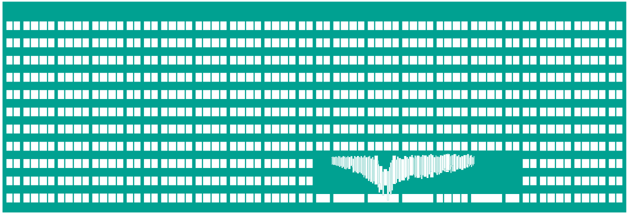
VŠB TECHNICKÁ UNIVERZITA OSTRAVA

VSB TECHNICAL UNIVERSITY OF OSTRAVA



www.vsb.cz

Advanced Functional Programming

behalek.cs.vsb.cz/wiki/Functional_Programming

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

December 5, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Functional programming

- Declarative style of programming
  - We define what needs to be computed, a run-time environment responsibility is how it will be evaluated.
  - Similar to math, we have various rules how to simplify an expression, but there are different ways how these rules can be applied for given expression.
- Programming with expressions (no statements)
  - Functional program is a set of function's definitions.
  - Functions are first class citizens - a function can return a function, high-order functions, partially evaluated functions.
  - Program's evaluation is the evaluation of some `main` expression.
- Immutable data structures - once created data can not be changed.
- No side effects (if possible)
  - Functions only return values, no changes other changes.
  - For the same parameters, we always get the same result (referential transparency).
  - Sometimes side effects can not be avoided (input - output operations) - programming with actions.

Functional programming vs OOP

- Today, probably the most popular programming style is Object Oriented Programming.
- Object Oriented Programming - objects and (most often) classes
  - Encapsulation - data are hidden inside and are accessible only trough given interface.
  - *Abstraction* - objects can be black boxes and we can use them even if do not know how they are working inside (works for most programming styles).
  - Composition, inheritance, and delegation - objects can be white boxes and new objects can be created with/based on existing objects.
  - Polymorphism - in OOP, it is usually refering to a situation, when calling code can be agnostic as to which class in the supported hierarchy it is operating on.
- Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts. (M. Feathers)

Functional programming in popular languages

- OK, but what if I do NOT want to use Haskell?
- Today's most popular programming languages are mostly multi-paradigm languages → they support various style of programming.
- What we really need for functional style of programming?
  - Functions - they are there, side effects are mostly optional.
    - Recursion - widely supported in all relevant languages.
    - What if we have a cycle inside in a function, is this a problem?
    - Functions as first class citizens - more of a problem, but most languages covers this.
  - Immutable data types - a choice of a programmer.
  - A strong type system to capture errors.
- Notable items on *a nice to have list*
  - Algebraic data types - rare, in OOP some solution can be inheritance.
  - Higher-kinded polymorphism - bigger issue, partially can be solved by generic data types.
- In C# we have: delegates, lambda expressions, pattern matching, tuples...

Immutable data types - Haskell

```haskell
module Stack (Stack (..), push, pop, isEmpty, empty) where

data Stack a = Stack [a] deriving Show

push :: a -> Stack a -> Stack a
push x (Stack xs) = Stack (x:xs)

pop :: Stack a -> (a,Stack a)
pop (Stack (x:xs)) = (a, Stack xs)

isEmpty (Stack []) = True
isEmpty _ = False

empty = Stack []
```

Mutable data types

```
public class Stack<T>
{
    private List<T> data;

    public Stack()
    {
        data = new List<T>();
    }

    public void Push(T item)
    {
        data.Add(item);
    }

    public T Pop()
    {
        T item = data[data.Count-1];
        data.RemoveAt(data.Count-1);
        return item;
    }

    static void Main(string[] args)
    {
        Stack<int> stack =
            new Stack<int>();
        stack.Push(1);
        stack.Push(2);
        var x = stack.Pop();
        Console.WriteLine(x);
    }
}
```

