Notes on the history of functional programming and Haskell

June 6, 2018

Sebastian Muskalla

TU Braunschweig

Summer term 2018

Contents

Preface

These are notes for the first of my lectures on functional programming in Haskell taught at TU Braunschweig in the summer term of 2018. It contains a short overview of the history of functional programming and the development of Haskell.

The other lectures that are concerned with presenting Haskell and its concepts are not covered by these notes (except for the Lecture syllabus) However, there is an abundance of material on learning Haskell that anybody who is interested can use to get started (see Literature). Furthermore, the code examples that were used in the lecture are available here:
`https://labrador.tcs.cs.tu-bs.de/fp/code_examples` .

In case you spot a bug in these notes, please send me a mail: `s.muskalla@tu-bs.de` .

Sebastian Muskalla

Braunschweig, June 6, 2018

Literature

The main resource that was used to prepare the lectures is the book

"Thinking functionally wityh Haskell"
R. Bird.
Cambridge University Press, 2015.

Other resources that were used include:

- "Software-Entwicklung 1"
  A. Poetzsch-Heffter.
  Lecture slides, TU Kaiserslautern, 2009 (in German).
- "Fortgeschrittene funktionale Programmierung in Haskell"
  J. Betzendahl, S. Dresselhaus.
  Lecture slides, Universität Bielefeld, 2016 (in German).
  `https://github.com/FFPiHaskell/Vorlesung2016`
- "Funktionale Programmierung"
  J. Knoop.
  Lecture slides, TU Wien, 2017 (in German).
- "Fortgeschrittene funktionale Programmierung"
  J. Knoop.
  Lecture slides, TU Wien, 2018.
- "Methodical, industrial software-engineering using the Haskell functional programming panguage"
  H. V. Riedel
  Lecture slides, TU Wien, 2017.
- "Learn you a Haskell for great good!"
  Miran Lipovača
  No Starch Press, 2011.
  `http://learnyouahaskell.com`
- "Functional programming"
  P. Wadler
  Lecture, University of Edinburgh, 2011.
- "The first monad tutorial"
  P. Wadler
  Talk, YOW! 2013.
  `https://www.youtube.com/watch?v=yjmKMhJOJos`
- "What I Wish I Knew When Learning Haskell"
  S. Diehl
  `http://dev.stephendiehl.com/hask/`
- and the Haskell Wiki, e.g. `https://wiki.haskell.org/What_a_Monad_is_not` .

Notes on the history of functional programming and Haskell

History of programming

- Early computers: Strict separation between program (e.g. electronic circuits) and data (e.g. punch cards)
- von Neumann architecture: program and data share the same memory (stored-program computers)
- von Neumann-style imperative programing: Commands/statements (representing state changes) vs. expressions (representing data)
- Imperative programs are sequences of commands (with jumps, e.g. realized via go `to`)
- Dijkstra 1968: "Go To statement considered harmful"
  ↳ Structured programming (if, while, ...) & procedural programming
- Procedural programs: Functionality realized by procedures that can call each other
- Procedures have explicit parameters and return values, but still have side effects (like reading and writing the global state and intersection with the outside world)
- John Backus (Fortran language designer) in his Turing Award lecture 1977 "Can programming be liberated from the von Neumann style?"
  ↳ Boost of research into functional programming

What is functional programming?

- A (pure) function is a procedure without side effects: It can only read immutable parts of the global state
- Pure functions are functions in the sense of mathematics: If $x = y$, then $f(x) = f(y)$, no matter where this expression occurs in the code.
- In functional programming, functionality is realized by composing functions
- In the ideal case: No commands/statements, only expressions

History of functional programming

- Since 1950s: LISP and its dialects
- After Backus' lecture: *Research* into functional programming became popular...
- but its practical applications were limited to special fields (e.g. Erlang (since 1986) for communication systems)
- In the last 10 years: Rise of functional programming and multi-paradigm languages

Languages

- Non-pure functional languages: LISP (including dialects like Clojure, Scheme), ML (including dialects like (O)Caml), Erlang, …
- Multi-paradigm languages: Python, Ruby, JavaScript, …
- Rise and fall of Scala (around 2010)
- Since Java 8 (2014) and C++11: Some functional concepts in Java and C++
- Nowadays: Almost no new language without functional programming (Apples's Swift, Kotlin, Ceylon). Counter-example: Google's Go.

Haskell

- Since 1986
- Named after the logician Haskell Curry (1900 - 1982)
- Only popular pure functional language
- According to R. Bird: The most radical functional language
-
  - Compiled (but also has an interpreter)
  - Non-strict semantics, Lazy evaluation
  - Strong static typing
  - Concepts from category theory like monads
- Initially mostly a scientific project, but increasing use in practice in the last years

Environment

- The Haskell platform including the GHC is the de-facto standard implementation
- compiler GHC: (Glorious) Glasgow Haskell Compiler
- "interpreter" GHCi: Glasgow Haskell Compiler interactive version
- Warning: The default values of some settings ("language pragmas") differ by default between GHCi und GHC
- No popular IDE specifically for Haskell
- Used by me during the lecture: IntelliJ IDEA with HaskForce plugin

A tiny example

Listing 1: `../code/haskell/intro/Main.hs`

```haskell
module Main where

import Data.Char

main = echo

echo =
    do
        s ← getLine
        putStrLn $ map toUpper s
```

- Either compile by `ghc Main.hs` and execute using `./Main` (resp. `Main.exe`)
- or load in GHCi by `ghci Main.hs` and execute the main function by typing **`main`**

Why functional programming?

