Haskell
Functional Programming

http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Gustave Eiffel

November 8, 2022
Programming paradigms

- Declarative programming
  - Procedural (C)
  - Object-Oriented (C++, Java)

- Imperative Programming
  - Functional (Haskell)
  - Logic (Prolog)
Programming paradigms are not distinct
Functional languages

- Scala
- f
t
- elixir
- cljs
- F#
- PureScript
- BS
- OCaml
- R


Everybody’s talking about functional programming

Lisp
Lisp (historically, LISP) is a family of computer programming languages with a long history and a distinctive, fully parenthesized prefix notation. Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today. (Only Fortran is older, by one year.)
Everybody’s talking about functional programming

Erlang

Erlang (https://www.erlang.org/) is a general-purpose, concurrent, functional programming language, as well as a garbage-collected runtime system.
Everybody’s talking about functional programming

**Elixir**
Elixir (https://elixir-lang.org/) is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM).
Everybody’s talking about functional programming

**F#**

F# ([http://fsharp.org/](http://fsharp.org/)) is a strongly typed, multi-paradigm programming language that encompasses functional, imperative, and object-oriented programming methods. It is being developed at Microsoft Developer Division and is being distributed as a fully supported language in the .NET framework.
Everybody’s talking about functional programming

**Ocaml**

Ocaml ([http://ocaml.org/](http://ocaml.org/)) originally named Objective Caml, is the main implementation of the programming language Caml. OCaml’s toolset includes an interactive top-level interpreter, a bytecode compiler, a reversible debugger, a package manager (OPAM), and an optimizing native code compiler.
Everybody’s talking about functional programming

Clojure
Clojure (https://clojure.org/) is a dialect of the Lisp programming language. Clojure is a general-purpose programming language with an emphasis on functional programming. It runs on the Java virtual machine and the Common Language Runtime.
Racket
Racket (http://racket-lang.org/), formerly PLT Scheme, is a general purpose, multi-paradigm programming language in the Lisp-Scheme family. One of its design goals is to serve as a platform for language creation, design, and implementation.
Everybody’s talking about functional programming

Elm
Elm (http://elm-lang.org/) is a domain-specific programming language for declaratively creating web browser-based graphical user interfaces. Elm is purely functional, and is developed with emphasis on usability, performance, and robustness.
Everybody’s talking about functional programming

Scala

Scala (https://www.scala-lang.org/) is a general-purpose programming language providing support for functional programming and a strong static type system. Designed to be concise, many of Scala’s design decisions aimed to address criticisms of Java.
Everybody’s talking about functional programming

Haskell
Haskell (https://www.haskell.org/) is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing. The latest standard of Haskell is Haskell 2010. As of May 2016, a group is working on the next version, Haskell 2020.
Characteristics of functional programming

- First-class functions
- Higher-order functions
- Lazy Evaluation
- Immutable data
- Manipulation of lists
- Pure functions
- Recursion
Functional programming

• Programming with pure functions.
• The output of a function is solely determined by the input (much like mathematical functions).
• No side-effects.
• No assignments.
• Functions compose.
• Expression-oriented programming.
Why FP matters?

1. FP offers concurrency/parallelism with tears.
2. FP has succinct, concise and understandable syntax.
3. FP offers a different programming perspective.
4. FP is becoming more accessible.

FP is fun!
Functions everywhere
Design patterns

<table>
<thead>
<tr>
<th><strong>OO patterns</strong></th>
<th><strong>Functional programming patterns</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Single responsability</td>
<td>Functions</td>
</tr>
<tr>
<td>Open / Closed</td>
<td>Functions</td>
</tr>
<tr>
<td>Interface segregation</td>
<td>Functions</td>
</tr>
<tr>
<td>Factory</td>
<td>Functions</td>
</tr>
<tr>
<td>Strategy</td>
<td>Functions</td>
</tr>
<tr>
<td>Decoration</td>
<td>Functions again</td>
</tr>
<tr>
<td>Visitor</td>
<td>Resistance is futile!</td>
</tr>
</tbody>
</table>

Seriously, functional programming patterns are different.
FP has succint, concise and understandable syntax

The abstract nature of FP leads to considerably simpler programs. It also supports a number of powerful new ways to structure and reason about programs.

\[ x = x+1; \] We understand this syntax because we often resort to telling the computer what to do, but this equation really makes no sense at all!

Ask, don’t tell.
FP offers a different programming perspective

For me, the most important thing about FP isn’t that functional languages have some particular useful language features, but that it allows to think differently and simply about problems that you encounter when designing and writing applications. This is much more important than understanding any new technology or a programming language.