Immutable solution in C# (1)

```
public class NewStack<T>
{
    private T Data { get; init; }

    private NewStack<T> Next { get; init; }
        private NewStack()  { }

    static public NewStack<T> Empty() => null;
    static public bool IsEmpty(NewStack<T> stack) => stack == null;

    static public NewStack<T> Push(NewStack<T> stack, T item) =>
    new NewStack<T> { Data = item, Next = stack };

    static public (T Item, NewStack<T> Stack) Pop(NewStack<T> stack) =>
        (IsEmpty(stack)) ? throw new Exception("Empty stack.") : (stack.Data, stack.Next);
}
```

Immutable solution in C# (2)

```
static void Main(string[] args)
{
    NewStack<int> newStack = NewStack<int>.Empty();
    newStack = NewStack<int>.Push(newStack, 1);
    newStack = NewStack<int>.Push(newStack, 2);

    (x,newStack) = NewStack<int>.Pop(newStack);

    Console.WriteLine(x);
}
```

Immutable array

- Sometimes they are called persistent data structures.
- https://en.wikipedia.org/wiki/Persistent_data_structure
- https://en.wikipedia.org/wiki/Persistent_array

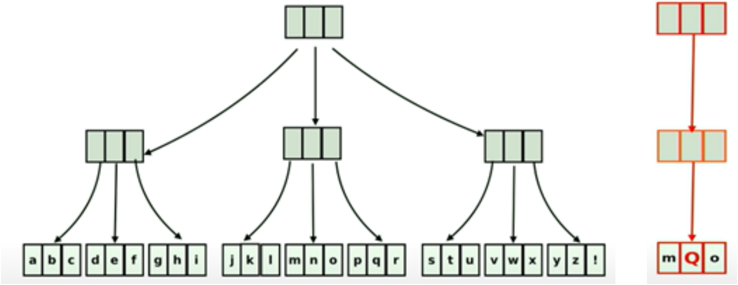


Figure: An idea how to implement immutable array.

Immutable data types

- Immutable data types
  - Studied problem, plenty of posibilities.
  - Common in API of many languages (C#: string, DateTime, `https://www.nuget.org/packages/System.Collections.Immutable/`).
- What if I really need mutable data structure?
  - For example *quick* implementation of `quicksort`?
  - No big deal, even Haskell has them.
  - `https://hackage.haskell.org/package/vector-0.12.3.1/docs/Data-Vector-Unboxed-Mutable.html`
  - `https://koerbitz.me/posts/Efficient-Quicksort-in-Haskell.html`

Functions with No Side Effects (1)

- What are side effects, how do i recognise them?

```
public double Add(double a, double b) {
    return a + b;
}
public double Add2(double a, double b) {
    try {
        Console.WriteLine($"a={a}, b={b}");
    } catch (Exception ex) { }
    return a + b;
}
public int Divide(int a, int b) {
    return a / b;
}
```

Functions with No Side Effects (2)

- How can I avoid them?

```
public int? Divide2(int a, int b) {
    if (b == 0)
        return null;
    return a / b;
}

public int Divide3(int a, NonZeroInteger b) {
    return a / b.Number;
}
```

```
public class NonZeroInteger {
    public int Number { get; }

    public NonZeroInteger(int number) {
        Number = number;
        if (number == 0)
            throw new ArgumentException();
    }
}
```

Functions with No Side Effects (3)

- What if they can not be avoided?
  - For example input - output operations?

    ```
    inputInt :: Int

    inputDiff = inputInt - inputInt

    funny :: Int-> Int
    funny n = inputInt + n
    ```
  - Library functions like: `Datetime.Now`
- Haskell uses **monads** to solve this issue.
  - *Think* from category theory → theoretical aspects are beyond the scope of this presentations.
  - Monad is a monoid in the category of endo-functors.
  - From the theory, there are some rules that a programmer should obey, but even Haskell can not enforce them.
  - Functor → Applicative functor → Monad
- Informally, monads are a sort of pure functional envelop for non-pure actions.
- Practically, its a set of design patterns solving plenty of situations that are frequently occurring in practice.
- For example in C#, these principles are used for LINQ.

