

# **Introduction to the course and basics of functional programming**

Lecture 1 of CSE 3100  
Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

# Today's lecture

- Welcome to the course
- What is functional programming?
- Installing GHC
- Haskell basics

# About the course

---

# First, a bit about me

- I'm assistant professor in the Programming Languages group.
- I do research into dependent type systems and am part of the team working on the Agda programming language.
- I'm originally from Belgium, and have been in Delft for 4 years.
- I speak Dutch & English.



# Teaching team

- Jesper Cockx (lecturer)
- Ivar de Bruin (lab responsible)
- Bohdan Liesnikov (project responsible)
- Casper Dekeling (teaching assistant)
- Matej Havelka (teaching assistant)
- Anna Kalandadze (teaching assistant)
- Davis Kažemaks (teaching assistant)
- Duuk Niemandsverdriet (teaching assistant)
- Miloš Ristić (teaching assistant)
- Pallabi Sree Sarker (teaching assistant)
- Giulio Segalini (teaching assistant)

# Course materials and resources

- Book: **Programming in Haskell** by Graham Hutton (second edition)
- Brightspace: news + slides + additional lecture notes
- Weblab: weekly exercises + final project
- Queue: asking for help during labs
- Answers: asking questions (any time)
- Course email: `fp-cs-ewi@tudelft.nl`

# Lectures (Monday + Tuesday)

## Week 1-4: Functional programming in Haskell

- Week 1: Haskell basics
- Week 2: polymorphism and data types
- Week 3: type classes and functors
- Week 4: monads and lazy evaluation

## Week 6-7: Dependently typed programming in Agda

- Week 6: Agda and dependent types
- Week 7: theorem proving in Agda

## Week 8-9: Summary + working on the project

# Lab sessions (Monday + Wednesday)

Each week there will be *ungraded* programming exercises on Weblab

- Tests are run automatically
- You can submit as often as you want
- Ask questions via Queue or on Answers

See Brightspace for links and more info

# Assessment

Individual project (50% of final grade)

- Assignment in three parts (week 3, 4, and 5)
- Deadline for submission: **19th of April**
- Repair option in Q4, but grade is capped at **6.0/10.0**.

Final exam (50% of final grade)

- On-campus digital Weblab exam
- Allowed material: course book, Haskell documentation
- Resit in mid-Q4

# Giving feedback

This is the fourth run of this course, things have improved but are still not perfect.

If you have any feedback on the organization and/or contents of the course, please send a mail to [fp-cs-ewi@tudelft.nl](mailto:fp-cs-ewi@tudelft.nl).

I will do my very best to improve things based on your feedback!

# Response to feedback of last year

Feedback on the lectures, Weblab assignments, and the project was generally positive.

Main points for improvement:

- High workload for the project  
⇒ Updated description of requirements + test cases
- Not enough TAs in final weeks  
⇒ Ivar joined as extra support for coordinating TAs
- Poor support for Haskell/Agda on Weblab  
⇒ We are rolling out (experimental) LSP support for Haskell
- Many technical problems during exam  
⇒ We are working on a solution

# **Introduction to Functional Programming**

---

- What is functional programming?
- Why would you study functional programming?
- Have you done functional programming before? In what languages?
- What is a functional programming language?
- What is a *pure* functional programming language?

# What is functional programming?

In imperative programming, computation happens primarily by **updating values**:

```
sum = 0;  
for (int i = 1; i <= 10; i++) {  
    sum = sum + i;  
}
```

In functional programming, computation happens primarily by **applying functions**:

```
sum [1..10]
```

# Why study functional programming?

- Functional code is (often) shorter
- Functional code is (often) easier to understand
- Functional code is (often) easier to refactor
- It is (often) easier to reason about correctness of functional code
- Writing functional programs will teach you new ways to think about programming
- Functional programming languages are at the cutting edge of PL design

Also: functional programming is (or can be) **fun!**

# What is a functional language?

Functional programming provides a **toolbox** of techniques for writing functional programs:

- Lambda expressions
- Higher-order functions
- Algebraic datatypes
- Pattern matching
- Recursion
- Immutable data
- Pure functions
- Lazy evaluation
- Type classes
- Monads

A ‘functional programming language’ is any language that allows us to use these tools!

# What is a pure functional language?

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

# What is Haskell?

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

- **Static typing** means all types are checked at compile time.<sup>1</sup>
- **Lazy evaluation** means inputs are evaluated as needed.
- **Purity** means functions do not have side effects.

---

<sup>1</sup>Static typing ≠ explicit type annotations: Haskell can infer types automatically!

# What is Agda?

Agda is a **dependently typed**, **total functional** programming language and a **proof assistant**:

- **Dependent types** are types that depend on program expressions.
- **Totality** means functions are defined and terminating for all inputs.
- **Proof assistants** allow you to state and prove properties of your programs.

# The Glasgow Haskell Compiler

---

# The Glasgow Haskell Compiler

**GHC** (the Glasgow Haskell Compiler) is the most popular and modern Haskell compiler.

**GHCi** is the interactive mode of GHC where you can type-check and evaluate Haskell expressions.

**GHCup** is a tool for installing GHC and related tools.

**Installation:** See instructions on Brightspace.

# GHCI: interactive mode of GHC

Some useful commands for GHCI:

```
> 1 + 1      -- evaluate an expression  
2  
> :t True   -- get type of expression  
True :: Bool  
> :l MyFunctions.hs  -- load file  
[1 of 1] Compiling Main  
Ok, one module loaded.  
> :q        -- quit GHCI  
Leaving GHCI.
```

# Writing Haskell files

A Haskell file has extension .hs and consists of definitions of functions and types:

```
-- file MyFunctions.hs
double x = x + x
quadruple x = double (double x)

password = "123456"
checkPassword x = x == password
```

# Hello world in Haskell

A Haskell program is a script with a `main` function:

```
-- file HelloWorld.hs
main = putStrLn "Hello, world!"
```

We can compile `HelloWorld.hs` using GHC:

```
$ ghc HelloWorld.hs
[1 of 1] Compiling Main
Linking HelloWorld ...
$ ./HelloWorld
Hello, world!
```

# The GHC ecosystem

Apart from the GHC compiler itself, there are many other tools for writing Haskell code:

- **Hackage** is a repository of Haskell libraries.
- **Hoogle** is a powerful search engine for Hackage.
- **Cabal** and **Stack**<sup>2</sup> are two build tools for building Haskell projects.
- The **Haskell Language Server** (HLS) provides editor services.

---

<sup>2</sup>You will use Stack for the course project.

# A sneak peak at the power of Haskell

Some programs to demo (section 1.5 in the book):

- `sum`: Summing a list of numbers
- `qsort`: Sorting values
- `seqn`: Sequencing a list of actions

# Haskell syntax

---

# Function application in Haskell

Instead of  $f(x)$ , Haskell uses the syntax  $f\ x$ .

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

# Naming rules

- Names of **constructors** start with a capital letter (`True`, `False`, ...)
- Names of **functions** and **variables** start with a small letter (`not`, `length`, ...)
- Names of **concrete types** start with a capital letter (`Bool`, `Int`, ...)
- Names of **type variables** start with a small letter (`a`, `t`, ...)

**Reserved keywords** (`if`, `let`, `data`, `type`, `module`, ...) are not allowed as names.

# Haskell comments

```
-- This is a single-line comment
x = 42      -- It can appear in-line
-- Each line must start with --

{- This is a multi-line comment
It starts with {- and ends with -}
Comments can be {- nested -}
-}
```

# Conditional expressions

Conditionals can be expressed with

```
if ... then ... else ...
```

```
abs x = if x < 0 then -x else x
```

```
signum x =
  if x < 0 then -1 else
    if x == 0 then 0 else 1
```

- The `else` branch is required
- `if/then/else` can be nested

# Let bindings

We can give a name to a subexpression using a **let binding**:

```
cylinderSurfaceArea r h =  
  let sideArea = 2 * pi * r * h  
      topArea = pi * r ^2  
  in sideArea + 2 * topArea
```

# Anatomy of a Haskell function

```
doubleSmallNumber :: Int -> Int
doubleSmallNumber x =
    if isSmall x then 2*x else x
where
    isSmall x = x < 10
```

- Each definition can have an (optional) **type signature** `f :: ...`
- A definition consists of one or more **clauses** `f x = ...`
- Helper functions can be put in a **where block**

## where vs let

Two differences between `let` and `where`:

- `let` defines variables **before** they are used, `where` defines variables **after** they are used.
- `let ... in ...` can appear **anywhere** in an expression, while `where` is always attached to a clause.

Which one you use depends on the situation and your personal preference.

# The Layout Rule (1/3)

Unlike C or Java, Haskell is **layout-sensitive**:  
indentation of code matters!

Definitions at the same level of indentation  
belong to the same block:

```
a = b + c
```

**where**

```
  b = 1
```

```
  c = 2
```

```
d = a * 2
```

# The Layout Rule (2/3)

**Layout rule:** Within a block, all definitions must start at the exact same column.

Blocks start with one of the keywords `where`,  
`let`, `case`, or `do`.

# The Layout Rule (3/3)

## Bad:

```
a = b + c
```

**where**

```
b = 1
```

```
c = 2
```

## Also bad:

```
a = b + c
```

**where**

```
b = 1
```

```
c = 2
```

## Good:

```
a = b + c
```

**where**

```
b = 1
```

```
c = 2
```

## OK but discouraged:

```
a = b + c
```

**where** b = 1

```
c = 2
```

# Types in Haskell

---

# What is a type?

A **type** is a name for a collection of values.