Tomas Petricek
http://tomasp.net/blog/
Quicksort
Quicksort

Erlang

-module( quicksort ).
-export( [qsort/1] ).

qsort([]) -> [];
qsort([X|Xs]) ->
    qsort([ Y || Y <- Xs, Y < X]) ++
    [X] ++
    qsort([ Y || Y <- Xs, Y >= X]).
Elixir

```elixir
defmodule Sort do
def qsort([],), do: []
def qsort([h | t]) do
  {lesser, greater} = Enum.split_with(t, &(&1 < h))
  qsort(lesser) ++ [h] ++ qsort(greater)
end
end
```
let rec qksort gt = function
  | [] -> []
  | x::xs ->
    let ys, zs = List.partition (gt x) xs in
    (qsort gt ys) @ (x :: (qsort gt zs))
Quicksort

Lisp

(defun qsort (list)
  (when list
    (destructuring-bind (x . xs) list
      (nconc (qsort (remove-if (lambda (a) (> a x)) xs))
             `(~(,x)
                (qsort (remove-if (lambda (a) (<= a x)) xs)))))))
Quicksort

Clojure

(defn qsort [[pivot & xs]]
  (when pivot
    (let [smaller #(>= % pivot)]
      (lazy-cat (qsort (filter smaller xs))
                [pivot]
                (qsort (remove smaller xs))))))
Quicksort

Racket

#lang racket
(define (quicksort < l)
  (match l
    [() ()]
    [(cons x xs)
     (let-values ([([xs-gte xs-lt]) (partition (curry < x) xs)])
       (append (quicksort < xs-lt)
                (list x)
                (quicksort < xs-gte)))]))
Haskell

csort [] = []
csort (x:xs) = csort [y | y <- xs, y < x] ++ [x] ++ csort [y | y <- xs, y >= x]
Quicksort

Haskell

import Data.List (partition)

qsort' :: Ord a => [a] -> [a]
qsort' [] = []
qsort' (x:xs) = qsort' ys ++ x : qsort' zs
where
  (ys, zs) = partition (< x) xs
Quicksort

Python

def qsort(xs):
    return qsort([y for y in xs[1:] if y < xs[0]]) + xs[:1] +
    qsort([y for y in xs[1:] if y >= xs[0]])
Functional programming is becoming more accessible

More language options.

Tooling, IDEs.

Supports.

Books.

Blogs, podcasts and screencasts.

Conferences and user groups.
Haskell is becoming more accessible

IntelliJ IDEA
Haskell is becoming more accessible

Atom
Haskell is becoming more accessible

Emacs

```haskell
import qualified Data.Hashable as Hash
import Data.Time

-- | Task priority
data Priority
  = L -- ^ low priority
  | M -- ^ medium priority
  | H -- ^ high priority
      deriving (Eq, Ord, Show, Read, Bounded, Enum)

-- | A single task
data Task = Task {
    tId :: Int, -- ^ task id. might change after display
    tHashID :: Int, -- ^ unique hash. never changes
    tDesc :: String, -- ^ task description
    tCreated :: UTCTime, -- ^ creation time (October 23, 4004 BC?)
    tDue :: Maybe UTCTime, -- ^ task due time
    tPri :: Maybe Priority, -- ^ task priority
    tProj :: Maybe String, -- ^ associated project
} deriving (Show, Read)

-- | Constructs a new Task. tHashID of a task is calculated
-- automatically based on the description and creation time
createTask :: Int
    -- ^ id of a new task
    -> String
    -- ^ task description
    -> UTCTime
    -- ^ creation time
    -> Maybe UTCTime
    -- ^ task due time
    -> Maybe Priority
    -- ^ task priority
    -> Maybe String
    -- ^ project to assign a task to
    -> Task
    -- ^ the new task
createTask taskId taskDesc taskCreated taskDue taskPri taskProj =
    Task taskId tHashID taskDesc taskCreated taskDue taskPri taskProj
    where tHashID = hashTask taskCreated taskDesc

-- | Create unique hash of a task based on creation time and description
```
Haskell is becoming more accessible

Leksah
Key Haskell concepts

High order functions, map, filter reduce (i.e., fold).
Recursion.
Pattern matching.
Currying.
Lazy/eager evaluation.
Strict/non-strict semantics.
Type inference.
Monads.
Continuations.
Closures.
Haskell
Haskell

Haskell is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing.

It is named after logician Haskell Curry.
Haskell

Side effects shall receive no mercy!
What can Haskell offer the programmer?

Purity
Unlike some other functional programming languages Haskell is pure. It doesn’t allow any side-effects. This is probably the most important feature of Haskell.

• Functions have no side effects.
• Given the same parameters, a function will always return the same result.
• There are other needs that can’t be met in a pure fashion.
What can Haskell offer the programmer?