Motivation (1)

- Complicated theory, but really it solves some practical issues.
- Lets start with data type `Maybe`

```
data Maybe a = Nothing | Just a

betterDiv :: Int -> Int -> Maybe Int
betterDiv x y | y==0 = Nothing
              | otherwise = Just (x `div` y)
```

- Now we want to compute some *expressions* where we use it like a value type.

```
data Expr = Num Int
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
```

Motivation (2)

- Now we need to compute such expression

```haskell
eval :: Expr -> Maybe Int
eval (Num x) = Just x
eval (Div x y) = case eval x of
                  Nothing -> Nothing
                  Just x' -> case eval y of
                      Nothing -> Nothing
                      Just y' -> betterDiv x' y'
eval (Add x y) = case eval x of
                  Nothing -> Nothing
                  Just x' -> case eval y of
                      Nothing -> Nothing
                      Just y' -> Just (x' + y')
```

- We can see emerging *patter*, how *actions* are *linked* one after the other.

Monads (1)

- New functions are produced like a **composition** of functions → important abstraction mechanism. `(.) :: (b -> c) -> (a -> b) -> a -> c`
- The ordering of functions does not matter, we can introduce:
  `(>.>) :: (a -> b) -> (b -> c) -> a -> c`
- We want to have something similar to that for our `Functor` class. How the functions from our examples looked liked?
  ```
  eval :: Expr -> Maybe Int
  compare :: Int -> Maybe Bool
  ```
- So, to be able to *compose* such functions, we need something like:
  `(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c`
- Consider, we have an operator >>= (bind): `(>>=) :: m a -> (a -> m b) -> m b`
- Then it is easy, operator >=> (Fish operator, Klesli category) can be defined as:
  ```
  f (>=>) g = \ a -> let mb = f a
                     in mb >>= g
  ```

Monads (2)

- OK, we have eliminated some unnecessary staff, but we still need: `(>>=) :: m a -> (a -> m b) -> m b`, right?
- That is precisely how monads are defined in Haskell.

```
class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b
  return :: a -> f a
```

- The final step will be defining monad for our type Maybe.

```
class Monad Maybe where
  Just x  >>= f = f x
  Nothing >>= f = Nothing

  return x =  Just x
```

Monads (3)

- Now, we can chain actions better.

```
*Main> (Just 1) >>= (\x-> return (x+1))
Just 2
*Main> (Just (+)) >>= (\y -> Just (y 1 2)) >>= (\x -> return (x+1))
Just 4
*Main> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
*Main> Just 3 >>= \x -> Just "!" >>= \y -> Just (show x ++ y)
Just "3!"
```

- We can even solve our original problem!

Monads (4)

- Solving *maybe* expressions with *monads*.

```
eval :: Expr -> Maybe Int
eval (Num x) = return x
eval (Div x y) = eval x >>= (\x' -> eval y >>= (\y' -> betterDiv x' y'))
eval (Add x y) = eval x >>= \x' -> eval y >>= \y' -> return ( x'+ y')
eval (Mul x y) = eval x >>=
                 \x' -> eval y >>=
                 \y' -> return ( x'* y')
eval (Sub x y) = do x' <- eval x
                    y' <- eval y
                    return ( x'- y')
```

Monads (5)

- What are restrictions placed on Monads?
- What type of a type (it is called *kind* in Haskell) is `Maybe`

```
 *Main> :kind Int
Int :: *
*Main> :kind Maybe
Maybe :: * -> *
```

- If we check the kind of `Monad` you get: `(* -> *) -> Constraint`.
- `Monad` definition contains `Applicative`

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b -- $ :: (a -> b) -> a -> b
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Monads (6)

- Now, we can add type `Maybe` into these type classes.

```haskell
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap _ Nothing = Nothing
instance Applicative Maybe where
    pure x = Just x
    (Just f) <*> (Just x) = Just (f x)
    _ <*> _ = Nothing
```

- What we get for chaining actions?

```
*Main> (+1) `fmap` ((*2) `fmap` ((+3) `fmap` (Just 1)))
Just 9
*Main> (+) <$> (Just 1) <*> (Just 2)
Just 3
```

Monads (7)

- If we have `Monad`, we also have `Functor` and `Applicative`.

```
fmap fab ma   =  ma >>= (\x -> return (fab x)) -- (return.fab)
pure a        =  return a
mfab <*> ma   =  mfab >>= (\ fab -> ma >>= (return . fab))
```