**Example:** `Bool` has two values `True` and `False`.

Haskell syntax for typing: `True :: Bool`

You can write type annotations (almost) anywhere in your Haskell program.

# What is type inference?

In Haskell type annotations are **optional**:  
the compiler will try to infer the type if it is not given.

In GHCi, we can ask the type of an expression with ":t"

```
> :t True  
Bool
```

# Interpreting Haskell type errors (1/2)

```
> not 't'  
<interactive>:1:5: error:

- Couldn't match expected type 'Bool' with actual type 'Char'
- In the first argument of 'not', namely ''t'  
  In the expression: not 't'  
  In an equation for 'it': it = not 't'

```

Haskell tells us that `not` takes an argument of type `Bool`, but was given an argument of type `Char` instead.

# Interpreting Haskell type errors (2/2)

```
> 5 + True
<interactive>:2:1: error:
• No instance for (Num Bool) arising from
  a use of '+'
• In the expression: 5 + True
In an equation for 'it': it = 5 + True
```

Haskell tells us that `+` works on any type that implements the `Num` typeclass, but `Bool` (the type of `True`) does not.

When you get a type error, don't ignore it but try to **understand** what GHC is telling you!

# Basic Haskell types

- **Bool**: `True` and `False`
- **Int**: `0`, `42`, `-9000`, ... (64 bit integer)
- **Integer**: `9223372036854775808`, ... (arbitrary-size integers)
- **Float**: `1.23`, `Infinity`, `Nan`, ...
- **Double**: `1.0e40`, `-1.0e300` ...
- **Char**: `'a'`, `'Z'`, `'/'`, ...
- **String**: `"Hello, world!"`,  
`"line 1\nline 2"`

# Function types $a \rightarrow b$

## Examples:

`not :: Bool -> Bool`

`isDigit :: Char -> Bool`

In general,  $a \rightarrow b$  is the type of (pure) functions from  $a$  to  $b$ .

A function of type  $a \rightarrow b$  can be applied to an argument of type  $a$  to get a result of type  $b$ :

```
> isDigit '5'
```

**True**

# Tuple types

$(a, b)$  is the type of tuples  $(x, y)$  where  
 $x :: a$  and  $y :: b$ .

## Examples:

$(42, \text{True}) :: (\text{Int}, \text{Bool})$

$('x', 'y', 'z') :: (\text{Char}, \text{Char}, \text{Char})$

**Warning:**  $(a, b, c)$  is not the same as  
 $((a, b), c)$  or  $(a, (b, c))$

# Functions with several arguments

Two ways to define a function with 2 arguments:

```
add1 :: (Int, Int) -> Int
```

```
add2 :: Int -> (Int -> Int)
```

`add1` takes as argument a **tuple** `(x, y)` and returns an **Int**.

`add2` takes as argument an **Int** and returns a **function** of type `Int -> Int` that can be applied to the second argument.

# Working with curried functions

A function that returns another function is called a **curried** function, after Haskell B. Curry.

```
> :t add2 1  
Int -> Int  
> (add2 1) 1  
2
```



As a convention, Haskell functions with multiple arguments are usually curried.

# Syntax conventions

The function arrow `->` associates to the right:

```
Int -> Bool -> Char  
= Int -> (Bool -> Char)
```

Function application associates to the left:

```
f x y = (f x) y
```

# The list type [a]

[t] is the type of (singly-linked) lists

[x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>] with elements of type t

**Question.** What is the type of the following lists?

- [True, False] :: ???
- [1, 2, 3] :: ???
- [[1], [1, 2], [1, 2, 3]] :: ???
- ['a', 'b', 'c'] :: ???

# The list type [a]

[t] is the type of (singly-linked) lists

[x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>] with elements of type t

**Question.** What is the type of the following lists?

- [True, False] :: Bool
- [1, 2, 3] :: Num a => [a]
- [[1], [1, 2], [1, 2, 3]] ::  
Num a => [[a]]
- ['a', 'b', 'c'] :: String

# Strings as lists

The type of strings is defined as

```
type String = [Char].
```

We can use the two types interchangeably:

```
> ['a', 'b', 'c'] == "abc"
```

**True**

```
> ['a' .. 'z']
```

```
"abcdefghijklmnopqrstuvwxyz"
```

**Note.** The module **Data.Text** has a more efficient representation of strings.

# Quiz question

**Question.** Which one is a correct type of the list `[ "True", "False" ]`?

1. `String`
2. `[Bool]`
3. `[ [Char] ]`
4. `( [Char], [Char] )`

# Lists vs. tuples

Two differences between a list and a tuple:

- An element of type  $[t]$  can have any length, while the size of a tuple of type  $(t_1, t_2, \dots, t_n)$  is **fixed**.
- All elements in a list must have the **same type**, while the elements of a tuple can have different types.

# Ranges of values

The list of values from `i` to `j` is written  
`[i .. j]`.

Try to evaluate these ranges in GHCi:

- `[1..10]`
- `[1,3..10]`
- `[42..42]`
- `[10..1]`
- `[10,8..1]`

# Ranges of values

The list of values from `i` to `j` is written  
`[i .. j]`.

Try to evaluate these ranges in GHCi:

- `[1..10] = [1,2,3,4,5,6,7,8,9,10]`
- `[1,3..10] = [1,3,5,7,9]`
- `[42..42] = [42]`
- `[10..1] = []`
- `[10,8..1] = [10,8,6,4,2]`

# List comprehensions

---

# List comprehensions

We can construct new lists using a **list comprehension**:

```
[ x*x | x <- [1..5] ]
```

# List comprehensions

We can construct new lists using a **list comprehension**:

```
[ x*x | x <- [1..5] ]  
= [1, 4, 9, 16, 25]
```

# List comprehensions

We can construct new lists using a **list comprehension**:

```
[ x*x | x ← [1..5] ]  
= [1, 4, 9, 16, 25]
```

The part `x ← [1..5]` is called a **generator**.

(Compare with **set comprehensions** from mathematics:  $\{x^2 \mid x \in \{1\dots 5\}\}$ )

# Using multiple generators (1/2)

We can use more than one generator:

```
[ x*y | x<- [1,2,3], y<- [10,100] ]
```

```
[ x*y | y<- [10,100], x<- [1,2,3] ]
```

# Using multiple generators (1/2)

We can use more than one generator:

```
[ x*y | x<- [1,2,3], y<- [10,100] ]  
= [10,100,20,200,30,300]
```

```
[ x*y | y<- [10,100], x<- [1,2,3] ]  
= [10,20,30,100,200,300]
```

The order of the generators matters!

# Using multiple generators (2/2)

Later generators can depend on previous ones:

```
[ 10*x+y | x<- [1..3], y<- [x..3] ]
```

# Using multiple generators (2/2)

Later generators can depend on previous ones:

```
[ 10*x+y | x<- [1..3], y<- [x..3] ]  
= [11, 12, 13, 22, 23, 33]
```

# Filtering lists

We can select only elements that satisfy a boolean predicate:

```
[ x | x <- [1..10] , even x ]
```

# Filtering lists

We can select only elements that satisfy a boolean predicate:

```
[ x | x <- [1..10] , even x ]  
= [2, 4, 6, 8, 10]
```

The predicate `even x` is called a **guard**.

**Note:** a guard must be of type **Bool**.

# List comprehensions as a general control structure

Haskell does not have built-in control (such as `for` or `while`). Instead, we can use **lists** to define iterative algorithms:

**Imperative code:**      **Haskell equivalent:**

```
int result = 0;      result =
int i = 0;           sum [ i*i
while (i<=10) {       | i <- [1..10]
    result += i*i;
    i += 1;
}
```

# Two exercises

Using list comprehensions,...

- define

`addMod3Is2 :: [Int] -> [Int]` that keeps only values equal to 2 modulo 3 (2, 5, 8, etc.), and adds 3 to each.

`addMod3Is2 [2, 3, 4, 8] = [5, 11]`

- define `concat :: [[Int]] -> [Int]` that ‘flattens’ a list of lists into a single list.

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

# What's next?

Next lecture: Defining and testing functions

To do:

- Get the book
- Read the book:
  - Today: 1.1-1.5, 2.1-2.5, 3.1-3.6, 5.1-5.2
  - Next lecture: 3.7-3.9, 4.1-4.4, 6.1-6.6 + QuickCheck notes
- Install Stack and GHC
- Start with the exercises on WebLab

# Defining and testing functions

Lecture 2 of CSE 3100  
Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

# The many restrictions of Haskell

- No mutable variables
- No while loops / for loops / ...
- No try/catch blocks
- No side effects
- No objects
- Not even goto!

# The many restrictions of Haskell

- No mutable variables
- No while loops / for loops / ...
- No try/catch blocks
- No side effects
- No objects
- Not even goto!

How do we even write programs in such a language?



# Lecture plan

- Tools for writing functional programs
  - Pattern matching
  - Recursion
- Writing reusable functions:  
polymorphic types and type classes
- Testing functions:  
QuickCheck

# Pattern matching

---

# Pattern matching

We can define functions by case analysis using pattern matching:

```
not :: Bool -> Bool  
not True = False  
not False = True
```

Pattern matching is one of the most powerful and useful features of Haskell.

