Functional Programming

Søren Haagerup

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

October 24, 2016

Practical Information

- **Lectures (I):**
  - Søren Haagerup
  - Monday 14-16 (Weeks 43-44)
  - Monday 12-14 (Weeks 45-50)
- **Exercises (TE):**
  - Sævar Berg Sævarsson
  - U157: Wednesday 10-12 (Weeks 43-50)
- **Labs (TL):**
  - Sævar Berg Sævarsson
  - IMADA Terminalrum: Friday 10-12 (Week 44)
  - IMADA Terminalrum: Friday 14-16 (Weeks 45, 47, 49)

Exam rules

- Evaluation based on 2 mandatory projects
  - Logic Programming Project (due November 15th)
  - Functional Programming Project
- You have to pass both projects to pass the course.
- An obligatory assignment is an exam project. Cooperation is not permitted: It will be considered cheating and will be treated as such.
- If you fail one project, you only need to do the reexam for this part of the course.
- You have to hand in at least one of the projects to qualify for the reexam! An empty document with only your name on is enough.
- But: Please do your best to pass the first time.

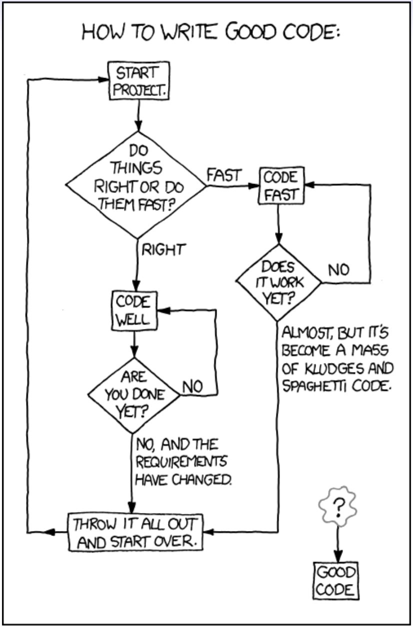
Literature

We draw upon many different sources. In the first couple of lectures we will mainly use:

GS Giesl, Jürgen and Schneider-Kamp, Peter (2012).
Programming Languages. [Download PDF]

L Lipovača, Miran (2011).
Learn You a Haskell for Great Good! [Available online]

INTRO


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Why another programming language?

A programming language forms the way you solve problems. Haskell is a language which

- encourages a style of programming which result in few bugs
- encourages clear separation of concerns between different parts of your program
- allows you to solve problems on a fairly high level of abstraction, while still maintaining high performance

This is done by a set of features which is quite different from other general purpose languages like Java, Python.

High-level comparison to Java and Python

- The **type safety** known from other statically typed languages like **Java**, just much better. *"If it compiles, it works!"*.
- The **conciseness** known from languages like **Python** - you can write powerful programs with very few lines of code.
- Completely different way of structuring programs, compared to object-oriented languages.

Designed by Researchers

- Solid scientific foundation, building on lambda calculus and type theory.
- Fewer “dirty hacks” compared to other languages, where many design decisions have come as an afterthought.

History (1930-1950)

- *1930s and 1940s*
  Alonzo Church develops the **Lambda Calculus** and **Simply Typed Lambda** calculus

- *1950s*
  Haskell B. Curry, Robert Feys, work on **Combinatory Logic**

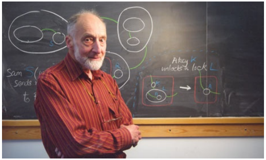
Alonzo Church

Haskell B. Curry

History (1950-)

- *1950s* John McCarthy develops **Lisp**, the first functional language, with some influences from the lambda calculus,
- *1960s-1970s* Roger Hindley, Robin Milner, theoretical work on type inference, and the programming language **ML**
- *1970s-1980s* David Turner - investigates *lazy* functional languages, culminating in the **Miranda** language
- *1987+* **Haskell** is born

John McCarthy



Robin Milner

Haskell

In September 1987 at the FPCA conference [Functional Programming Languages and Computer Architecture (FPCA ’87) in Portland, Oregon], the functional programming language community decided to design a **common language**:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages.

Haskell

Since then, there have been two major revisions of the language: Haskell98 and **Haskell2010**.