List Monad (1)

- Nice *example* of a monad is the list. Informally, required operations are implemented:

```
myFmap :: (a -> b)    -> [a] -> [b]
myFmap = map

myApply :: [a -> b]    -> [a] -> [b]
myApply fs xs = [f x | f <- fs, x <- xs]

myBind :: [a] -> (a -> [b]) -> [b]
myBind xs f = concat (map f xs)
```

- Now, we can observe, what we *can do* with such defined operators.

```
*Main> (+1) <$> [1,2,3]
[2,3,4]
*Main> (+) <$> [1,2,3] <*> [1,2,3]
[2,3,4,3,4,5,4,5,6]
*Main> [1,2] >>= \n -> ['a','b']
             >>= \ch -> [(n,ch)]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
*Main> [3,4,5] >>= (return . (+1))
               >>= (return . (*2))
[8,10,12]
```

List Monad (2)

- Consider following variants for a function finding Pythagoras triplets.

```
pythagoreanTriples :: Integer -> [(Integer, Integer, Integer)]
pythagoreanTriples n =
  [1 .. n] >>= (\x ->
  [x+1 .. n] >>= (\y ->
  [y+1 .. n] >>= (\z ->
  if x^2 + y^2 == z^2 then return (x,y,z) else [])))

pythagoreanTriples' :: Integer -> [(Integer, Integer, Integer)]
pythagoreanTriples' n = do x <- [1 .. n]
                           y <- [x+1 .. n]
                           z <- [y+1 .. n]
                           if x^2 + y^2 == z^2 then return (x,y,z) else []

pythagoreanTriples'' :: Integer -> [(Integer, Integer, Integer)]
pythagoreanTriples'' n =
  [(x,y,z) | x <- [1 .. n], y <- [x+1 .. n], z <- [y+1 .. n], x^2 + y^2 == z^2]
```

IO Monad (1)

- This part is for programmers, that do not care about a theory.
- There is a special type `()` with only value `()` called *unit* type - representing a sort of dummy value.
- All input and output *actions* can be recognized by having `IO` in their type definition.
  - Input: `getLine :: IO String`
  - Output: `putStr :: String -> IO ()`
  - Usually, when we are talking about monads, we say, that they represents some sort of *containers* $\rightarrow$ better intuition for `IO` is: `bake :: Recipe Cake`.
- You can *glue* these actions by syntax construct: `do`.
- How to get value from/to `IO`?
  - There is a syntactic construct in `do` (called bind): `x <- action`, where if `action :: IO a`, then the type of variable x is a.
  - There is a function `return :: a -> IO a`, it can be used to *put* a common value into `IO`.
- Finally, the function `main` has a type: `main :: IO a`
- And that is all, Is it clear?

IO Monad (2)

- Simple example:

```haskell
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    let bigName = map toUpper name
    putStrLn ("Hey " ++ bigName ++ ", you rock!")
```

- Now, we can compile it and execute.

```
PS C:\> ghc .\test.hs
[1 of 1] Compiling Main             ( test.hs, test.o )
Linking test.exe ...
PS C:\> .\test.exe
Hello, what's your name?
Marek
Hey MAREK, you rock!
```

IO Monad (3)

- The construct do is just an expression, we can use it in the same way...

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            print $ reverseWords line
            main
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

- You should notice, that return does not end the function like in *common* languages.

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

From IO Monad to State?

- In previous part, we have introduced a mechanism how *actions* can be chained → nicer way how to write it.
- But we have started with the idea, that impure actions (manipulating with *state*) will be solved with monads.
- Our example was IO Monad that solves input - output operations.

```
-- inputLine :: String
getLine :: IO String
putStr :: String -> IO ()

do x <- getLine
   putStr x -- y <- putStr x, y == ()

ready :: IO Bool
ready = do c <- getChar
           return (c == 'y')
```

- Nice example what getLine `:: IO String` is: bake `:: Recipe Cake`.