**Referential transparency**

- Pure computations yield the same value each time they are invoked.
- Side effects like (uncontrolled) imperative update break this desirable property.
- Make it easier to reason about the behavior of programs.

```haskell
random :: Int
random = 4 -- chosen by fair dice roll, guaranteed to be random.

today :: String
today = "Mon 21 Sep 2020" -- guaranteed at the time of writing.

getInputChar :: Char
getInputChar = 'a' -- The user did type 'a', so what!?```
What can Haskell offer the programmer?

Higher-order functions

- Functions that take other functions as their arguments.
- Useful for refactoring code.
- Reduce the amount of repetition.

```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = smallerSorted ++ [x] ++ biggerSorted
    where
        smallerSorted = quicksort (filter (<=x) xs)
biggerSorted = quicksort (filter (>x) xs)
```
What can Haskell offer the programmer?

Immutable data

• Expressions in Haskell are immutable. They cannot change after they are evaluated.
• Immutability makes refactoring super easy and code much easier to reason about.
• To change an object, most data structures provide methods taking the old object and creating a new copy.

```haskell
>> let a = [1,2,3]
>> reverse a
[3,2,1]
>> a
[1,2,3]
```
What can Haskell offer the programmer?

**Laziness**
Haskell is *lazy* (technically speaking, it’s *non-strict*). This means that nothing is evaluated until it has to be evaluated.

- Laziness is important.
- Laziness let us separate producers and consumers and still get efficient execution.
- This allows us to work with infinite lists without getting stuck in an infinite computation.
What can Haskell offer the programmer?

**Laziness**

Haskell is *lazy* (technically speaking, it’s *non-strict*). This means that nothing is evaluated until it has to be evaluated.

In a strict language, evaluating `f 5 (29^{35792})` will first completely evaluate `5` (already done) and `29^{35792}` (which is a lot of work) before passing the results to `f`.
What can Haskell offer the programmer?

Laziness
Haskell is lazy (technically speaking, it’s non-strict). This means that nothing is evaluated until it has to be evaluated.

\[
\lambda: 1 \div 0
\]

*** Exception: divide by zero

\[
\lambda: (1 == 2-1) || (1 \div 0 == 1)
\]

True

\[
\lambda: (1 /= 2-1) && (1 \div 0 == 1)
\]

False

\[
\lambda: \text{head} \ [1, 2 \div 0, 3]
\]

1

\[
\lambda: \text{last} \ (\text{tail} \ [1, 2 \div 0, 3])
\]

3
What can Haskell offer the programmer?

**Strong typing**
Haskell is strongly typed, this means just what it sounds like. Unlike other strongly typed languages types in Haskell are automatically inferred.

- It’s impossible to inadvertently convert a `Double` to an `Int`, or follow a null pointer.
- Types are checked at compile-time.
- You can easily defined your own types.
What can Haskell offer the programmer?

Elegance
Another property of Haskell that is very important to the programmer, even though it doesn’t mean as much in terms of stability or performance, is the elegance of Haskell.

- A function definition usually resembles the informal description of the function very closely.
- To put it simply: stuff just works like you’d expect it to.
So what !?

All possible programs

code that works

code written in Haskell
Learn You a Haskell for Great Good!

A Beginner’s Guide

Miran Lipovača
Copyright and Material
Show me some code!
Hello, World!

module Main where

main :: IO ()
main = putStrLn "Hello, World!"
Hello, World!: Compile to native code

barbalala: ghc -o Hello Hello.hs
[1 of 1] Compiling Main
( Hello.hs, Hello.o )
Linking Hello ...
barbalala: ./Hello
Hello, World!
barbalala:
Hello, World!: Interpreter

barbalala: ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/
:? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load "Hello"
[1 of 1] Compiling Main ( Hello.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Hello, World!
*Main>
Quicksort in Haskell

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = quicksort lesser ++ [p] ++ quicksort greater

where
  lesser = filter (< p) xs
  greater = filter (>= p) xs
```
The Fibonacci sequence

fib :: (Eq a, Num a, Num b) => a -> b
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fib :: (Integral b, Integral a) => a -> b
fib n = round $ phi ** fromIntegral n / sq5
  where
    sq5 = sqrt 5 :: Double
    phi = (1 + sq5) / 2

fibs :: Num a => [a]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
The speed of Haskell

For most applications the difference in speed between C++ and Haskell is so small that it’s utterly irrelevant.
The speed of Haskell

There’s an old rule in computer programming called the "80/20 rule". It states that 80% of the time is spent in 20% of the code. The consequence of this is that any given function in your system will likely be of minimal importance when it comes to optimizations for speed. There may be only a handful of functions important enough to optimize.

Remember that algorithmic optimization can give much better results than code optimization.

Last but not least, Haskell offers substantially increased programmer productivity (Ericsson measured an improvement factor of between 9 and 25 using Erlang, a functional programming language similar to Haskell, in one set of experiments on telephony software.)
Haskell in Industry

Code written in Haskell is guaranteed to have no side effects.

...because no one will ever run it?
Functional programming languages