# Course recap

Lecture 13 of CSE 3100
Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

# Lecture plan

- Overview of course material
- Organization of the exam
- Q&A
- Course survey

# Course recap

# Lecture 1: What is Haskell?

Haskell is a statically typed, lazy, purely functional programming language.

**Static types** All types are checked at compile time[1]

**Laziness** Expressions are only evaluated when required

**Purity** Functions do not have side effects

[1]Static typing ≠ explicit type annotations: Haskell can infer types automatically!

# Lecture 1: Pure vs effectful languages

What can happen when we call a function?

- It can return a value
- It can modify a (global) variable
- It can do some IO (read a file, write some output, ...)
- It can throw an exception
- It can go into an infinite loop
- ...

In a **pure** language (like Haskell), a function can only return a value or loop forever.

# Lecture 1: What is a type?

A type is a name for a collection of values.

- Basic types: `Bool`, `Int`,
- `Integer`, `Float`, `Double`, `Char`, `String`
- List types: `[a]`
- Tuple types: `(a,b)`, `(a,b,c)`, ...
- Function types: `a -> b`

# Lecture 1: List comprehensions

We can construct new lists using a list comprehension:

```
> [ x*x | x <- [1..10] , even x ]
[4,16,36,64,100]
```

The part `x <- [1..10]` is called a generator.

The predicate `even x` is called a guard.

# Lecture 2: Pattern matching and recursion

We can define new functions by pattern matching and recursion:

```
product :: Num a => [a] -> a
product []     = 1
product (x:xs) = x * product xs

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)
zip _      _      = []
```

# Lecture 2: Property-based testing

Instead of writing individual test cases, we can write down properties of our programs and generate test cases from those.

```
prop_reverse xs = reverse (reverse xs) == xs

prop_replicate n x =
  forAll (chooseInt (0,n-1)) (\i ->
    replicate n x !! i == x
```

QuickCheck will shrink counterexamples to their smallest form.

# Lecture 3: Defining datatypes

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

occurs :: Eq a => a -> Tree a -> Bool
occurs x (Leaf y)   = x == y
occurs x (Node l r) = occurs x l || occurs x r

flatten :: Tree a -> List a
flatten (Leaf x)   = [x]
flatten (Node l r) = flatten l ++ flatten r
```

# Lecture 4: Higher-order functions

A higher-order function is a function that either takes a function as an argument or returns a function as a result.

Higher-order functions allow you to abstract over programming patterns.

**Examples.**

```
map :: (a -> b) -> [a] -> [b]
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

# Lecture 4: the `foldr` function

Many recursive functions on lists follow the following pattern:

```
f []     = v
f (x:xs) = x # f xs
```

The higher-order function `foldr` encapsulates this pattern. Instead of the above, we can simply write:

```
f = foldr (#) v
```

# Lecture 5: What is a type class?

A type class is a family of types that implement a common interface (= set of functions).

**Example:** `Eq` is the family of types that implement `(==)` and `(/=)`.

A type that belongs to this family is called an instance of the type class.

# Lecture 5: What is a type class?

A type class is a family of types that implement a common interface (= set of functions).

**Example:** `Eq` is the family of types that implement `(==)` and `(/=)`.

A type that belongs to this family is called an instance of the type class.

**Example:** `Int` is an instance of the `Eq` class.

Alert: Type classes have little in common with classes from OO languages.

# Lecture 5: Working with subclasses

Some type classes are a subclass of another class: each instance must also be an instance of the base class.

**Example:** `Ord` is a subclass of `Eq`:

```
class (Eq a) => Ord a where
  (<) :: a -> a -> a
  -- ...
```

# Lecture 5: The `Functor` type class

The function

```
map :: (a -> b) -> [a] -> [b]
```

applies a function to every element in a list.

`Functor` is a family of type constructors that have a `map`-like function, called `fmap`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

We often think of a functor as a container storing elements of some type `a`.

# Lecture 5: Applicative functors

`Applicative` is a subclass of `Functor` that adds two new operations `pure` and `(<*>)` (pronounced '*ap*' or '*zap*').

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

# Lecture 6: The `IO` type

`IO a` is the type of programs that interact with the world and return a value of type `a`.

An expression of type `IO a` is called an action.

Actions can be passed around and returned like any Haskell type, but they are not performed except in specific cases:

- `main :: IO ()` is performed when the whole program is executed.
- GHCi will also perform any action it is given.
- Other actions are only performed when called by another action.

# Lecture 6: The `Monad` class

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

**Examples.**

- Possible failure: `Maybe`
- Throwing exceptions: `Either`
- Reading and writing global state: `State`
- Non-determinism: `[]`
- Interacting with the world: `IO`

# Lecture 6: Some terminology on monads

A monad is a type constructor that is an instance of the `Monad` type class.