State Monad (1)

- How does it work? The idea is captured in more general monad that captures state.
- Lets first focuse on the idea → state manipulation can be captured like a function taking original state and producing a pair (some value, new state).

```
type SimpleState s a = s -> (s, a)

retSt :: a -> SimpleState s a
--retSt a s = (s,a)
retSt a = \s -> (s,a)
```

- Now, lets create a simple *input* containing a list of integers (our state is just this list).

```
type ListInput a = SimpleState [Int] a

readInt :: ListInput Int
readInt stateList = (tail stateList, head stateList)
```

State Monad (2)

- Finally, lets try to make a function chaining actions (like >>=).

```
bind :: (s -> (s,a))              -- SimleState s a
     -> (a -> (s -> (s, b))) -- a -> SimpleState s b
     -> s -> (s, b)           -- SimpleState s b
bind step makeStep oldState =
    let (newState, result) = step oldState
    in  (makeStep result) newState
```

- Finally, we can bind actions as with monads.

```
*Main> (readInt `bind` \a->readInt `bind` (\b->retSt (a+b))) [1,2,3]
([3],3)
```

- In our example, we have created a function defining what to do with the input. When it is executed it *bakes* the result. If provided the same *ingredients*, it *bakes* the same result.

State Monad (3)

- What if we want to realy make it a part of Monad type class (it will not work for type synonym)?

```
newtype State s a = State { runState :: s -> (s, a) }

readInt' :: State [Int] Int
readInt' = State {runState = \s->(tail s, head s)}

instance Functor (State s) where
    fmap f m = State $ \s-> let (s',a) = runState m s in (s',f a)

instance Applicative (State s) where
    pure a = State (\s->(s,a))
    f <*> m = State $ \s-> let  (s',f') = runState f s
                                (s'',a) = runState m s' in (s'',f' a)
instance Monad (State s ) where
    return a = State (\s->(s,a))
    m >>= k = State $ \s -> let (s',a) =  runState m s  in runState (k a) s'
```

State Monad (4)

- We can even use do syntax now.

```
add :: State [Int] Int
add = do x<-readInt'
         y<-readInt'
         return (x+y)
```

- Examples, how to use this state monad:

```
*Main> runState (readInt' >>= \a->readInt' >>= (\b->return (a+b))) [1,2,3]
([3],3)
*Main> runState add [1,2,3]
([3],3)
```

- Finally, assuming we have RealWorld, we ca define type IO as:

```
type IO a = State RealWorld a
--getChar :: RealWorld -> (RealWorld, Char)
--main :: RealWorld -> (RealWorld, ())
```

Monads in C#(1)

- Can we implement the same ideas in C#?
- Lets start with something simple, function composition.

```
public static Func<A, C> After<A, B, C>(this Func<B, C> f, Func<A, B> g)
        => value => f(g(value));

public static Func<A,C> Composition<A, B, C>(Func<B, C> f, Func<A, B> g)
        => value => f(g(value));

Func<string, int> parse = int.Parse; // string -> int
Func<int, int> abs = Math.Abs; // int -> int

Func<string, int> composition1 = abs.After(parse);
Func<string, int> composition2 = Composition(abs, parse);
```

Monads in C#(2)

- What if we want to have Maybe monad (there are various possible solutions).

```
public abstract class Maybe<A> {}

public class Just<T> : Maybe<T> { public T Value { get; init; } }
public class Nothing<T> : Maybe<T> {}

public static Maybe<A> Return<A>(this A value) => new Just<A> { Value = value };

public static Maybe<B> Bind<A, B>(this Maybe<A> x, Func<A, Maybe<B>> f) => x switch
  {
    Nothing<A> => new Nothing<B>(),
    Just<A> value => f(value.Value),
    _ => throw new Exception("Unexpected value.")
  };
```

Monads in C#(3)

- Now, we can chain actions as before in Haskell.

```
var result2 = new Just<int>() { Value = 1 }
                .Bind(x => new Just<int> { Value = x + 1 })
                .Bind(x => new Just<string>() { Value = "Value: " + x });
```