**You will have to use it a lot!**

# Pattern variables and wildcards

A **pattern variable** matches any value that didn't match the previous patterns:

```
rank 1 = "first"  
rank 2 = "second"  
rank 3 = "third"  
rank n = show n ++ "th"
```

A **wildcard** `_` is like a pattern value for which you don't care about the value:

```
isItTheAnswer 42 = True  
isItTheAnswer _ = False
```

# Matching on multiple arguments

```
xor :: Bool -> Bool -> Bool
xor True False = True
xor False True  = True
xor _         _  = False
```

# Side note: Order of clauses

```
f True _ = 1  
f _ True = 2  
f _ _ = 3
```

vs.

```
g _ True = 2  
g True _ = 1  
g _ _ = 3
```

Haskell will use the **first clause that matches**,

so `f True True = 1` but

`g True True = 2!`

# Side note: operator syntax

Infix operations such as `(+)`, `(-)`, `(==)`,  
`(!!)`, ... are just regular Haskell functions:

- Name must consist of special characters only
- Must be parentesized when appearing by themselves

They can also be used as normal functions:

```
(+) 1 1 == 1 + 1
```

# Three definitions of `(&&)`

Here are three definitions of the library function

```
(&&) :: Bool -> Bool -> Bool
```

# Three definitions of `(&&)`

Here are three definitions of the library

function `(&&)` :: Bool → Bool → Bool

-- version 1

```
True && True = True
True && False = False
False && True = False
False && False = False
```

-- version 2

```
True && True = True
_ && _ = False
```

-- version 3

```
True && b = b
False && _ = False
```

**Question:** Is there any difference in practice?

# Pattern matching on lists

```
isEmpty :: [a] -> Bool  
isEmpty []      = True  
isEmpty (x:xs)  = False
```

- Any list is either `[]` or `x:xs`
- `[1, 2, 3]` is syntactic sugar for  
`1:2:3:[ ]`

# Incomplete matches

```
takeTwo (x1:x2:xs) = (x1,x2)
```

```
> takeTwo [6]
*** Exception: Non-exhaustive
patterns in function takeTwo
```

**Tip.** Add `-Wincomplete`-patterns to the ghc-options of your .cabal project to get a warning for incomplete patterns!

# Using guards

We can use **guards** to add boolean conditions to a clause:

```
signum x
| x < 0      = -1
| x == 0     = 0
| otherwise   = 1
```

- Guards `| b` appear after the patterns (and before `=`)
- The condition `b` should be of type **Bool**
- `otherwise` is defined to be always **True**

# Mixing guards and pattern matching

```
capitalize :: String -> String
capitalize (c : cs)
| isLower c = (toUpper c) : cs
capitalize cs = cs
```

# Recursion

---

# Recursion example: factorial

```
fac :: Int -> Int
```

```
fac 0 = 1
```

```
fac n = n * fac (n-1)
```

Evaluating `fac` step by step:

$$\begin{aligned} \text{fac } 3 &= 3 * \text{fac } 2 \\ &= 3 * (2 * \text{fac } 1) \\ &= 3 * (2 * (1 * \text{fac } 0)) \\ &= 3 * (2 * (1 * 1)) = 6 \end{aligned}$$

**Question.** What happens if  $n < 0$ ?

# Recursion on lists

Recursion is not limited to numbers: we can also **recurse over structured data** such as lists:

```
product :: [Int] -> Int
```

```
product [] = 1
```

```
product (x:xs) = x * product xs
```

```
zip :: [Int] -> [Int] -> [(Int, Int)]
```

```
zip (x:xs) (y:ys) = (x, y) : (zip xs ys)
```

```
zip _ _ = []
```

# Why use recursion?

- Often the **most natural way** to write functional programs
- Recursion + list comprehensions completely **remove the need for traditional loops**
- We can prove properties of recursive functions by **induction**

# Implementing recursive functions

A 4-step plan for implementing a function:

1. Write down the type
2. Enumerate the cases
3. Define the base case(s)
4. Define the recursive case(s)

# Example: Implementing insertion sort

Step 1: write down the type

```
isort :: [Int] -> [Int]  
isort xs = _
```

# Example: Implementing insertion sort

Step 2: enumerate the cases

```
isort :: [Int] -> [Int]
isort []     = _
isort (x:xs) = _
```

# Example: Implementing insertion sort

Step 3: define base cases

```
isort :: [Int] -> [Int]
isort []      = []
isort (x:xs)  = _
```

# Example: Implementing insertion sort

Step 4: define recursive cases

```
isort :: [Int] -> [Int]
isort []      = []
isort (x:xs)  = insert x (isort xs)
```

# Example: Implementing insertion sort

Step 1: write down the type

```
isort :: [Int] -> [Int]
isort []      = []
isort (x:xs)  = insert x (isort xs)
```

**where**

```
insert :: Int -> [Int] -> [Int]
insert x ys = _
```

# Example: Implementing insertion sort

Step 2: enumerate the cases

```
isort :: [Int] -> [Int]
isort []      = []
isort (x:xs)  = insert x (isort xs)
```

**where**

```
insert :: Int -> [Int] -> [Int]
insert x []      = _
insert x (y:ys)
  | x <= y      = _
  | otherwise     = _
```

# Example: Implementing insertion sort

Step 3: define base cases

```
isort :: [Int] -> [Int]
isort []      = []
isort (x:xs)  = insert x (isort xs)
```

**where**

```
insert :: Int -> [Int] -> [Int]
insert x []      = [x]
insert x (y:ys)
  | x <= y      = _
  | otherwise     = _
```

# Example: Implementing insertion sort

Step 4: define recursive cases

```
isort :: [Int] -> [Int]
isort []      = []
isort (x:xs)  = insert x (isort xs)
```

**where**

```
insert :: Int -> [Int] -> [Int]
insert x []      = [x]
insert x (y:ys)
  | x <= y      = x:y:ys
  | otherwise     = y:(insert x ys)
```

# **Polymorphic types**

---

# Recap: Haskell Types

Typing annotations: `x :: a`

**Basic types** `Bool`, `Int`, `Integer`, `Float`,  
`Double`, `Char`, `String`, ...

**List type** `[a]`<sup>1</sup>

**Tuple types** `(a, b)`, `(a, b, c)`, ...

**Function types** `a -> b`, `a -> b -> c`, ...

---

<sup>1</sup> `String = [Char]`

# Question

What other types could be given to these functions?

- `length :: [Int] -> Int`
- `concat :: [[Int]] -> [Int]`
- `isSorted :: [Int] -> Bool`

# Polymorphic functions

Many functions can be given several types, for example `length :: [Int] -> Int,`  
`length :: [Bool] -> Int, ...`

These functions are given a **polymorphic** type:

`length :: [a] -> Int`

Unlike generics in Java/C#/..., we do not need to give the type since Haskell can infer it for us:

```
> length [2,3,5,7]
```

4

# Some polymorphic functions on tuples

`fst :: (a, b) -> a`

`snd :: (a, b) -> b`

`swap :: (a, b) -> (b, a)`

**Question.** Can you guess what they do from their types? Is there anything *else* that they could possibly do?

# Some polymorphic functions on lists

( : )	::	a	->	[a]	->	[a]		
head	::	[a]	->	a	--	<i>partial!</i>		
tail	::	[a]	->	[a]	--	<i>partial!</i>		
(++)	::	[a]	->	[a]	->	[a]		
( !! )	::	[a]	->	<b>Int</b>	->	a	--	<i>partial!</i>
take	::	<b>Int</b>	->	[a]	->	[a]		
drop	::	<b>Int</b>	->	[a]	->	[a]		
zip	::	[a]	->	[b]	->	[ (a, b) ]		
unzip	::	[ (a, b) ]	->	( [a], [b] )				

# Quiz question

**Question.** Which of the following equations is true for all Haskell lists `xs :: [a]` ?

1. `[] : xs == [ [], xs ]`
2. `xs : xs == [ xs, xs ]`
3. `[ [] ] ++ xs == xs`
4. `[ [] ] ++ [ xs ] == [ [], xs ]`

# Type classes

---

# Polymorphic functions with constraints

**Question:** What should be the type of

```
double x = x + x?
```

- `double :: Int -> Int` is too restrictive (it also works for `Float`!)
- `double :: a -> a` is too general (it doesn't work for `Bool`!)

# Polymorphic functions with constraints

**Question:** What should be the type of

```
double x = x + x?
```

- `double :: Int -> Int` is too restrictive (it also works for `Float`!)
- `double :: a -> a` is too general (it doesn't work for `Bool`!)

**Solution:** Add a constraint `Num a =>`:

```
double :: Num a => a -> a
```

# Type classes in Haskell

**Num** is an example of a **type class**: a collection of types that support a common interface.

```
class Num a where
    (+)      :: a -> a -> a
    (-)      :: a -> a -> a
    (*)      :: a -> a -> a
    negate   :: a -> a
    abs      :: a -> a
    fromInteger :: Integer -> a
```

**Question.** Is this the same as interfaces in Java?

# The **Eq** class

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

# The `Ord` class

```
class Eq a => Ord a where
  (<)    : a -> a -> Bool
  (≤)    : a -> a -> Bool
  (>)    : a -> a -> Bool
  (≥)    : a -> a -> Bool
  max   : a -> a -> a
  min   : a -> a -> a
```

`Ord` is an example of a `subclass`: any instance of `Ord` must also be an instance of `Eq`.