It enjoys increasing interest from the industry, especially in projects requring *high assurance* of program correctness - aerospace, defense and finance industries.

Haskell Popularity

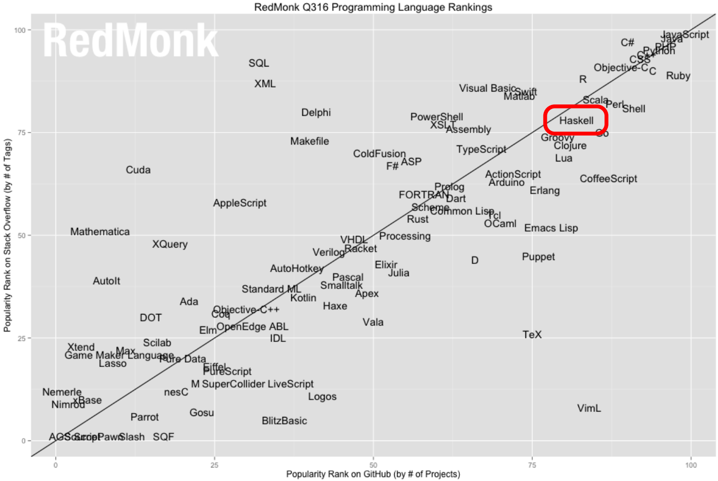


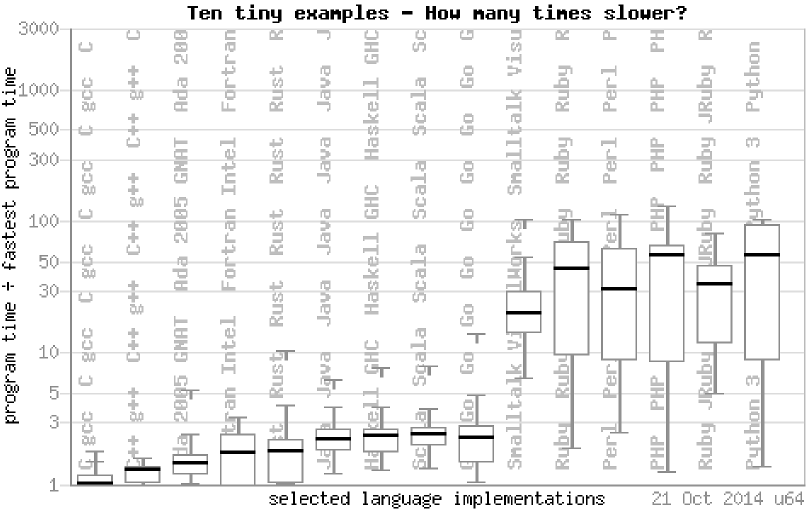
Figure: The RedMonk Programming Language Rankings: June 2016. Activity on GitHub/StackOverflow.

Haskell at Facebook - HAXL

- Facebook is currently starting to use Haskell for a Data Access Layer, used internally for writing programs that detect malware and spam.
- Haskell is used to deal cleanly with concurrency issues

How fast is Haskell?

Syddansk Universitet
University of Southern Denmark



Referential transparency?

Consider this **Python** program:

$$
\begin{array}{l}
a = f(5) \\
b = f(5) \\
\textbf{print } a \\
\textbf{print } b
\end{array}
$$

Are the two printed lines identical?

Referential transparency?

Consider this **Python** program:

```
evil = 0
def f(x):
    global evil
    evil += x
    return evil

a = f(5)
b = f(5)
print a
print b
```

Are the two printed lines identical?

Imperative programming: For-loops + index calculations

```
quicksort(A, i, k):
  if i < k:
    p := partition(A, i, k)
    quicksort(A, i, p - 1)
    quicksort(A, p + 1, k)

partition(array, left, right)
 pivotIndex := choosePivot(array, left, right)
 pivotValue := array[pivotIndex]
 swap array[pivotIndex] and array[right]
 storeIndex := left
 for i from left to right - 1
     if array[i] < pivotValue
         swap array[i] and array[storeIndex]
         storeIndex := storeIndex + 1
 swap array[storeIndex] and array[right]
 return storeIndex
```