- Even more, C# have something called query syntax.
  - It is related to LINQ, it uses a syntax similar to SQL, but it is also convenient when we threat `IEnumerable` as a monad.
  - It requires to define: `Select`, `SelectMany`

```
public Maybe<B> SelectMany<B>(Func<A, Maybe<B>> f) => (Maybe<B>)this.Bind(f);

public Maybe<C> SelectMany<B, C>(Func<A, Maybe<B>> f, Func<A, B, C> resultSelector)
  => (Maybe<C>)this.Bind(x => f(x).Bind(y => Return(resultSelector(x, y))));

var test =  from x in new Just<int> { Value = 1 }
            from y in new Just<int> { Value = 2 }
            select x + y;
```

Monads in C#(4)

- What about *State monad*, is it possible to define them in C#?

```
public delegate (TState State, T Value) State<TState, T>(TState state);

public static State<TState, C> SelectMany<TState, A, B, C>(
  this State<TState, A> source,
  Func<A, State<TState, B>> selector,
  Func<A, B, C> resultSelector) =>
    oldState => {
      (TState State, A Value) value = source(oldState);
      (TState State, B Value) result = selector(value.Value)(value.State);
      return (result.State, resultSelector(value.Value, result.Value));
    };
```

Monads in C#(5)

- Now, the usage in fact compose from two parts, first we are creating the function then we are executing it with chosen state (list of numbers in our case).

```
(List<int>, int Max) FindMax(List<int> list) =>
  (list.Where(x => x != list.Max()).ToList(), list.Max());
(List<int>, int Min) FindMin(List<int> list) =>
  (list.Where(x => x != list.Min()).ToList(), list.Min());

State<List<int>, string> query =
  from max in (State<List<int>,int>)(oldState => FindMax(oldState))
  from min in (State<List<int>, int>)(FindMin)
  from count in (State<List<int>, int>)(oldState => (oldState, oldState.Count))
  select $"Max {max}, Min {min}, beside {count} elements.";

  var (_, Value) = query(new List<int>{ 7,1,2,3,5});
```

- The result in `Value` will be:

```
Max 7, Min 1, beside 3 elements.
```

Advantages of functional style programming I

- In current popular programming languages, usage of functional programming style depends on programmer.
  - Today's most popular programming languages support multiple programming paradigms.
  - Functional style of programming can be easily applied in most of them.
  - Moreover, we can use even some fundamental functional concepts like monads.
  - And if we need mutable data or side-effect → no big deal, even Haskell have them.
- Big question that needs to be addressed is: **What will be the gain, if i use functional style of programming?**
- (Personal opinion) Functional programs are often shorter and more concise → easier to comprehend → easier to maintain.
  - Recursion is simpler, though not necessarily easier to learn.
  - Function signatures are more meaningful.
- ~~No~~ Fewer side effects (immutable data, pure functions)
  - Easier for concurrent execution.

Advantages of functional style programming II

- Much simpler testing → possible are even concepts like proving programs properties.
- More error prone → Haskell's type system captures a lot of errors → huge difference in run-time errors.

- New features like: lazy evaluation, infinite structures,

- *Guidelines to the usage of functional programming*
  - Like other style of programming, it does not solve all problems.
  - Like in other areas, benefits should overweight the costs.
  - We get mentioned benefits, even if just a part of the solution uses functional style of programming.

Software Verification and Validation - what it is about? I

- Software engineering is the systematic application of engineering approaches to the development of software.
  - **Verification** $\rightarrow$ Are we building the product right? $\rightarrow$ The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
  - **Validation** $\rightarrow$ Are we building the right product? $\rightarrow$ The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.
- We have plenty of of strategies and methodologies to software development $\rightarrow$ determines how and when *validation and verification* are conducted.
- The most common strategy how it conducted is some sort of testing (`https://en.wikipedia.org/wiki/V-Model_(software_development)`).
- Probably, the most basic form of testing are *unit tests*.
  - In the V-Model, unit tests eliminates bugs at code level or unit level (module, class,..). It verifies that an isolated unit is working correctly.