A monadic type is a type of the form `m a` where `m` is a monad.

An action is an expression of a monadic type.

A monadic function is a function that returns an action.

# Lecture 6: `do` notation

## With `do`-notation

```haskell
do
  x <- f
  g x
  y <- h x
  return (p x y)
```

## Without `do`-notation

```haskell
f >>= (\x ->
  g x >> (
    h x >>= (\y ->
      return (p x y)
    )
  )
)
```

# Lecture 7: Monadic parsing

```haskell
type Parser a =
  String -> [(a,String)]

item :: Parser Char
item (x:xs) = [(x,xs)]
item []     = []
```

- Parsing returns a *list* of possible parses
- Each parse comes with a ‘remainder’ of the string for further parsing

# Lecture 7: Type class laws

Most Haskell type classes have one or more laws that instances should satisfy.

**Laws for `Eq`:**

- Reflexivity: `x == x = True`
- Symmetry: `(x == y) = (y == x)`
- Transitivity: If `(x == y && y == z) = True` then `x == z = True`
- Substitutivity: If `x == y = True` then `f x == f y = True`
- Negation: `x /= y = not (x == y)`

# Lecture 7: The monad laws

- **Left identity:**

```
return x >>= f  =  f x
```

- **Right identity:**

```
mx >>= (\x -> return x)  =  mx
```

- **Associativity:**

```
(mx >>= f) >>= g
=
mx >>= (\x -> (f x >>= g))
```

# Lecture 8: Lazy evaluation

A redex is an expression where the top-level function call can be unfolded.

An evaluation strategy gives a general way to pick a redex to evaluate next.

- Call-by-value reduction: evaluate arguments before unfolding the definition of a function
- Call-by-name reduction: unfold function definition without evaluating arguments
- Lazy evaluation is call-by-name but avoiding double evaluation.

# Lecture 8: Pros & cons of lazy evaluation

**Advantages:**

- It never evaluates unused arguments.
- It always terminates if possible.
- It takes the minimal number of steps.
- It enables use of infinite data structures

**Pitfalls:**

- Thunks has some runtime overhead.
- Big intermediate expressions sometimes cause a drastic increase in memory usage.
- It becomes much harder to predict the order of evaluation.

# Lecture 8: Infinite data structures

An infinite data structure is an expression that would contain an infinite number of constructors if it is fully evaluated.

With infinite data structures, we can define what we want to compute (the data) independently of how it will be used (the control flow).

We can get the data we need for each situation by applying the right function to the infinite list: `take`, `!!`, `takeWhile`, `dropWhile`, …

# Lecture 9: Agda vs. Haskell

**Typing** uses a single colon:
$b$ : Bool instead of $b$ :: Bool.

**Naming** has fewer restrictions: any name can start with small or capital letter, and symbols can occur in names.

**Whitespace** is required more often: 1+1 is a valid function name, so you need to write 1 + 1 instead.

**Infix operators** are indicated by underscores: _+_ instead of (+)

# Lecture 9: Types as first-class values

In Agda, types such as Nat and (Bool $\rightarrow$ Bool) are themselves expressions of type Set.

We can define polymorphic functions as functions that take an argument of type Set:

$$
\begin{aligned}
&\mathsf{id} : (A : \mathsf{Set}) \rightarrow A \rightarrow A \\
&\mathsf{id}\ A\ x = x
\end{aligned}
$$

# Lecture 9: Total functional programming

Agda is a total language:

- **NO** runtime errors
- **NO** incomplete pattern matches
- **NO** non-terminating functions

So functions are true functions in the mathematical sense: evaluating a function call always returns a result in finite time.

# Lecture 10: Dependent types

A dependent type is a type that depends on a value of some base type.

With dependent types, we can specify the allowed inputs of a function more precisely, ruling out invalid inputs at compile time.

**Examples of dependent types.**

- $\mathsf{Food}\ f$, indexed over $f : \mathsf{Flavour}$
- $\mathsf{Vec}\ A\ n$, indexed over $n : \mathsf{Nat}$
- $\mathsf{Fin}\ n$, indexed over $n : \mathsf{Nat}$
- $\mathsf{Expr}\ t$, indexed over $t : \mathsf{Term}$

# Lecture 10: A safe lookup

$$
\begin{aligned}
&\mathsf{lookupVec} : \{A : \mathsf{Set}\}\ \{n : \mathsf{Nat}\} \\
&\qquad\qquad \to \mathsf{Vec}\ A\ n \to \mathsf{Fin}\ n \to A \\
&\mathsf{lookupVec}\ (x :: xs)\ \mathsf{zero} \quad\;\; = x \\
&\mathsf{lookupVec}\ (x :: xs)\ (\mathsf{suc}\ i) = \mathsf{lookupVec}\ xs\ i
\end{aligned}
$$

This is a safe and total version of the Haskell `(!!)` function, without having to change the return type in any way.