# Other useful type classes

**Show** is for printing values:

```
show 123 == "123"
```

**Read** is for parsing values:

```
read "123" == 123
```

**Integral** is for integral division:

```
9 `div` 2 == 4, 9 `mod` 2 == 1
```

**Fractional** is for floating-point division:

```
9.0 / 2.0 == 4.5
```

# Discussion

**Question.** Can we define a Haskell function

```
isSorted :: Ord a => [a] -> Bool
```

that checks if a given list is sorted *without* using pattern matching or list comprehensions, only using functions from the Prelude?

```
isSorted [1,3,6,10] == True
```

```
isSorted [5,6,1,3] == False
```

```
isSorted [] == True
```

# **Property-based testing with QuickCheck**

---

# Unit testing

Writing unit tests is **important** but also **boring** and **difficult**:

- *Boring* because you need a lot of unit tests
- *Difficult* because it is very easy to miss cases

What if we could generate test cases automatically?

# Unit testing

Writing unit tests is **important** but also **boring** and **difficult**:

- *Boring* because you need a lot of unit tests
- *Difficult* because it is very easy to miss cases

What if we could generate test cases automatically?

**Enter property-based testing.**

# Property-based testing

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

```
prop_abs_symmetric :: Int -> Int -> Bool  
prop_abs_symmetric x y =  
    abs (x - y) == abs (y - x)
```

```
prop_reverse_idempotent :: [Int] -> Bool  
prop_reverse_idempotent xs =  
    reverse (reverse xs) == xs
```

# Random testing of properties

To test a property, we can simply generate many inputs **randomly** and check if the property holds for all of them.

Randomly testing the property

`prop_abs_symmetric`:

```
abs (0.0 - 2.7)      == abs (2.7 - 0.0)
abs ((-0.7) - (-0.9)) == abs ((-0.9) - (-0.7))
abs (6.5 - (-4.0)) == abs ((-4.0) - 6.5)
abs (2.0 - 7.7)      == abs (7.7 - 2.0)
abs ((-19.0) - 2.8) == abs (2.8 - (-19.0))
```

...

# Advantages of property-based testing

- You spend **less time writing tests**: a single property replaces many tests
- You get **better coverage**: test lots of combinations you'd never try by hand
- You spend **less time on diagnosing errors**: failing tests can be minimized

# The QuickCheck library for Haskell

QuickCheck is a Haskell library for writing property-based tests.

It was introduced in 1999 by Koen Claessen and John Hughes.<sup>2</sup>

It has been ported to many other languages: C, C++, Java, JavaScript, Python, Scala, ...<sup>3</sup>

---

<sup>2</sup>K. Claessen and J. Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*

<sup>3</sup><https://en.wikipedia.org/wiki/QuickCheck>

# Installing QuickCheck

Install QuickCheck with Cabal:

```
> cabal install QuickCheck
```

Or in a Stack project, add the following to the list of dependencies in package.yaml:

- QuickCheck >= 2.14

# Basic usage of QuickCheck

```
import Test.QuickCheck

dist :: Int -> Int
dist x y = abs (x - y)

prop_dist_self :: Int -> Bool
prop_dist_self x = dist x x == 0

prop_dist_sym :: Int -> Int -> Bool
prop_dist_sym x y = dist x y == dist y x

prop_dist_pos :: Int -> Int -> Bool
prop_dist_pos x y = dist x y > 0
```

# Running QuickCheck from GHCI

```
> quickCheck prop_dist_self
+++ OK, passed 100 tests.
> quickCheck prop_dist_sym
+++ OK, passed 100 tests.
> quickCheck prop_dist_pos
*** Failed! Falsified (after 1 test):
0
0
```

# Running QuickCheck tests in batch

```
main = do
    quickCheck prop_dist_self
    quickCheck prop_dist_sym
    quickCheck prop_dist_pos
```

```
> runghc Distance.hs
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
*** Failed! Falsified (after 1 test):
0
0
```

# Anatomy of a QuickCheck test

```
prop_isort_isSorted :: [Int] -> Bool
prop_isort_isSorted xs =
    isSorted (isort xs)
```

- Name starts with `prop_`  
(convention, but required on Weblab)
- Type of argument must be one for which  
we can generate arbitrary values
- Return type must be `Bool` or `Property`

# Finding minimal counterexamples

If QuickCheck finds a counterexample, it will apply **shrinking** to find a counterexample that is as small as possible.

```
prop_all_sorted :: [Int] -> Bool  
prop_all_sorted xs = isSorted xs
```

Running `quickCheck prop_all_sorted` will always return either `[1, 0]` or `[0, -1]`.

# Shrinking inputs

QuickCheck determines how to shrink a counterexample based on its type:

- `Int`: try number closer to 0
- `Bool`: try `False` instead of `True`
- `(a, b)`: shrink one of the components
- `[a]`: either shrink one value in the list, or delete a random element

# What's next?

Next lecture: Algebraic data types

To do:

- Read the book:
  - Today: 3.7-3.9, 4.1-4.4, 6.1-6.6, QuickCheck notes
  - Next lecture: 8.1-8.4, 8.6
- Finish week 1 exercises on WebLab
- Ask & answer questions on TU Delft Answers

# Data types

Lecture 3 of CSE 3100  
Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

# Lecture plan

- More about QuickCheck
- Type aliases and `newtype` declarations
- Algebraic data types
- Parametrized data types

# More about QuickCheck

---

# Property-based testing with QuickCheck

A QuickCheck property is a function that returns a **Bool**:

```
prop_isort :: [Int] -> Bool  
prop_isort xs = isSorted (isort xs)
```

QuickCheck will:

- generate inputs until property is **False**
- shrink the counterexample as far as it can

**Warning.** QuickCheck sets type variables to `()`, so **avoid polymorphic properties**.

# Side note: testing polymorphic properties

QuickCheck will instantiate all polymorphic types with `()` (the *empty tuple*), which is usually not what we want:

```
prop_isort_isSorted_bad ::  
  (Ord a) => [a] -> Bool  
prop_isort_isSorted_bad xs =  
  isSorted (isort xs)  
-- ^ will test if `isort [(), ..., ()]`  
  is sorted, which is always true.
```

# Three common kinds of QuickCheck tests

**Roundtrip properties.** For example:

- `reverse (reverse xs) == xs`
- `del x (ins x xs) == xs`

**Equivalent implementations.** For example:

- `isort xs == qsort xs`

**Algebraic properties.** For example:

- `0 + x == x + 0`
- `x + (y + z) == (x + y) + z`
- `x + y == y + x`

# Quiz question

Consider the following function:

```
intersect :: Eq a => [a] -> [a] -> [a]
intersect xs ys = [ x | x <- xs, x `elem` ys ]
```

**Question.** Which of these properties is NOT satisfied?

1. `intersect xs ys == intersect ys xs`
2. `intersect [] xs == []`
3. `intersect (intersect xs ys) zs == intersect xs (intersect ys zs)`
4. `intersect xs xs == xs`

# Properties with a limited domain

```
-- replicate n x produces the list
-- [x, x, . . . , x] (with n copies of x)
prop_replicate n x i =
    replicate n x !! i == x
```

```
> quickCheck prop_replicate
*** Failed! Exception:
'Prelude.!!: index too large'
```

# Solution #1: silencing invalid tests

```
prop_replicate n x i =  
    i < 0 || i >= n ||  
    replicate n x !! i == x
```

```
> quickCheck prop_replicate  
+++ OK, passed 100 tests.
```

**Problem:** This gives a false sense of security: index is out of bounds in almost all tests.

## Solution #2: adding preconditions

```
prop_replicate n x i =  
  (i >= 0 && i < n) ==>  
  replicate n x !! i == x
```

```
> quickCheck prop_replicate  
+++ OK, passed 100 tests;  
    695 discarded.
```

... ==> ... is a **conditional property**: test cases that do not satisfy the condition are *discarded*.

## Solution #3: using a custom generator

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

```
> quickCheck prop_replicate  
+++ OK, passed 100 tests.
```

`chooseInt (0, n-1)` is an example of a **generator**<sup>1</sup>: an object that can be used to generate random values of type `Int`.

---

<sup>1</sup>More generators can be found in the module `Test.QuickCheck`

# Live coding

**Exercise.** Write a test case for Luhn's algorithm  
(see Weblab exercises for this week).

- Test that `luhn :: [Int] -> Bool` has same output as  
`luhnSpec :: [Int] -> Bool`
- Length should be at least 1
- All numbers should be between 0 and 9

# Type aliases and newtype declarations

---

# Type aliases

A **type alias** gives a new name to an existing type:

```
type String = [Char]
```

```
type Coordinate = (Int, Int)
```

They can be used to convey **meaning**, but are treated transparently by the compiler.