Unit Tests

- Unit tests are written by a programmer that have created the *unit*.
- Most common units in OOP are classes.
- There are plenty of tool helping with unit test.
  - Most basic toolkit usually represents *XUnit*: JUnit-Java, NUnit-C#, HUnit-Haskell...
  - Such tool is in fact a library allowing the test definition and containing some useful infrastructure.
  - Units testing is integrated for example in Visual Studio (helps with test creation, environment to execute and maintain tests).

```
[TestClass()]
public class StackTests
{
    [TestMethod()]
    public void PopTest()
    {
        Stack<int> s = new Stack<int>();
        s.Push(1);
        Assert.AreEqual(
            s.Pop(),
            1,
            "Value in stack should be 1.");
    }
    [TestMethod()]
    public void PushTest()  {  }
}
```

Unit Tests - difficulties

- What it takes to write a good test? Was our previous example OK? → Write a good test is not an easy task.
- Moreover, what if the tested function uses database or some device? What if we have a complex application relaying on some third party components? → *test fixtures*
- How meaningful is then the function's type definition?
  - If we have no side effects → all we need is to prepare the input → all changes are encapsulated in the result
- What if the function have some side effects?
  - What we really need to test?
  - How do we even prepare the test? How do we prepare *some state of the system*?
  - How do we check if the result fulfils the requirements?
- Even in pure functional languages, there are side effects, but the are bounded (monads in Haskell).

Reasoning about programs

- OK, functional languages have mathematical background, but is this any good for me (I am a programmer, not mathematician;-)?
- Formal definition of language semantic allows to prove program's properties → more trustworthy then just some *tests*.
  - Emended systems, automotive, ...
  - Tools: Formal proof management system Coq `https://coq.inria.fr/` → based on richly-typed functional programming language *Gallina*
    - CompCert - verification of C programs
    - Extract certified programs to Haskell
- Mathematical induction (informally)
  - Prove for $\mathbf{n} = \mathbf{0}$ (base case)
  - On assumption that it holds for $\mathbf{n}$, prove that it holds for $\mathbf{n+1}$
- Principle of structural induction for lists – we want to prove property **P**
  - Base case – prove **P** for [] outright.
  - Prove **P** for `(x:xs)` on assumption that **P** holds for `xs`.

Reasoning about programs - Example (1)

- We want to prove: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
- We start with *equations* from the source code.

```
[] ++ ys     = ys                -- ++.1
(x:xs) ++ ys = x: (xs ++ ys)     -- ++.2
```

- Now we can start proving (using mathematical induction).

```
-- a) [] => xs
([] ++ ys) ++ zs
= ys ++ zs            -- ++.1
= [] ++ (ys ++ zs) -- ++.1
-- b) (x:xs) => xs
((x:xs)++ys)++zs
= x:(xs++ys)++zs     -- ++.2
= x:((xs++ys)++zs)  -- ++.2
= x:(xs++(ys++zs))  -- assumption
= (x:xs)++(ys++zs)  -- ++.2
```

Reasoning about programs - Example (2)

- Better example: `(length (xs++ys) = length xs + length ys`
- We start with *equations* from the source code.

```
length []     = 0              --len.1
length (_:xs) = 1 + length xs  --len.2
```

- Now we can start proving (using mathematical induction).

```
-- a) [] => xs
length ([] ++ ys)
= length ys                -- ++.1
= 0 + length ys            -- + zero element
= length [] + length ys -- len.1
-- b) (x:xs) => xs
length ((x:xs) ++ ys)
= length (x:(xs++ys)         -- ++.2
= 1 + length (xs++ys)        -- len.2
= 1 + (length xs + length ys)  -- assumption)
= (1 + length xs) + length ys  -- associativity of +
= length (x:xs) + length ys    -- len.2
```

Lambda calculus I

- $\lambda - calculus$ is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution *(wiki)*.
- It was invented in 1930s by Alonzo Church.
- Universal model of computation, *as good as* Turing machine $\rightarrow$ all that can be compute by Turing machine can be expressed in $\lambda - calculus$ $\rightarrow$ roughly, this corresponds to problems that can be solved by a computer.
- Omitting many details, theoretical background for all functional programming languages.
  - Originally $\lambda - calculus$ is *untyped* $\rightarrow$ in programming we need types $\rightarrow$ not that easy to add them.