Functional programming: Recursion + pattern matching

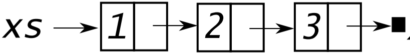
$$
\begin{aligned}
&qsort :: (Ord\ a) \Rightarrow [a] \rightarrow [a]\\
&qsort\ [\,] \qquad\quad = [\,]\\
&qsort\ (x : xs) = qsort\ (filter\ (\leqslant x)\ xs)\\
&\qquad\qquad\qquad\quad +\!\!+\ x : qsort\ (filter\ (>x)\ xs)
\end{aligned}
$$

- Loops are gone, and we use recursion instead
- Indices are gone - we use pattern matching instead
- `partition` is gone - we use higher-order function *filter* from the Haskell Prelude

Immutable data structures

- With immutable data structures, the programmer never changes or destroys objects
- If an object is not used anymore, it is the garbage collectors job to remove it.

```
xs = [1,2,3]
```




```
xs =
ys =
```

What is Haskell?

Haskell

- Purely functional ($\rightarrow$ equational reasoning)
- Functions are values
- Immutable data structures
- Lazy evaluation
- Garbage collection
- Static types
- Implicit types
- Polymorphic functions
- Type classes and modules

Other languages

- Imperative ($\rightarrow$ side-effectful)
- Sep. function/value scopes
- Mutable data structures
- Eager evaluation
- Manual memory mgmt.
- Dynamic types
- Explicit types
- Monomorphic functions
- Classes and namespaces

EXPRESSIONS, TYPES AND PATTERNS

Function application in Haskell

Math	**Haskell**
$f(x)$	$f\ x$
$f(x,y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ y)$
$f(x,g(x))$	$f\ x\ (g\ y)$
$f(x)g(x)$	$f\ x * g\ y$

- Function and argument names begin with lower-case letter.
- *camelCase* is preferred to *lower_case_with_underscores*.
- f', f'' etc. are valid function names.

- Prefix $\rightarrow$ infix: $div\ 295\ 7 = 295\ `div`\ 7$
- Infix $\rightarrow$ prefix: $40 + 2 = (+)\ 40\ 2$

DEMO 1: GHCi - Interactive REPL environment

```
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/
Prelude>
```

1. Demo of simple numerical calculations
2. Demo of simple list functions *head*, *tail*, (!!), *take*, *drop*, *length*, *sum*, *product*, $(+\!\!+)$
3. Demo of **.hs**-files with bindings, indentation and let/where-subscoping

The evaluation model

Let's say we have the definition

$$square\ a = a * a$$

and we want to evaluate the *expression*

$$square\ (3 + 4)$$

Every definition in Haskell corresponds to a **reduction rule**, which are applied during **evaluation** of expressions. An example of a reduction:

$$\begin{array}{lll} & square\ (3+4) & \\ \equiv & square\ 7 & \text{-- apply } (+) \\ \equiv & 7 * 7 & \text{-- apply } square \\ \equiv & 49 & \text{-- apply } (*) \end{array}$$

The evaluation model

A different reduction:

$$
\begin{aligned}
& \mathit{square}\ (3+4) & \\
\equiv\ & (3+4) * (3+4) & \text{-- apply } \mathit{square} \\
\equiv\ & 7 * (3+4) & \text{-- apply } (+) \\
\equiv\ & 7 * 7 & \text{-- apply } (+) \\
\equiv\ & 49 & \text{-- apply } (*)
\end{aligned}
$$

When at a irreducible expression, we say that the expression is in its **normal form**.

The evaluation model

- **Church-Rosser Theorem:** If two reduction strategies arrive at a **normal form**, these are equal to each other.
- It turns out, that the order of the reductions you apply matters - if choosing the wrong evaluation strategy, you might never arrive at the normal form.
- Also **time** and **space** usage of your algorithm might be different for different reduction strategies.
- For the functions defined in the first few lectures it will be feasible just to use *any* evaluation order.

Prerequisites for evaluation

Haskell is **statically typed**. Therefore the compilation + evaluation process of every Haskell script consists of the following separate phases:

1. Syntax analysis
2. Type analysis (In Haskell: Type inference + Type checking)
3. Evaluation

DEMO 2: Types in GHCi

Every valid expression e in Haskell has a type t. This is written as

$$e :: t$$

Examples:

`'5'`	$:: Char$
$True$	$:: Bool$
(`'5'`, $True$)	$:: (Char, Bool)$
$length$ `"hello"`	$:: Int$
$null\ [2,3]$	$:: Bool$

Use `:set +t` or :t to see types in GHCi

Error messages - Type inference

- Reading the error messages from a type-inferring language is tricky:
  - The implementation issues a message when it realizes that the type inference cannot possibly succeed.
  - It may find this dead-end many source code lines past the actual error.
  - It may "explain" this dead-end in terms of the last inference step which exposed the contradiction, and not in terms of that line where the inference started to go in the wrong direction.

Example: Try e.g. 4 *div* 3 instead of 4 ‘*div*‘ 3

Number types

- *Int* is the type “machine integer”:
    - fast machine arithmetic
    - which wraps around when there is overflow
- *Integer* is the type “mathematical integer”:
    - much slower arithmetic
    - with infinite precision (or until memory runs out).
- *Float* is a single precision floating point number type according to the IEEE 754 standard.
- *Double* is the corresponding double precision type.

Functions - $f :: a \to b$

A named function is defined as

$$name\ parameter = expression$$

where

$$\begin{aligned} parameter &:: argtype \\ expression &:: restype \\ name &:: argtype \to restype \end{aligned}$$

Functions - $f :: a \to b$

A function of n parameters has type

$$f :: a1 \to a2 \to ... \to an \to a$$

This is just a **special case** of the function of 1 parameter case! Consider $\to$ as an operator which associates to the right. The function

$$f :: a1 \to (a2 \to ... \to (an \to a)...)$$

is a function which takes 1 parameter and returns a function taking $n - 1$ parameters.
When calling f:

$$f\ x1\ x2\ ...\ xn \equiv (((f\ x1)\ x2)...)\ xn$$

Questions about functions

Is this a valid function?

$$f\ a\ b = a\ b$$

What is the type of f?

What about this function?

$$g\ a\ b = b\ a$$

Is this a valid expression?

$$f\ '-'$$

What about this expression?

$$f\ length$$

Tuples - $e :: (a, b)$

Constructor

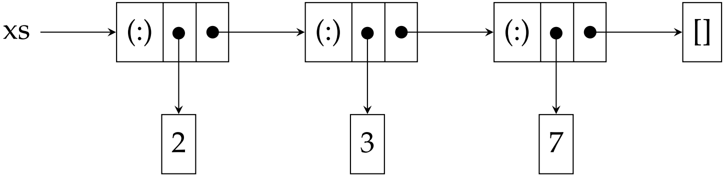
$$(,) :: a \to b \to (a, b)$$

Pattern matching (destructing)

$$\begin{aligned} fst\ (x, \_) &= x \\ snd\ (\_, y) &= y \\ add\ (x, y) &= x + y \end{aligned}$$

Pattern matching is the *only* way to get values out of the tuple - functions from the standard library does this too.
What are the types of these functions?

Lists - $e :: [a]$



Constructors

$$
\begin{aligned}
(:) &:: a \to [a] \to [a] \\
[\,] &:: [a]
\end{aligned}
$$

Pattern matching (destructing) – Try to define a function $length :: [a] \to Int$ **to compute the length of a list?**

$$
\begin{aligned}
&sum :: [Integer] \to Integer \\
&sum\ [\,] \qquad\quad = 0 \\
&sum\ (x : xs) = x + sum\ xs
\end{aligned}
$$

$$
\begin{aligned}
head\ (x{:}\ \_) &= x \\
tail\ (\_ : xs) &= xs
\end{aligned}
$$

Booleans

Not language primitives – Booleans and their operations are defined in the standard library! **Constructors**

$$\begin{aligned} \mathit{True} &:: \mathit{Bool} \\ \mathit{False} &:: \mathit{Bool} \end{aligned}$$

Pattern matching (destructing)

$$\begin{aligned} \mathit{True} \wedge a &= a \\ \mathit{False} \wedge \_ &= \mathit{False} \end{aligned}$$

$$\begin{aligned} \mathit{False} \vee a &= a \\ \mathit{True} \vee \_ &= \mathit{True} \end{aligned}$$

We could define our own inline-if:

$$\mathit{iif} :: \mathit{Boolean} \to a \to a \to a$$

How would the definition look? Haskell also provides special syntax: **if** $(a < 5)$ **then** `"Hello"` **else** `"World"`.

Maybe - $e :: Maybe\ a$

Constructors

$$\begin{aligned} &Just &&:: a \to Maybe\ a \\ &Nothing &&:: Maybe\ a \end{aligned}$$

Pattern matching (destructing)

$$\begin{aligned} &maybeAdd\ Nothing\ \_ &&= Nothing \\ &maybeAdd\ \_\ Nothing &&= Nothing \\ &maybeAdd\ (Just\ x)\ (Just\ y) &&= Just\ (x + y) \end{aligned}$$

Example: Naïve Fibonacci Implementation

$$
\mathit{fib}(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ \mathit{fib}(n-1) + \mathit{fib}(n-2), & n \geqslant 2 \end{cases}
$$

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{fib}\ 0 = 0 \\
&\mathit{fib}\ 1 = 1 \\
&\mathit{fib}\ n = \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)
\end{aligned}
$$

Example: Greatest Common divisor

$$gcd(a, b) = \begin{cases} a, & b = 0 \\ gcd(b, a \bmod b), & b > 0 \end{cases}$$

$$\begin{aligned} &gcd :: Integer \to Integer \to Integer \\ &gcd\ a\ 0 = a \\ &gcd\ a\ b = gcd\ b\ (a\ \text{`}mod\text{`}\ b) \end{aligned}$$

Example: Digits of a number

$$
\begin{array}{l}
\mathit{digits} :: \mathit{Integer} \rightarrow [\mathit{Integer}] \\
\mathit{digits}\ n \\
\quad \mid n \equiv 0 \qquad\quad = [\,] \\
\quad \mid \mathit{otherwise} = n \text{ `}\mathit{mod}\text{` } 10 : \mathit{digits}\ (n \text{ `}\mathit{div}\text{` } 10)
\end{array}
$$

```
*Main> digits 7331
[1,3,3,7]
```

Division tests

$$
\begin{aligned}
&\mathit{dividesThree} :: \mathit{Integer} \rightarrow \mathit{Bool}\\
&\mathit{dividesThree}\ x = (\mathit{sum} \circ \mathit{digits})\ x \text{ `}\mathit{mod}\text{` } 3 \equiv 0\\
&\mathit{dividesNine} :: \mathit{Integer} \rightarrow \mathit{Bool}\\
&\mathit{dividesNine}\ x\ \ = (\mathit{sum} \circ \mathit{digits})\ x \text{ `}\mathit{mod}\text{` } 9 \equiv 0
\end{aligned}
$$

Division tests

$$
\begin{aligned}
&\mathit{alternate}, \mathit{alternate}' :: [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{alternate}\ [\,] \qquad\qquad = [\,] \\
&\mathit{alternate}\ (d : ds) \ \ = d : \mathit{alternate}'\ ds \\
&\mathit{alternate}'\ [\,] \qquad\quad\ = [\,] \\
&\mathit{alternate}'\ (d : ds) = (-d) : \mathit{alternate}\ ds
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dividesEleven} :: \mathit{Integer} \to \mathit{Bool} \\
&\mathit{dividesEleven}\ x \quad = (\mathit{sum} \circ \mathit{alternate} \circ \mathit{digits})\ x\ `\mathit{mod}`\ 11 \equiv 0
\end{aligned}
$$

Modular exponentiation

We want to compute $r = b^e \bmod m$.

We use that any number e can be written as $2e' + k$ where $e' = \lfloor \frac{e}{2} \rfloor$ and $k = e \bmod 2$.

Then

$$
\begin{aligned}
r = b^e \bmod m = b^{2e'+k} \bmod m &= (b^2)^{e'} b^k \bmod m \\
&= (b \cdot b \bmod m)^{e'} (b \bmod m)^k \bmod m
\end{aligned}
$$

```haskell
powm :: Integer -> Integer -> Integer -> Integer -> Integer
powm b 0 m r = r
powm b e m r
  | e `mod` 2 == 1 = powm b' e' m (r * b `mod` m)
  | otherwise      = powm b' e' m r
  where
    b' = b * b `mod` m
    e' = e `div` 2
```

Resources

- Haskell Platform
- Hackage
- Hoogle