    });
    std::cout << " }" << std::endl;
  });
  return 0;
}
```

Powerset in Haskell

In Haskell the powerset is:

Example

```
powerset :: [a] -> [[a]]
powerset = filterM (\x -> [True, False])
-- powerset xs = filterM (\x -> [True, False]) xs
```

Here's a standard polymorphic monadic HOF filterM:

Example

```
-- This generalizes the list-based 'filter' function.
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
filterM _ []     = return []
filterM p (x:xs) = do
  flg <- p x
  ys  <- filterM p xs
  return (if flg then x:ys else ys)
```

Questions

Additional stuff:

- TeX sources at github