Lambda calculus - simplified definition

- Syntax (how it is written) - a lambda term is:
  - $x, y, z...$ - variables, representing a parameter or mathematical/logical value.
  - $(\lambda x.M)$ - abstraction, M is a lambda term, the variable x becomes bound in the expression.
  - (MN) - application, applying a function to an argument. M and N are lambda terms.
- Semantics (how to compute it)
  - $\alpha - conversion : (\lambda x.M[x]) \rightarrow (\lambda y.M[y])$ - renaming the bound variables in the expression. Used to avoid name collisions.
  - $\beta - reduction : ((\lambda x.M)E) \rightarrow (M[x := E])$ -replacing the bound variables with the argument expression in the body of the abstraction *(this really moves forward the computation)*.
  - $\eta - reduction : ((\lambda x.fx) \rightarrow f$ - expresses the idea of extensionality (two functions are the same if and only if they give the same result for all arguments).

Lambda calculus - normal form

- Redex - **Red**ucible **Ex**pression - expression that can be reduced with defined rules.
  - $\alpha - redex, \beta - redex$
- Church–Rosser theorem - when applying reduction rules to terms, the ordering in which the reductions are chosen does not make a difference to the eventual result.
- In other words, if there are two distinct reductions or sequences of reductions that can be applied to the same term, then there exists a term that is reachable from both results.
- **Normal form** - expression that contains no $\beta - redex$.
  - `42, (2, "hello"), \x -> (x + 1)`
- Haskell uses **weak head normal form** - stops when *head* is a lambda abstraction or a data constructor.
  - `(1 + 1, 2 + 2), \x -> 2 + 2,'h' : ("e" ++ "llo").`
- The question that remains is, how do we get the weak head normal form?

Lazy evaluation - what are our option for evaluation strategies?

- When choosing an evaluation strategy for expressions in languages like Haskell, what are key factors?
  - Evaluation order - which reductions are performed first (inner-most, outer-most)
  - How do we pass parameters to a function - by *value*, by name, by reference, by need...
- Function f is strict when and only when: $f\perp = \perp$
- *Strict evaluation* - function's arguments are evaluated completely before the function is applied.
  - innermost reduction, eager evaluation or greedy evaluation
  - Sometime also *Call by value* - it requires strict evaluation, arguments are passed as evaluated values.
  - It is used by most programming languages: Java, C#, F#, OCalm, Scheme...
- Non-strict evaluation - a function may return a result before all of its arguments are fully evaluated.
  - outer-most reduction, normal order evaluation (does not evaluate any of the arguments until they are needed in the body of the function).

Lazy evaluation (1)

- Lazy evaluation - When we are lazy enough, to call our evaluation lazy?
  - Sub-expressions will be evaluated only when they are needed for in evaluation.
  - If they are evaluated, they are evaluated only once.
- In pure functional languages, if we use outer-most reduction, we are doing normal order evaluation $\rightarrow$ only needed sub-expressions are evaluated, only needed arguments are evaluated.
- In pure functional languages, to be lazy enough, all we need is some clever way, how to pass arguments $\rightarrow$ **call by need**.
  - Used in Haskell, option in OCalm, Scheme, some languages simulate lazy behaviour for some sub-systems.
- In pure functional languages, the terms lazy evaluation, call by need, or non-strict evaluation mean the same *thing*.

Lazy evaluation (2)

- Eager evaluation

```
square(1+2)
square(3)
3*3
9
```

- Lazy evaluation

```
square(1+2)
let x = 1+2 in x*x
let x = 3 in x*x
3*3
9
```

Advantages of Lazy evaluation

- If an expression has a normal form, it will be reached by lazy evaluation strategy (theory nonsense:-).
- It allows to use new concepts, like infinite structures or functions → new way how to solve a problem (i still wont use it:-).

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- It is useful when processing (large) data (LINQ, Apache Spark,..)
  - Consider following example:

    ```
    map (\x->x^4) (concat (map (\x->[1..x]) [1..10]))
    ```

  - Will be the intermediate results constructed?
  - In fact, we are continually getting items from the final list!
  - How the equivalent in C++ will look like?
    - We need to sacrifice code clarity, or all intermediate results will be computed before we get some result.

Thank you for your attention

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

December 5, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE