Concepts of Higher Programming Languages
A Taste of Functional Programming and Haskell

Jonathan Thaler

Department of Computer Science

FH Vorarlberg
University of Applied Sciences

Functional Programming

What exactly is Functional Programming?

What is Functional Programming?

- A **style** of programming with the basic method of computation is **application of functions** to arguments.
- Functions are **first-class citizens**
  - Can be **assigned** to variables
  - Can be **passed as arguments** to other functions
  - Can be **constructed as return values** from functions.
- **Declarative** style of programming: describing **what** to compute instead of how.
- A functional language **supports** and **encourages** the functional style.
- Built on foundations of the **Lambda Calculus**.

Lambda Calculus

Is a notation for **expressing computation** based on the concepts of:

1. Function abstraction
2. Function application
3. Variable binding
4. Variable substitution

Calculus

A calculus is a **notation** that can be manipulated **mechanically** to achieve some end.

Function Abstraction

Function Abstraction

Function abstraction allows to **define functions** in the Lambda Calculus. The λ symbol denotes an expression of a function which takes exactly **one argument** which is used in the body-expression of the function to **calculate** something which is then the **result**.

Examples

$f(x) = x^2 - 3x + a$
`Lambda Expression:` $\lambda x.x^2 - 3x + a$

$f(x, y) = x^2 + y^2$
`Lambda Expression:` $\lambda x.\lambda y.x^2 + y^2$

Function Application

Function Application

To get the **result** of a function one **applies arguments** to it.

Examples

```
x = 3, y = 4
```

$f(x, y) = x^2 + y^2$
$f(3, 4) = 25$

$\lambda x.\lambda y.x^2 + y^2$
$((\lambda x.\lambda y.x^2 + y^2)\ 3)\ 4$
$= 25$

Variable Substitution

Variable Substitution

To compute the result of a Lambda expression (**evaluating** the expression) it is necessary to **subsitute** the bound variables by the argument to the function. This process is called β-reduction.

β-reduction

$((\lambda x.\lambda y.x^2 + y^2)\ 3)\ 4$

{`substitute x with 4`}

$(\lambda y.4^2 + y^2)\ 3$

{`substitute y with 3`}

$(4^2 + 3^2)$

$= 25$

Variable Substitution

β-reduction

$((\lambda x.\lambda y.x^2 + y^2)3)y$

$\{$`substitute x with y?`$\}$

$(\lambda y.y^2 + y^2)3$

$= 3^2 + 3^2$

$= 18$

No, we need to perform α-conversion!

Variable Binding

Definition

In the Lambda expression $\lambda x.x^2 - 3x + a$ the variable x is **bound** in the body of the function whereas a is said to be **free**.

A **bound** variable can be **renamed** within its scope without changing the meaning of the expression: $\lambda y.y^2 - 3y + a$ has the same meaning as the expression $\lambda x.x^2 - 3x + a$.

A **free** variable **must not be renamed** because it would change the meaning of the expression.

This process is called α-conversion and it becomes sometimes necessary to avoid **name-conflicts** in variable substitution.

Variable Substitution

β-reduction with α-conversion

$((\lambda x.\lambda y.x^2 + y^2)\ 3)\ y$

$\{\alpha$`-conversion: rename bound y to z`$\}$

$((\lambda x.\lambda z.x^2 + z^2)\ 3)\ y$

$\{$`subsitute x with free y`$\}$

$(\lambda z.y^2 + z^2)\ 3$

$\{$`subsitute z with 3`$\}$

$(y^2 + 3^2)\ 3$

$= y^2 + 9$

The result is actually a **new** function!

Function Examples

Identity Function $(\lambda x.x)$

$(\lambda x.x)\ 42 = 42$

Constant Function $(\lambda x.y)$

$(\lambda x.y)\ 42 = y$

? Function $(\lambda x.xx)$

$(\lambda x.xx)\ 42 = 42\ 42$

$(\lambda x.xx)(\lambda x.xx) = (\lambda x.xx)(\lambda x.xx)$

Summing the integers 1 to 10 in Java

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

The computation method is destructive **variable assignment**.

Declarative Programming

Summing the integers 1 to 10 in Functional Programming

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

The computation method is **function application**.

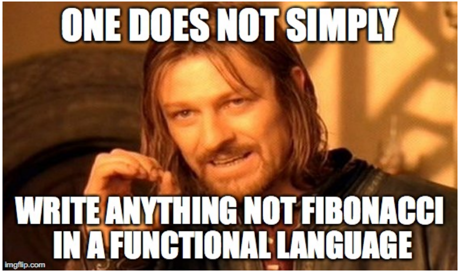
A Taste of Haskell

A Taste of Haskell

Higher-Order Functions
Property-Based Testing
Free Monads
Concurrency
Foldable and Monoids
Type Classes
Explicit Recursion
Static Types
Data Definitions
Pattern Matching
Functors
Basic Functions
Monads
Dependent Types
Lazy Evaluation
Immutable Data
Applicatives
Currying
Lambdas
STM
IO

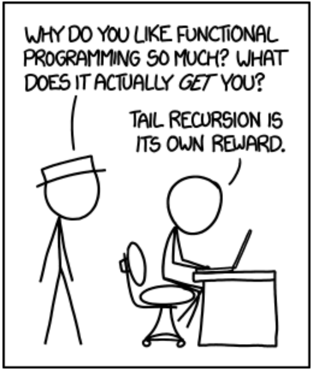
Key Learning:

Static Types, Immutable Data, Pure Functions



Key Learning:

Currying and Lambdas



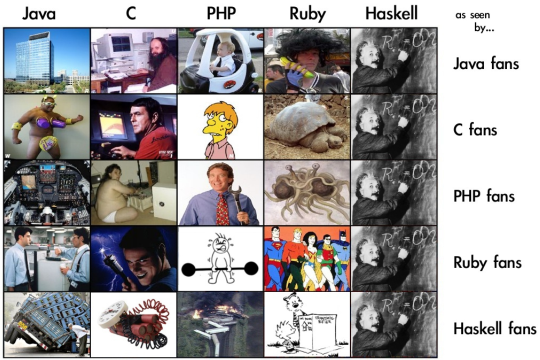
Key Learning:

Explicit Recursion

Key Learning:

Higher-Order Functions

A Taste of Haskell



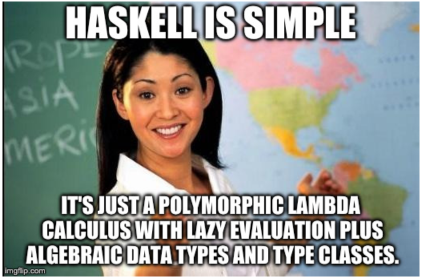
Key Learning:

Data Definitions and Pattern Matching

A Taste of Haskell

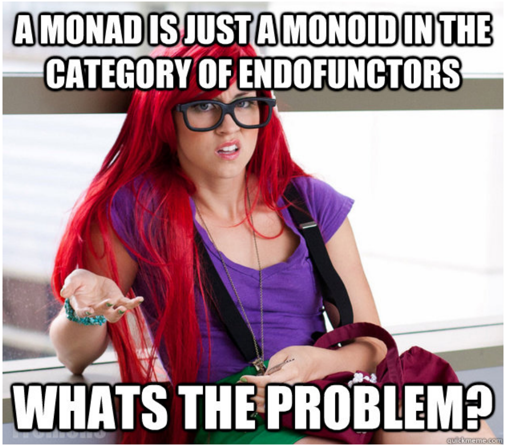
Key Learning:

Lazy Evaluation



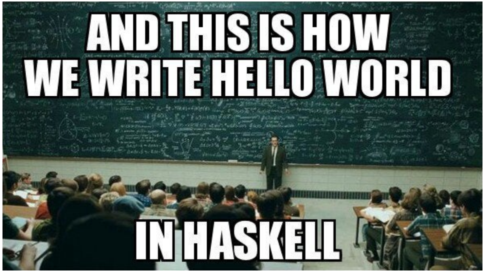
Key Learning:

Polymorphism through Type Classes



Key Learning:

Explicit in Side Effects



Key Learning:

Pure and Impure Side Effects (Monads)

A Taste of Haskell

Key Learning:

Concurrency and STM

A Taste of Haskell

Key Learning:

Property-Based Testing

Key Learning:

Data as Code (Free Monads)

A Taste of Haskell

Me, looking at a list of advantages of dynamic typing.

Key Learning:

Dependent Types

A Taste of Haskell

Factorial in Haskell

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- Declarative
- Immutable Data
- Recursion
- Static Types
- Explicit Input and Output
- Referential Transparency

A Taste of Haskell

Side-Effects in Haskell

```haskell
queryUser :: String -> IO Bool
queryUser username = do
  -- print text to console
  putStr "Type in user-name: "
  -- wait for user-input
  str <- getLine
  -- check if match
  if str == username
    then do
      putStrLn "Welcome!"
      return True
    else do
      putStrLn "Wrong user-name!"
      return False
```

- IO Type
- Side-Effects
- Sequencing
- History Sensitive
- "Functional C"

A Taste of Haskell

What is this code doing?

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