# More examples of type aliases

```
-- Two parametrized types
type Pair a = (a , a)
type Assoc k v = [ (k , v) ]

-- An alias for a function type
type Transformation =
Coordinate -> Coordinate
```

**Warning:** type aliases cannot be recursive:

```
type Tree = (Int, Tree, Tree)
```

Cycle in type synonym declarations:

```
type Tree = (Int, Tree, Tree)
```

# **newtype** declarations

A **newtype** declaration is a specialized kind of **data** declaration with exactly one constructor taking exactly one argument:

```
newtype EuroPrice    = EuroCents    Integer
newtype DollarPrice = DollarCents Integer

dollarToEuro :: DollarPrice -> EuroPrice
dollarToEuro (DollarCents x) =
    EuroCents (round (0.93 * fromInteger x))
```

# `newtype` vs `type` vs `data`

Differences of `newtype` compared to `type`:

- Cannot accidentally mix up two types
- Need to wrap/unwrap elements by hand

Differences of `newtype` compared to `data`:

- Only one constructor with one argument
- More efficient representation
- No recursive types

# Algebraic datatypes (ADTs)

---

# A simple algebraic datatype

```
data Answer = Yes | No | DontKnow  
deriving (Show)
```

```
answers :: [Answer]
```

```
answers = [Yes, No, DontKnow]
```

```
flip :: Answer -> Answer
```

```
flip Yes      = No
```

```
flip No       = Yes
```

```
flip DontKnow = DontKnow
```

# The `Bool` type

**Question.** How to define `Bool`?

# The **Bool** type

**Question.** How to define **Bool**?

**Answer.**

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

# The `Ordering` type

The Prelude defines the following:

```
data Ordering = LT | EQ | GT
```

```
compare :: Ord a =>  
          a -> a -> Ordering
```

`compare` returns `LT`, `EQ`, or `GT` depending on whether the first argument is smaller, equal or greater than the second.

# Constructors arguments

```
data Shape = Circle Double  
           | Rect Double Double
```

```
square :: Double -> Shape
```

```
square x = Rect x x
```

```
area :: Shape -> Double
```

```
area (Circle r) = pi * r * r
```

```
area (Rect l h) = l * h
```

# Constructors as functions

Each constructor defines a **function** into the datatype:

```
> :t Circle
```

```
Circle :: Double -> Shape
```

```
> :t Rect
```

```
Rect :: Double -> Double -> Shape
```

# Record syntax

---

# Record syntax (1/3)

Haskell provides an alternative **record syntax** to define constructors with arguments:

```
data Shape
  = Circle { radius :: Double }
  | Rect    { width   :: Double
             , height  :: Double }
```

This is syntactic sugar for the previous definition but also defines functions `radius`, `width`, and `height`.

## Record syntax (2/3)

Each field also defines a **function** from the datatype:

```
radius :: Shape -> Double  
radius (Circle r) = r
```

**Warning.** Fields such as `radius` and `width` are **partial functions**: they raise a runtime error when applied to the wrong constructor.

# Record syntax (3/3)

We can also use record syntax when applying or matching on a constructor:

```
square :: Double -> Shape
square x = Rect { width = x }
```

```
getWidth :: Shape -> Double
getWidth (Circle{ radius = r }) = 2*r
getWidth (Rect { width = w }) = w
```

# Functional style vs. OO style

## Haskell

```
data Shape
  = Circle { radius :: Double }
  | Rect    { width  :: Double
             , height :: Double
             }

square :: Double -> Shape
square x = Rect x x

area :: Shape -> Double
area (Circle r) = pi * r * r
area (Rect w h) = w * h
```

## Java

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    double r;
    Circle(double radius) { r = radius; }
    double area() { return Math.PI*r*r; }
}

class Rectangle extends Shape {
    double w;
    double h;
    Rectangle(double width, double height) {
        w = width; h = height;
    }
    Rectangle(double side) {
        w = side; h = side;
    }
    double area() { return w*h; }
}
```

# The expression problem

In an **object-oriented** language, it is *easy* to add new cases to a type but *hard* to add new functions.

In a **functional** language it is *easy* to add new functions to a type but *hard* to add new cases.

This tradeoff is known as the **expression problem**.<sup>2</sup>

---

<sup>2</sup>John Reynolds (1975): *User-defined types and procedural data as complementary approaches to data abstraction*

# A recursive type: unary natural numbers

We can define a type `Nat` represents natural numbers (inefficiently) as `Zero`, `Suc Zero`, `Suc (Suc Zero)`, ...:

```
data Nat = Zero | Suc Nat

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n
| n > 0 = Suc (int2nat (n-1))
```

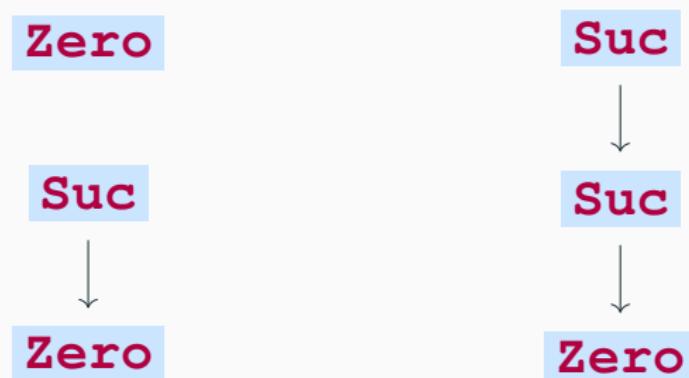
**Exercise.** Define

```
maximum :: Nat -> Nat -> Nat
```

# Drawing elements of HsNat

```
data Nat = Zero | Suc Nat
```

Three values of **Nat**:



# Parametrized datatypes

---

# The Haskell type **Maybe**

The type **Maybe** `a` represents an **optional** value of type `a`:

```
data Maybe a = Nothing  
              | Just a
```

**Maybe** is often used to represent **functions** that can fail:

```
safeDiv :: Int -> Int -> Maybe Int  
safeDiv x y  
| y == 0      = Nothing  
| otherwise   = Just (x `div` y)
```

# A safer `head` function

```
safeHead :: [a] -> Maybe a
safeHead []      = Nothing
safeHead (x:xs) = Just x
```

# Non-empty lists

The type `NonEmpty a` represents lists with at least one element:

```
data NonEmpty a = a :| [a]
```

```
toList :: NonEmpty a -> [a]
```

```
toList (x :| xs) = x : xs
```

# A safer **head** function

```
-- version using Maybe
safeHead :: [a] -> Maybe a
safeHead []      = Nothing
safeHead (x:xs) = Just x

-- version using NonEmpty
safeHead' :: NonEmpty a -> a
safeHead' (x :| xs) = x
```

**Question.** Which version is better in what situation?

# The **Either** type

The **Either** type represent a **disjoint union** of **a** and **b**: each element is either **Left**  $x$  for  $x :: a$  or **Right**  $y$  for  $y :: b$

```
data Either a b = Left a  
                | Right b
```

**Convention.** **Right** is often used to represent a successful operation, while **Left** is often used to represent an error.

# A poor man's exceptions

```
get :: Int -> [a] -> Either String a
get i xs
| i < 0          = Left "Negative index!"
| i >= length xs = Left "Index too large!"
| otherwise       = Right (xs !! i)

getTwo :: (Int, Int) -> [a] ->
          Either String (a, a)
getTwo (i, j) xs =
  case (get i xs) of
    Left err1 -> Left err1
    Right x     ->
      case (get j xs) of
        Left err2 -> Left err2
        Right y    -> Right (x, y)
```

# Counting the elements of a type

How many elements are in the following types:<sup>3</sup>

- **Either Bool Answer**
- **(Bool, Bool, Answer)**
- **Maybe (Bool, Bool)**

---

<sup>3</sup>Not counting any terms with `undefined`.

# Counting the elements of a type

How many elements are in the following types:<sup>3</sup>

- **Either Bool Answer**  $2 + 3 = 5$
- **(Bool, Bool, Answer)**
- **Maybe (Bool, Bool)**

---

<sup>3</sup>Not counting any terms with `undefined`.

# Counting the elements of a type

How many elements are in the following types:<sup>3</sup>

- **Either Bool Answer**  $2 + 3 = 5$
- **(Bool, Bool, Answer)**  $2 \times 2 \times 3 = 12$
- **Maybe (Bool, Bool)**

---

<sup>3</sup>Not counting any terms with `undefined`.

# Counting the elements of a type

How many elements are in the following types:<sup>3</sup>

- **Either Bool Answer**  $2 + 3 = 5$
- **(Bool, Bool, Answer)**  $2 \times 2 \times 3 = 12$
- **Maybe (Bool, Bool)**  $1 + (2 \times 2) = 5$

---

<sup>3</sup>Not counting any terms with `undefined`.

# Counting functions

How many possible functions of type  
**Bool → Answer** are there?

# Counting functions

How many possible functions of type  
**Bool → Answer** are there?

- `\b → if b then Yes else Yes`
- `\b → if b then Yes else No`
- `\b → if b then Yes else Unknown`
- `\b → if b then No else Yes`
- `\b → if b then No else No`
- `\b → if b then No else Unknown`
- `\b → if b then Unknown else Yes`
- `\b → if b then Unknown else No`
- `\b → if b then Unknown else Unknown`

# What's algebraic about ADTs?

An **algebraic datatype** is a type that is formed from other types using *sums* and *products*:

- The product of  $a$  and  $b$  is the **tuple type**  
 $(a, b)$
- The sum of  $a$  and  $b$  is the **disjoint union type** **Either**  $a$   $b$

Each constructor of an ADT is the *product* of the types of its arguments, and the ADT itself is the *sum* of the constructor types.

# A question for discussion

Suppose a fellow student says the following:

*There is no need for datatypes other than **Either**. For example, **Shape** can simply be defined as*

```
type Shape =  
    Either Double (Double, Double)  
circle x = Left x  
rect x y = Right (x, y)
```

Do you agree with this statement? Why (not)?

# Defining lists

**Question.** How would you define the list type  
[a] as a datatype?

# Defining lists

**Question.** How would you define the list type  
[a] as a datatype?

**Answer.**