- Succinctness of the code(factor 5-10)
  ↳ Hope that maintenance costs decrease
- Functional programming has an easier mathematical theory, supports equational reasoning
- The absence of side-effects simplifies code and enabled optimization and easier parallelization: Consider `x = f(a); y = g(b)`. Changing the code to `y = g(b); x = f(a)` or even to `x = f(a) || y = g(b);` (i.e. evaluating in parallel) is not necessarily valid under the presence of side effects, but it is valid in a functional language
  (But: Automatic optimizations do not work very well in practice yet)
- Declarative rather than imperative programming style

Imperative vs. declarative programming

- Declarative programming: Telling the computer **what** to do
- Imperative programming: Telling the computer **how to do it**

Example: Computing the first *n* square numbers

In Haskell, declarative:

Listing 2: `../code/haskell/intro/SquareNumbers.hs`

```haskell
numbers = [1..]
square = \x → x*x
square_numbers = map square numbers
first_squares :: Int → [Integer]
first_squares n = take n square_numbers

main = do
    n_string ← getLine
    putStrLn $ show $ first_squares (read n_string :: Int)
```

- **Let** `numbers` be the positive numbers
- **Let** `square` be the function that squares
- **Let** `square_numbers` be the list of all squares of positive numbers
- **Let** `first_squares` be the function that takes the first *n* square numbers from the list
- (Plus **`main`** function that reads a number *n* and prints the first *n* square numbers)

In Java, imperative:

Listing 3: `../code/java/src/SquareNumbers.java`

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Scanner;

public class SquareNumbers
{
    public static void main (String[] args)
    {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        List<Integer> squares = first_squares(n);
        System.out.println(squares.toString());
    }

    static List<Integer> first_squares (int n)
    {
        List<Integer> numbers = new ArrayList<>();
        for (int i = 1; i <= n; i++)
        {
            numbers.add(i);
        }

        List<Integer> squares = new ArrayList<>();
        Iterator<Integer> itr = numbers.iterator();
        while (itr.hasNext())
        {
            Integer num = itr.next();
            Integer square = num * num;
            squares.add(square);
        }
        return squares;
    }
}
```

(The code can of course be optimized. It is written like this to highlight the issue.)

- **Create** a new list
- **Insert** the numbers from 1 to *n* into it in increasing order
- **Create** another list.
- **Iterate** over the first list from the beginning to the end
- In each step, **take** a number from the first let, **square** it and **append** it to the end of the second list
- **Return** the resulting list of squares
- (Plus `main` method that reads a number *n* and prints the first *n* square numbers)

In Java, declarative:

Using Java 8, we can reproduce the declarative Haskell version

Listing 4: `../code/java/src/SquareNumbers2.java`

```java
import java.util.List;
import java.util.Scanner;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class SquareNumbers2
{
    public static void main (String[] args)
    {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        List<Integer> squares = first_squares(n);
        System.out.println(squares.toString());
    }

    static List<Integer> first_squares (int n)
    {
        IntStream numbers = IntStream.iterate(1, i -> i + 1);
        IntStream squares = numbers.map((x) -> x * x);
        return squares.boxed().limit(n).collect(Collectors.toList());
    }
}
```

Lecture syllabus

History of functional programming and Haskell

Basic Haskell

- **Basic data types and their values:** `Bool`, `Int`, `Integer`, ..., `Char`, tuples, `()`, `undefined`
- **Functions:** if-then-else, guards, pattern matching, `let` ... `in`, `where`, recursion, currying, sections
- **Lists:** basic operations, pattern matching, sorting

Some advanced concepts

- **Higher-order functions:** composition, `map`, `reduce` / `fold`, $
- **Data types:** `type`, `newtype`, `data`, constructors, enums, recursive types, `Maybe`, records
- **Type classes:** Constrains, defining classes, implementing instances, `Eq`, `Ord`, `Show`, `Num`
- **Lazy evaluation:** strictness, infinite lists, memoization
- **Type inference and its pitfalls:** type holes, monomorphism restriction, mutually recursive binding groups, polymorphic recursion

Monads

- **IO in functional languages:** non-pure IO (e.g. SML), linear types (e.g. Clean), IO in Haskell
- **Functor, Applicative, Monad**
- **Examples:** Trivial monad (Burrito), `Maybe`, `List`, `IO`
- **Quick teaser:** parsing as a monad, `Alternative` and `MonadPlus`, `State` and `ST`