# Lecture 11: Curry-Howard correspondence

Haskell B. Curry

*We can interpret logical propositions ($A \wedge B$, $\neg A$, $A \Rightarrow B, \ldots$) as the types of all their possible proofs.*

**In particular:** A false proposition has no proofs, so it corresponds to an empty type.

# Lecture 11: the Curry-Howard correspondence

We interpret propositions as the types of their proofs:

| Propositional logic | | Type system |
|---|---|---|
| proposition | $P$ | type |
| proof of a proposition | $p : P$ | program of a type |
| conjunction | $P \times Q$ | pair type |
| disjunction | Either $P\ Q$ | either type |
| implication | $P \to Q$ | function type |
| truth | $\top$ | unit type |
| falsity | $\bot$ | empty type |
| universal quantification | $(x : A) \to P\ x$ | dependent function type |
| equality | $x \equiv y$ | identity type |

# Lecture 11: Induction in Agda

In general, a proof by induction in Agda looks like this:

$$\begin{aligned}
&\mathsf{proof} : (n : \mathsf{Nat}) \to \mathsf{P}\ n \\
&\mathsf{proof}\ \mathsf{zero} \quad\quad = \cdots \\
&\mathsf{proof}\ (\mathsf{suc}\ n) = \cdots
\end{aligned}$$

- proof zero is the base case
- proof (suc $n$) is the inductive case

When proving the inductive case, we can make use of the induction hypothesis proof $n$ : $P\ n$.

# Lecture 12: The identity type

The type identity type encodes the property of two elements of some type $A$ being equal:

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

- If $x$ and $y$ are equal, $x \equiv y$ has one constructor refl.
- If $x$ and $y$ are not equal, $x \equiv y$ is an empty type, so we can use an absurd pattern ().

# Lecture 12: Properties of equality

Symmetry:

$$\text{sym} : \{A : \text{Set}\}\ \{x\ y : A\} \to x \equiv y \to y \equiv x$$

Transitivity:

$$\text{trans} : \{A : \text{Set}\}\ \{x\ y\ z : A\}$$
$$\to x \equiv y \to y \equiv z \to x \equiv z$$

Congruence:

$$\text{cong} : \{A\ B : \text{Set}\}\ \{x\ y : A\}$$
$$\to (f : A \to B) \to x \equiv y \to f\ x \equiv f\ y$$

# Lecture 12: Equational reasoning in Agda

We can write down an equality proof in equational reasoning style in Agda:

- The proof starts with begin and ends with end.
- In between is a sequence of expressions separated by $=\langle\rangle$ or $=\langle$ *proof* $\rangle$, where each expression is equal to the previous one.

Unlike the proof on paper, here the typechecker of Agda guarantees that each step of the proof is correct!

# Exam organization

# What should I study?

You should study:

- The book
- The lecture notes (QuickCheck + Agda)
- The assignments on Weblab

See file `course-overview.pdf` on Brightspace for a detailed list of the learning objectives.

# What kind of questions can I expect?

1. Theory question about a FP concept
2. Programming assignment + QuickCheck
3. Implementing a data type or type class
4. Implementing and/or using a monad
5. Question on lazy evaluation and/or strictness
6. Question on dependent types and/or Curry-Howard
7. Question on equational reasoning

# What is ‘open book’?

The exam PCs will come with GHC, Agda, VS Code, and the Agda plugin for VS Code.[2]

You are allowed to bring anything you want on paper or USB stick (in text or pdf format):

- Exercises and solutions on Weblab
- Physical or digital version of the book
- Lecture notes
- Haskell documentation

[2] No Haskell plugin, sorry.

# Official course survey

https://evasys-survey.tudelft.nl/evasys/online.php?p=744SX

# Can't get enough?

There is much more to discover:

- CS4135 Software Verification
- CS4410 Category Theory for Programmers
- Agda Meeting in Delft[3] (10-16 May)
- Summer School on Advanced Functional Programming in Utrecht[4] (3-7 July)

---

[3] https://wiki.portal.chalmers.se/agda/Main/AIMXXXVI

[4] https://utrechtsummerschool.nl/courses/science/advanced-functional-programming-in-haskell

# What's next?

Exam: 15 April at 9:00-12:00

Project deadline: 23 April (submission via WebLab)

Finally: **Thank you for your enthusiasm and persistence, and good luck with the exam!**