CM0081 Automata and Formal Languages
Introduction to Haskell

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2024-1

Features/Advantages

- Purely functional (verification)
- Statically typed (type-safe and maintainability)
- Lazy evaluation (unbounded data structures and performance)
- Garbage collected memory (no need for pointers)

Functions

Example (function application)

Factorial function.

```
fac n = product [1..n]
```

Note: We use '`f n`' instead of '`f(n)`' for function application.

Types

Question

Is the `fac` function correct?

Types

Question

Is the `fac` function correct?

Example

```
import Numeric.Natural

fac :: Natural -> Natural
fac n = product [1..n]
```

By writing down the type of the function we could avoid run-time errors.

Types

Question

Is the `fac` function correct?

Example

```haskell
import Numeric.Natural

fac :: Natural -> Natural
fac n = product [1..n]
```

By writing down the type of the function we could avoid run-time errors.

Other implementations for the factorial

Google for 'The evolution of a HASKELL programmer'.

Curryfication

Example

Whiteboard.

Lists

Inductive definition

Haskell has built-in syntax for lists, where a list is either:

- the empty list, written `[]`, or
- a first element `x` and a list `xs`, written `(x : xs)`.

Lists

Example (pattern matching on lists)

```
length :: [Int] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Lists

Example (pattern matching on lists)

```
length :: [Int] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Question

What about the `length` function on lists of Booleans?

Lists

Example (pattern matching on lists)

```
length :: [Int] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Question

What about the `length` function on lists of Booleans?

```
length :: [Bool] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Lists

Example (pattern matching on lists)

```
length :: [Int] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Question

What about the `length` function on lists of Booleans?

```
length :: [Bool] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Question

Can we avoid the boilerplate lstlisting? Yes!

Parametric Polymorphism

Example (basic functions from the @Data.List@ library)

(i) Returns the length of a finite list as an `Int`.

```
length :: [a] -> Int
```

(ii) Append two lists.

```
(++) :: [a] -> [a] -> [a]
```

(iii) Extract the first element of a list, which must be non-empty.

```
head :: [a] -> a
```

Parametric Polymorphism

Example (basic functions from the @Data.List@ library)

(i) Extract the last element of a list, which must be finite and non-empty.

```
last :: [a] -> a
```

(ii) Extract the elements after the head of a list, which must be non-empty.

```
tail :: [a] -> [a]
```

(iii) Return all the elements of a list except the last one. The list must be non-empty.

```
init :: [a] -> [a]
```

(iv) Test whether a list is empty.

```
null :: [a] -> Bool
```

Lazy Evaluation

Description

Nothing is evaluated until necessary.

Lazy Evaluation

Example

Infinite (unbounded) list.

```
ones :: [Int]
ones = 1 : ones
```

Lazy Evaluation

Example

Infinite (unbounded) list.

```
ones :: [Int]
ones = 1 : ones
```

The expression `take n` applied to a list `xs` returns the prefix of `xs` of length `n`, or `xs` itself if `n` $>$ `length xs`.

```
take :: Int -> [a] -> [a]
```

Lazy Evaluation

Example

Infinite (unbounded) list.

```
ones :: [Int]
ones = 1 : ones
```

The expression `take n` applied to a list `xs` returns the prefix of `xs` of length `n`, or `xs` itself if `n` $>$ `length xs`.

```
take :: Int -> [a] -> [a]
```

Question

Which is the value of `take 5 ones`?

Lazy Evaluation

Example

Infinite (unbounded) list.

```
ones :: [Int]
ones = 1 : ones
```

The expression `take n` applied to a list `xs` returns the prefix of `xs` of length `n`, or `xs` itself if `n` $>$ `length xs`.

```
take :: Int -> [a] -> [a]
```

Question

Which is the value of `take 5 ones`? `[1,1,1,1,1]`

Lazy Evaluation

Example (also in other programming languages)

Non-terminating function.

```
foo :: Int -> Bool
foo n = foo (n + 1)
```

Boolean disjunction.

```
bar :: Int -> Bool
bar n = True || foo n
```

Lazy Evaluation

Example (also in other programming languages)

Non-terminating function.

```
foo :: Int -> Bool
foo n = foo (n + 1)
```

Boolean disjunction.

```
bar :: Int -> Bool
bar n = True || foo n
```

Question

Which is the value of `bar 10`?