```
data List a = Nil | Cons a (List a)
```

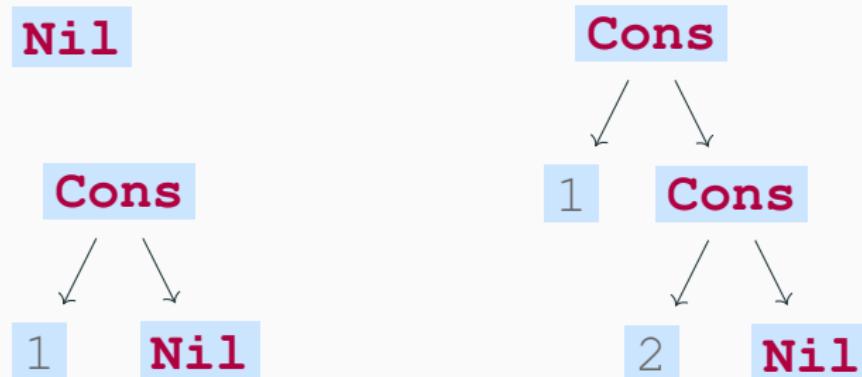
-- Closer but not valid syntax:

```
-- data [a] = [] | (:) a [a]
```

# Drawing elements of List

```
data List a = Nil  
           | Cons a (List a)
```

Three values of **List Nat**:



# Example: Binary trees

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

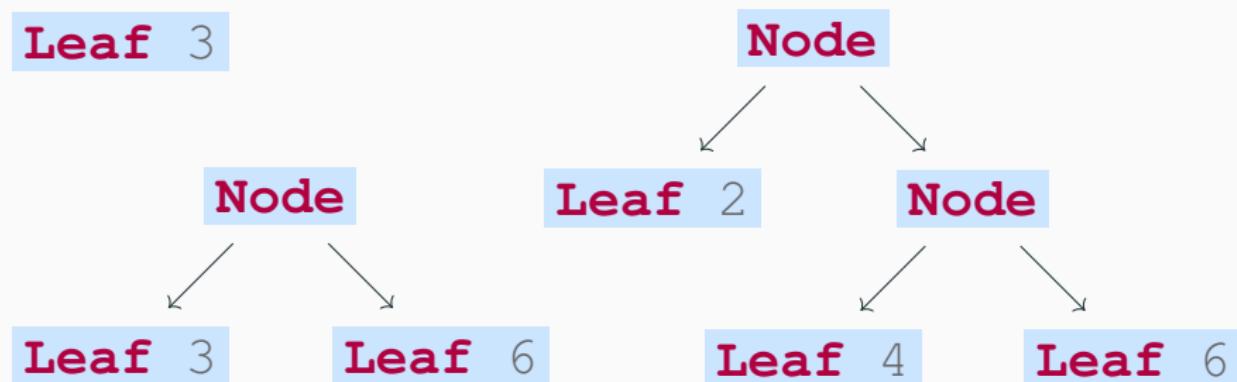
occurs :: Eq a => a -> Tree a -> Bool
occurs x (Leaf y)    = x == y
occurs x (Node l r) = occurs x l || occurs x r

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

# Drawing elements of Tree

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

Three values of **Tree Int**:



# Live coding: Tautology checker

**Assignment:** Implement a tautology checker for boolean expressions.

- Define type `Prop` of boolean expressions
- Define evaluation of expressions
- Define `pretty :: Prop -> String`  
and  
`parse :: String -> Maybe Prop`
- Define  
`isTautology :: Prop -> Bool`

# A brain teaser

**Question.** Can you construct an element of the following type?

```
data B a = C (B a -> a)
```

(not **error** or **undefined**)

# What's next?

Next lecture: Higher-order functions

To do:

- Read the book:
  - Today: 8.1-8.4, 8.6, QuickCheck lecture notes
  - Next lecture: 3.7-3.9, 4.5-4.6, 7.1-7.5
- Start on week 2 exercises on Weblab

# Higher-order functions

Lecture 4 of CSE 3100  
Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

# Lecture plan

- Higher-order functions
- `map`, `filter`, and other functions on lists
- The functions `foldr` and `foldl`

# Higher-order functions

---

# The DRY principle of programming

## DRY: Don't Repeat Yourself

*Every piece of knowledge must have  
a single, unambiguous, authoritative  
representation within a system<sup>1</sup>*

Higher-order functions are the ultimate expression of DRY, as they allow you to abstract over programming patterns.

---

<sup>1</sup>from *The pragmatic programmer* by Hunt & Thomas

# 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.

The latter is also called a curried function, so the term is mainly used for the former.

# Example of a higher-order function

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

```
> twice (\x -> x*x) 3
12
```

```
> twice reverse [1,2,3]
[1,2,3]
```

# Quiz question

**Question.** A function of type

`(Bool -> Int) -> Int`

1. ...takes a function as its input, which returns an integer as its output
2. ...takes a function as its input, which returns a boolean as its output
3. ...returns a function as its output, which takes a boolean as its input
4. ...returns a function as its output, which takes an integer as its input

# Higher-order functions: `curry`, `uncurry`, and `flip`

```
> :t curry
curry :: ((a , b) -> c) -> (a -> b -> c)
> :t uncurry
uncurry :: (a -> b -> c) -> ((a , b) -> c)
> :t flip
flip :: (a -> b -> c) -> (b -> a -> c)
> map (uncurry (+)) [(1,2),(3,4),(5,6)]
[3,7,11]
```

# Higher-order function: `(\$)`

```
> :t  (\$)  
(\$)  ::  (a -> b) -> a -> b
```

**Question.** Why would you ever want to use this?

# Higher-order functions on lists

---

# Higher-order function: `map`

```
> :t map
map :: (a -> b) -> [a] -> [b]
> map (\x -> x+1) [1,3,5,7]
[2,4,6,8]
```

`map f xs` corresponds to the `list comprehension` `[ f x | x <- xs ]`

# Quiz question

**Question.** Which of these equations does NOT hold for all `f :: Int -> Int` and `xs :: [Int]` ?

1. `map f (take n xs) == take n (map f xs)`
2. `map f (drop n xs) == drop n (map f xs)`
3. `map f (reverse xs) == reverse (map f xs)`
4. `map f (sort xs) == sort (map f xs)`

# Higher-order function: `filter`

```
> :t filter
filter :: (a -> Bool) -> [a] -> [a]
> filter even [1..8]
[2,4,6,8]
```

`filter p xs` corresponds to the `list`

`comprehension` `[ x | x <- xs , p x ]`

# Using `map` / `filter` with lambdas

```
> let f x = x*2+1 in map f [1..5]
[3,5,7,9,11]
```

```
> map (\x -> x*2+1) [1..5]
[3,5,7,9,11]
```

```
> let p x = x `mod` 3 == 0
  in filter p [1..10]
[3,6,9]
```

```
> filter (\x -> x `mod` 3 == 0) [1..10]
[3,6,9]
```

# Operator sections

An **operator section** is an operator that has been partially applied:

(+1) is shorthand for  $\lambda x \rightarrow x + 1$ .

```
> :t (+1)
(+1) :: Num a => a -> a
> map (+1) [1..5]
[2,3,4,5,6]
> filter (>5) [1..10]
[6,7,8,9,10]
```

# Three ways to write a program

-- using list comprehension

```
result = [ f x | x <- xs , p x ]
```

-- using pattern matching + recursion

```
result = aux xs
```

**where**

```
aux [] = []
```

```
aux (x:xs) | p x       = f x : aux xs
```

```
          | otherwise = aux xs
```

-- using higher-order functions

```
result = map f (filter p xs)
```

# Live programming problem

Implement the following functions using higher-order functions:

- `applyFuncs :: [a -> b] -> [a] -> [b]`
- `intersect :: Eq a => [a] -> [a] -> [a]`
- `allPairs :: [a] -> [b] -> [(a,b)]`

# Higher-order functions: `all` and `any`

```
> :t all
all :: Foldable t =>
    (a -> Bool) -> t a -> Bool
> :t +d all
all :: (a -> Bool) -> [a] -> Bool
> :t +d any
any :: (a -> Bool) -> [a] -> Bool
> import Data.Char (isSpace)
> any isSpace "Hello, world!"
True
```

# Higher-order functions: `takeWhile` and `dropWhile`

```
> :t takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
> :t dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]
> dropWhile isSpace "Hello, world!"
"Hello, world!"
```

# Testing properties of functions

---

# Higher-order properties

A **higher-order property** is a property that takes a **function** as an input:

```
prop_mapTwice
  :: (Int -> Int) -> [Int] -> Bool
prop_mapTwice f xs =
  map f (map f xs) == map (twice f) xs

-- prop> prop_mapTwice
-- No instance for (Show (Int -> Int))
```

QuickCheck tests properties by generating **random inputs**, but it cannot generate random functions.

# Higher-order properties

A **higher-order property** is a property that takes a **function** as an input:

```
prop_mapTwice
  :: (Int -> Int) -> [Int] -> Bool
prop_mapTwice f xs =
  map f (map f xs) == map (twice f) xs

-- prop> prop_mapTwice
-- No instance for (Show (Int -> Int))
```

QuickCheck tests properties by generating **random inputs**, but it cannot generate random functions. *Or can it?*

# Higher-order properties

QuickCheck provides the type `Fun a b` of shrinkable and printable functions.

```
prop_mapTwice
  :: Fun Int Int -> [Int] -> Bool
prop_mapTwice (Fn f) xs =
  map f (map f xs) == map (twice f) xs

-- prop> prop_mapTwice
-- +++ OK, passed 100 tests.
```

# Generating functions with QuickCheck

```
prop_bananas :: Fun String Int -> Bool
prop_bananas (Fn f) =
    f "banana" == f "monkey" ||
    f "banana" == f "elephant" ||
    f "monkey" == f "elephant"
```

```
-- prop> prop_bananas
-- *** Failed! Falsified
-- (after 5 tests and 163 shrinks):
-- { "banana"->2, "elephant"->0, _->1 }
```

# **Identity and function composition**

---

# Function composition

`(.) :: (b -> c) -> (a -> b) -> (a -> c)`  
`f . g = \x -> f (g x)`

Examples of how to use `(.)`:

`odd`         $\equiv$  `not . even`

`twice f`      $\equiv$  `f . f`

`sumsqeven`  $\equiv$  `sum`

`. map (^2)`

`. filter even`

# Building pipelines with `(.)`

We can use `(.)` to compose functions *without naming their arguments*:

```
processData = combine  
            . map process  
            . filter isValid
```

**where**

```
isValid = ...  
process = ...  
combine = ...
```

# The identity function

Haskell's boringest function:

```
id :: a -> a
```

```
id x = x
```

One possible use case:

```
compose :: [a -> a] -> a -> a
```

```
compose [] = id
```

```
compose (f:fs) = f . compose fs
```

**Question.** How does Haskell compute

```
compose [(-3), (*2), (+5)] 3?
```

# The three laws of function composition

$$\text{id} \circ f = f$$

$$f \circ \text{id} = f$$

$$f \circ (g \circ h) = (f \circ g) \circ h$$

This means Haskell functions form a **category**<sup>2</sup>

---

<sup>2</sup>See course CS4410 Category Theory for Programmers in CS master

# The higher-order function `foldr`

---

# A common pattern of recursion

sum [] = 0

sum (x:xs) = x + sum xs

product [] = 1

product (x:xs) = x \* product xs

or [] = **False**

or (b:bs) = b || or bs

and [] = **True**

and (b:bs) = b && and bs

**Don't Repeat  
Yourself!**

# A common pattern of recursion

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:

$$f = \text{foldr} \ (\#) \ v$$

# Examples of using `foldr`

sum        = foldr (+) 0

product = foldr (\*) 1

or        = foldr (||) **False**

and        = foldr (&&) **True**

# What does `foldr` do?

**Intuition:** `foldr (#) v` replaces each occurrence of `(::)` by `(#)` and the final `[]` by `v`:

$$\begin{aligned} & \text{foldr } (+) \ 0 \ [x_1, x_2, x_3] \\ = & \text{foldr } (+) \ 0 \ (x_1 : (x_2 : (x_3 : []))) \\ = & x_1 + (x_2 + (x_3 + 0)) \end{aligned}$$

Note that parentheses are associated to the right, hence the **r** in `foldr`.

# Recursive Definition of `foldr`

```
foldr :: (a -> b -> b) -> b  
        -> [a] -> b  
  
foldr (#) v []      = v  
foldr (#) v (x:xs) =  
    x # (foldr (#) v xs)
```

# Folding binary trees

For any recursive datatype we can define a **folding function** (not just for lists!)

```
foldt :: (a -> b) -> (b -> b -> b) ->
          Tree a -> b
foldt w f (Leaf x)    = w x
foldt w f (Node l r) = f (foldt l) (foldt r)

occurs x = foldt (\y -> x == y) (||)
flatten   = foldt (\x -> [x])      (++)
```

`foldt w f` replaces each **Leaf** by `w` and each **Node** by `f`.

# Folds in other languages

Folding functions exist in many other languages:

- Haskell: `foldr (+) 0 seq`
- Scala: `seq.fold(0)((a,b) => a + b)`
- Python: `reduce(lambda a,b:a+b, seq, 0)`
- Ruby: `seq.inject(0) {|a,b| a + b}`
- C#: `seq.Aggregate(func: (a,b) => a + b)`
- ...

# More suspicious patterns (1/5)

```
length []      = 0
```

```
length (x:xs) = 1 + length xs
```

# More suspicious patterns (1/5)

```
length []      = 0
```

```
length (x:xs) = 1 + length xs
```

```
length []      = 0
```

```
length (x:xs) =
```

```
(\_ n -> 1 + n) x (length xs)
```

# More suspicious patterns (1/5)

```
length []      = 0
```

```
length (x:xs) = 1 + length xs
```

```
length []      = 0
```

```
length (x:xs) =
```

```
    (\_ n -> 1 + n) x (length xs)
```

```
length = foldr (\_ n -> 1+n) 0
```

**Challenge.** Implement `length` in **point-free style** (without using pattern matching, recursion, or lambdas).<sup>3</sup>

<sup>3</sup>The function `const` might be useful.

## More suspicious patterns (2/5)

```
[ ]      ++ ys = ys  
(x:xs) ++ ys = x : (xs ++ ys)
```

## More suspicious patterns (2/5)

[ ]        ++ ys = ys

(x:xs) ++ ys = x : (xs ++ ys)

[ ]        ++ ys = ys

(x:xs) ++ ys = (:) x (xs ++ ys)

## More suspicious patterns (2/5)

[ ] ++ ys = ys

(x:xs) ++ ys = x : (xs ++ ys)

[ ] ++ ys = ys

(x:xs) ++ ys = (:) x (xs ++ ys)

xs ++ ys = foldr (:) ys xs

**Challenge.** Implement `(++)` in point-free style.

## More suspicious patterns (3/5)

map f [] = []

map f (x:xs) = f x : map f xs

## More suspicious patterns (3/5)

map f [] = []

map f (x:xs) = f x : map f xs

map f [] = []

map f (x:xs) =

(\x ys -> f x : ys) x (map f xs)

## More suspicious patterns (3/5)

map f [] = []

map f (x:xs) = f x : map f xs

map f [] = []

map f (x:xs) =

(\x ys -> f x : ys) x (map f xs)

map f = foldr (\x ys -> f x : ys) []

**Challenge.** Implement `map` in point-free style.<sup>4</sup>

---

<sup>4</sup>**Warning:** pretty hard.

## More suspicious patterns (4/5)

```
reverse []      = []
```

```
reverse (x:xs) = reverse xs ++ [x]
```

## More suspicious patterns (4/5)

```
reverse []      = []
```

```
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse []      = []
```

```
reverse (x:xs) =
```

```
(\x xs -> xs ++ [x]) x (reverse xs)
```

## More suspicious patterns (4/5)

```
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]

reverse []      = []
reverse (x:xs) =
  (\x xs -> xs ++ [x]) x (reverse xs)

reverse = foldr (\x xs -> xs ++ [x]) []
```

**Challenge.** Implement `reverse` in point-free style.<sup>5</sup>

---

<sup>5</sup>**Warning:** very hard, you might need the function `pure`.

# More suspicious patterns (5/5)

```
filter p []      = []
filter p (x:xs) =
  if  p x
  then x : filter p xs
  else filter p xs
```

# More suspicious patterns (5/5)

```
filter p []      = []
filter p (x:xs) =
    if p x
    then x : filter p xs
    else filter p xs

filter p = foldr (\x xs' -> if p x
                    then x : xs'
                    else xs') []
```

**Challenge.** Implement `filter` in point-free style.<sup>6</sup>

---

<sup>6</sup>**Warning:** *really* hard, you might need `bool` and `(&&&)`.

# Quiz question

**Question.** How can we implement

`concat :: [[a]] -> [a]` using `foldr`?

1. `concat = foldr (:) []`
2. `concat = foldr (++) []`
3. `concat = foldr (:) [[]]`
4. `concat = foldr (++) [[]]`

# Extra benefit of `foldr`: optimizations

Some advanced compiler optimizations are easier on programs with `foldr`:

- `foldr` fusion:

$$\text{foldr } f \ v \ (\text{map } g \ xs)$$
$$== \text{foldr } (\backslash x \ y \rightarrow f \ (g \ x) \ y) \ xs$$

- The ‘banana split rule’:<sup>7</sup>

$$(\text{sum } xs, \ \text{length } xs)$$
$$== \text{foldr } (\backslash n \ (x, y) \rightarrow (n+x, 1+y))$$
$$(0, 0)$$

---

<sup>7</sup>E. Meijer et. al. (1991): *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*

# From the book: Binary string transmitter

**Goal:** Simulate transmission of a string of characters encoded as a list of binary numbers.