Lazy Evaluation

Example (also in other programming languages)

Non-terminating function.

```
foo :: Int -> Bool
foo n = foo (n + 1)
```

Boolean disjunction.

```
bar :: Int -> Bool
bar n = True || foo n
```

Question

Which is the value of `bar 10`? `True`

Lazy Evaluation

Example

(From stackoverflow.com/questions/30688558/)

```
dh :: Int -> Int -> (Int, Int)
dh d q = (2^d, q^d)

a, b :: (Int, Int)
a = dh 2 (fst b)
b = dh 3 (fst a)
```

Lazy Evaluation

Example

(From `stackoverflow.com/questions/30688558/`)

```
dh :: Int -> Int -> (Int, Int)
dh d q = (2^d, q^d)

a, b :: (Int, Int)
a = dh 2 (fst b)
b = dh 3 (fst a)
```

Question

Which is the value of a?

Lazy Evaluation

Example

(From `stackoverflow.com/questions/30688558/`)

```
dh :: Int -> Int -> (Int, Int)
dh d q = (2^d, q^d)

a, b :: (Int, Int)
a = dh 2 (fst b)
b = dh 3 (fst a)
```

Question

Which is the value of a? `(4,64)`

Higher-Order Functions

Description

Functions are first-class citizen.

Higher-Order Functions

Example

The expression `map f xs` is the list obtained by applying `f` to each element of `xs`:

```
map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]
```

Higher-Order Functions

Example

The expression `map f xs` is the list obtained by applying `f` to each element of `xs`:

```
map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]
```

The function `map` can defined by

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

Higher-Order Functions

Example

The expression `map f xs` is the list obtained by applying `f` to each element of `xs`:

```
map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]
```

The function `map` can defined by

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

Question

Which is the value of `map (+1) [1..5]`?

Higher-Order Functions

Example

The expression `map f xs` is the list obtained by applying `f` to each element of `xs`:

```
map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]
```

The function `map` can defined by

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

Question

Which is the value of `map (+1) [1..5]`? `[2,3,4,5,6]`

Higher-Order Functions

Example

The function `foldr` applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] = x1 `f` (x2 `f` ... (xn `f` z)...)
```

Higher-Order Functions

Example

The function `foldr` applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] = x1 `f` (x2 `f` ... (xn `f` z)...)
```

The function `foldr` can be defined by

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x : xs) = f x (foldr f z xs)
```

Higher-Order Functions

Example

The function `foldr` applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] = x1 `f` (x2 `f` ... (xn `f` z)...)
```

The function `foldr` can be defined by

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x : xs) = f x (foldr f z xs)
```

Question

Which is the value of `foldr (+) 0 [1,2,3]`?

Higher-Order Functions

Example

The function `foldr` applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] = x1 `f` (x2 `f` ... (xn `f` z)...)
```

The function `foldr` can be defined by

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x : xs) = f x (foldr f z xs)
```

Question

Which is the value of `foldr (+) 0 [1,2,3]`? 6

Higher-Order Functions

Example

The function `foldl` applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

```
foldl f z [x1, x2, ..., xn] = (...((z `f` x1) `f` x2) `f`...) `f` xn
```

Higher-Order Functions

Example

The function `foldl` applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

```
foldl f z [x1, x2, ..., xn] = (...((z `f` x1) `f` x2) `f`...) `f` xn
```

The function `foldl` can be defined by

```
foldl f z []       = z
foldl f z (x : xs) = let z' = z `f` x
                     in foldl f z' xs
```

Higher-Order Functions

Example

The function `foldl` applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

```
foldl f z [x1, x2, ..., xn] = (...((z `f` x1) `f` x2) `f`...) `f` xn
```

The function `foldl` can be defined by

```
foldl f z []       = z
foldl f z (x : xs) = let z' = z `f` x
                     in foldl f z' xs
```

Question

Which is the value of `foldl (+) 0 [1,2,3]`?

Higher-Order Functions

Example

The function `foldl` applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

```
foldl f z [x1, x2, ..., xn] = (...((z `f` x1) `f` x2) `f`...) `f` xn
```

The function `foldl` can be defined by

```
foldl f z []       = z
foldl f z (x : xs) = let z' = z `f` x
                     in foldl f z' xs
```

Question

Which is the value of `foldl (+) 0 [1,2,3]`? 6

Algebraic Data Types

Example

```
data Bool = True | False
```

Algebraic Data Types

Example

```
data Bool = True | False
```

Functions by pattern matching

```
(||) :: Bool -> Bool -> Bool
True  || _ = True
False || x = x
```

Algebraic Data Types

Example (recursive data type)

```haskell
data Nat = Zero | Succ Nat
```

Algebraic Data Types

Example (recursive data type)

```
data Nat = Zero | Succ Nat
```

Structural recursive function by pattern matching

```
(+) :: Nat -> Nat -> Nat
Zero     + n = n
(Succ m) + n = Succ (m + n)
```

Algebraic Data Types

Example (polymorphic data type)

```
data List a = Nil | Cons a (List a)
data [] a   = []  | a : [a]
```

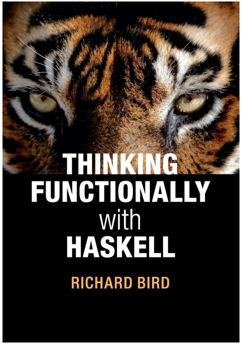
The Real World

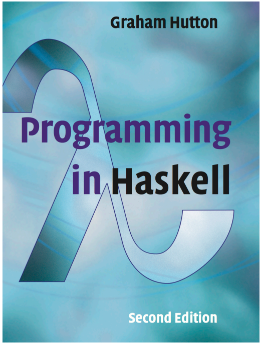
- Haskell in Industry (`www.haskell.org/haskellwiki/Haskell_in_industry`).
- Applications (`www.haskell.org/haskellwiki/Libraries_and_tools`).

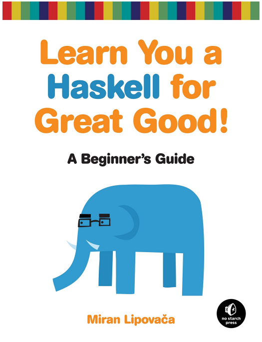
Using Haskell

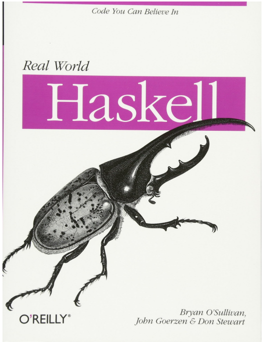
- Homepage: `www.haskell.org`
- GHC: The Glorious Glasgow Haskell Compilation System
- GHCi: Interactive interpreter
- Toolchain: `www.haskell.org/downloads`
  - For installing GHC we suggest to use GHCUP.
  - For installing libraries and compiling programs you can use STACK or CABAL-INSTALL.
- Hackage: The Haskell package repository
- Community: `www.haskell.org/community/`

Some Books


THINKING
FUNCTIONALLY
with
HASKELL
RICHARD BIRD


Graham Hutton
Programming
in Haskell
Second Edition


Learn You a
Haskell for
Great Good!
A Beginner's Guide
Miran Lipovača


Code You Can Believe In
Real World
Haskell
O'REILLY
Bryan O'Sullivan,
John Goerzen & Don Stewart

Some Books

- Bird, R. [2015]. Thinking Functionally with Haskell. Cambridge University Press.
- Hutton, G. [2007] [2016]. Programming in Haskell. 2nd ed. Cambridge University Press.
- Lipovača, M. [2011]. Learn You a Haskell for Great Good! No Starch Press.
- O'Sullivan, B., Goerzen, J. and Stewart, D. [2008]. Real World Haskell. O'Really Media, Inc.

Bonus Slides: Testing with QuickCheck

A paper

Claessen, Koen and Hughes, John [2000]. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP'00. DOI: https://doi.org/10.1145/357766.351266.

[†] See www.sigplan.org/Awards/ICFP/.

Bonus Slides: Testing with QuickCheck

A paper

Claessen, Koen and Hughes, John [2000]. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP'00. DOI: https://doi.org/10.1145/357766.351266.

Most Influential ICFP Paper Award 2010†

> *'The techniques described in the paper have spawned a significant body of follow-on work in test case generation. They have also been adapted to other languages …'*

†See www.sigplan.org/Awards/ICFP/.

Bonus Slides: Testing with QuickCheck

An open source library

QuickCheck on Hackage.[†]

[†] http://hackage.haskell.org/package/QuickCheck.

Bonus Slides: Testing with QuickCheck

An open source library

QuickCheck on Hackage.[†]

Commercialisation

QuviQ (`www.quviq.com/`).

[†] `http://hackage.haskell.org/package/QuickCheck`.

Bonus Slides: Testing with QuickCheck

Adaptations

QuickCheck has been ported to various languages (Wikipedia 2024-02-02).

C	C#	C++	Chicken	Clojure
Common Lisp	Coq	D	Elm	Elixir
Erlang	F#	Factor	Go	Io
Java	JavaScript	Julia	Logtalk	Lua
Mathematica	Objective-C	OCaml	Perl	Prolog
PHP	Pony	Python	R	Racket
Ruby	Rust	Scala	Scheme	Smalltalk
Standard ML	Swift	TypeScript	VB.NET	Vhiley

Bonus Slides: Testing with QuickCheck

False positive

The program works properly but the test pointed out a fail:

- There is a bug elsewhere.
- There is an error in the specification.

False positive

The program works properly but the test pointed out a fail:

- There is a bug elsewhere.
- There is an error in the specification.

False negative

There is a bug in the program but the test passed.

Recall Dijkstra's 1969 famous quote:

> *'Testing shows the presence, not the absence of bugs.'*

Bonus Slides: Testing with QuickCheck

Example

See demo.