```
bin2int :: [Int] -> Int  
bin2int = foldr (\x y -> x + 2*y)
```

```
int2bin :: Int -> [Int]  
int2bin 0 = []  
int2bin n =  
  n `mod` 2 : int2bin (n `div` 2)
```

See section 7.6 of the book for the full example.

# A discussion on `foldr`

Suppose a fellow student makes the following claim:

*“All recursive functions on lists can be written in terms of `foldr`. ”*

**Question.** Do you agree or disagree? Why?

# The `foldl` function

`foldl` is a version of `foldr` that associates to the **left**:

$$\begin{aligned}\text{foldl } (+) \ 0 \ [x_1, x_2, x_3] \\ == ((0 + x_1) + x_2) + x_3\end{aligned}$$

The difference is significant for **non-associative operations** such as `(-)`:

$$\begin{aligned}\text{foldr } (-) \ 0 \ [1, 2, 3] &= 1 - (2 - (3 - 0)) = 2 \\ \text{foldl } (-) \ 0 \ [1, 2, 3] &= ((0 - 1) - 2) - 3 = -6\end{aligned}$$

# Live programming exercise

**Exercise.** Implement the function

```
reverse :: [a] -> [a] using foldl
```

# Recursive definition of `foldl`

```
foldl :: (b -> a -> b) -> b  
        -> [a] -> b  
foldl (#) v []      = v  
foldl (#) v (x:xs) =  
  foldl (#) (v # x) xs
```

**Question:** Can we define

`foldl f = foldr (flip f)`? If not, can we define `foldl` in terms of `foldr` in some other way?

# The problem with `foldl`

**Warning:** The `foldl` function is notorious for causing performance problems, in particular transient space leaks.

The cause for this is Haskell's lazy evaluation strategy (see week 5).

The prelude provides a **strict version** `foldl'` (with the same type as `foldl`) that is almost always more efficient.

# What's next?

Next lecture: Type classes

To do:

- Read the book:
  - Today: sections 4.5-4.6, 7.1-7.5
  - Next lecture: 8.5, 12.1-12.2
- Read the binary string transmitter example in section 7.6 of the book
- Continue on week 2 exercises on Weblab

# Defining and working with type classes

Lecture 5 of CSE 3100  
Functional Programming

---

Jesper Cockx  
Q3 2023-2024  
Technical University Delft

# Lecture plan

- Type classes recap
- Defining type class instances
- Defining new type classes
- The classes **Functor** and **Applicative**

# Recap: Working with Type Classes

---

## Example: Generic `lookup` using `Eq`

```
type Assoc k a = [ (k, a) ]
```

```
lookup :: Eq k =>
```

```
    k -> Assoc k a -> Maybe a
```

```
lookup k [] = Nothing
```

```
lookup k ((l, x) : xs)
```

```
  | k == l = Just x
```

```
  | otherwise = lookup k xs
```

# Recap: type classes

A type class defines a **family of types** that share a common **interface**.

- **Show**: types with `show`
- **Eq**: types with `(==)` and `(/=)`
- **Ord**: types with `(<)`, `(<=)`, ...
- **Num**: types with `(+)`, `(-)`, `(*)`, ...

# Looking for common structure

A type class captures a family of types that share some structure.

**Question.** Why look for this common structure?

# Looking for common structure

A type class captures a family of types that share some structure.

**Question.** Why look for this common structure?

- Avoid boilerplate code (DRY!)
- Enforce abstraction barriers
- Typeclass laws provide a sanity check for correctness
- Deeper understanding of your code

*“Type classes are the design patterns of FP”*

# Definition of the `Eq` class

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

- First line declares a new type class `Eq` with one parameter `a`
- Body of class has one or more **function declarations**
- Each function optionally has a **default implementation**

# Declaring new instances of Eq

```
data TrafficLight =  
    Red | Yellow | Green  
instance Eq TrafficLight where  
    Red      == Red      = True  
    Green    == Green    = True  
    Yellow   == Yellow   = True  
    _        == _        = False
```

- First line declares `TrafficLight` to be an instance of `Eq`
- Body gives implementation of class functions
- Skipped functions use default implementation

# Using our type class instance

```
> Red == Red
```

True

```
> Red == Green
```

False

```
> let speeds = [ (Green , 100) ,  
                  (Yellow, 50 ) ,  
                  (Red     , 0    ) ]
```

```
> lookup Yellow speeds
```

Just 50

# Minimal complete definitions

An instance `Eq X` has to define either

`(==) :: X -> X -> Bool` or

`(/=) :: X -> X -> Bool`, or both.

In Haskell docs: “*Minimal complete definition:*

`(==) / (/=)`”

**Question.** What happens if we define neither?

**Question.** What if `Eq` didn't have default implementations?

## Another example: the `Show` class

```
class Show a where
    show :: a -> String
    showList :: [a] -> ShowS
    showList = ...
    showsPrec :: Int -> a -> ShowS
    showsPrec = ...
```

```
instance Show TrafficLight where
    show Red      = "Red light"
    show Yellow   = "Yellow light"
    show Green    = "Green light"
```

# Automatically deriving classes

Instead of writing instances by hand, Haskell can automatically **derive** instances of built-in type classes such as **Eq** and **Show**:

```
data TrafficLight  
  = Red | Yellow | Green  
deriving (Eq, Show)
```

```
> Red /= Red
```

```
False
```

```
> show Green
```

```
"Green"
```

# The **Arbitrary** type class

QuickCheck uses the **Arbitrary** type class to generate random values and shrink them:

```
class Arbitrary a where
    arbitrary :: Gen a
    shrink     :: a -> [a]
```

You can make QuickCheck work with functions on your own data type by implementing an instance for the **Arbitrary** class.

**Exercise:** Define an instance of **Arbitrary** for the type **TrafficLight**.

# **Instances of parametrized datatypes**

---

**Question.** How to define an `Eq` instance for `Maybe`?

**First attempt:**

```
instance Eq Maybe where ...
```

Expecting one more argument to  
'`Maybe`' Expected a type, but  
'`Maybe`' has kind '`* -> *`'

**Question.** How to define an **Eq** instance for **Maybe**?

**Second attempt:**

```
instance Eq (Maybe a) where
    Nothing == Nothing = True
    Just x == Just y = x == y
    _ == _ = False
```

No instance for (**Eq** a) arising from  
a use of '**==**'

**Question.** How to define an **Eq** instance for **Maybe** ?

**Third attempt:**

```
instance Eq a => Eq (Maybe a) where
    Nothing == Nothing = True
    Just x == Just y = x == y
    _ == _ = False
```

```
> Just 5 == Just 7
False
```

It works now!

# **Eq** and **Show** instances for **Tree**

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

# Eq and Show instances for Tree

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

instance Eq a => Eq (Tree a) where
    Leaf x == Leaf y = x == y
    Node u v == Node w x =
        u == w && v == x
    _ == _ = False
```

# Eq and Show instances for Tree

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

instance Eq a => Eq (Tree a) where
    Leaf x == Leaf y = x == y
    Node u v == Node w x =
        u == w && v == x
        - == - = False

instance Show a => Show (Tree a) where
    show (Leaf x) = "Leaf " ++ show x
    show (Node u v) =
        "Node " ++ showP u ++ " " ++ showP v
    where showP x = "(" ++ show x ++ ")"
```

# **Eq** and **Show** instances for **Tree**

```
data Tree a = Leaf a
             | Node (Tree a) (Tree a)
deriving (Show, Eq)
```

**deriving** also works for parametrized datatypes!

# Live coding: **Arbitrary** (**Tree** a)

```
class Arbitrary a where
    arbitrary :: Gen a
    shrink      :: a -> [a]
```

**Goal:** Define an instance of **Arbitrary** for the type **Tree** a.

# Working with subclasses

---

# Subclass example

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
    compare :: a -> a -> Ordering
    -- ...
```

# Why use subclasses?

Two reasons to declare a class as a subclass of another:

- Shorter type signatures: we can write  
`Ord a => ...` instead of  
`(Eq a, Ord a) => ...`
- We can use functions from the base class in default implementations:

```
x <= y = x < y || x == y
```

# More examples of subclasses

```
class (Num a, Ord a) => Real a
  where ...

class (Real a, Enum a) => Integral a
  where ...

class (Num a) => Fractional a
  where ...

class (Fractional a) => Floating a
  where ...
```

# Defining your own classes

---

# A simple type class: **Reversible**

```
class Reversible a where  
    rev :: a -> a
```

```
instance Reversible [a] where  
    rev xs = reverse xs
```

```
instance Reversible (Tree a) where  
    rev (Leaf x)      = Leaf x  
    rev (Node l r)   = Node (rev r) (rev l)
```

# Example: Truthy and Falsy values in Haskell

In Python and other languages, values of many types are considered ‘truthy’ or ‘falsy’

```
>>> if 5: print("hey whoah")
hey whoah
```

Let's simulate this behaviour in Haskell with a type class!

```
class Booly a where
    bool :: a -> Bool
```

## Instances of **Booly** (1/2)

```
instance Booly Bool where
```

```
  bool x = x
```

```
instance Booly Int where
```

```
  bool x = x /= 0
```

```
instance Booly Double where
```

```
  bool x = x /= 0.0
```

## Instances of `Booly` (2/2)

```
instance Booly (Maybe a) where
```

```
    bool Nothing = False
```

```
    bool (Just x) = True
```

```
instance Booly [a] where
```

```
    bool [] = False
```

```
    bool (_:_ ) = True
```

# An `if/then/else` for `Bool` values

```
iffy :: Bool a => a -> b -> b -> b
iffy b x y =
  if bool b then x else y
```

```
> iffy [] "yes" "no"
```

```
"no"
```

```
> iffy False "yes" "no"
```

```
"no"
```

```
> iffy (Just False) "yes" "no"
```

```
"yes"
```

# Functors

---

# The **Functor** type class

The function `map` applies a function to *every element in a list*.

**Functor** generalizes this to other data structures for which we have a `map`-like function:

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

We can think of a functor as a **container** storing elements of type `a`.

# Playing with `fmap`

What is the result of evaluating these expressions?

- `fmap (+1) [1, 2, 3]`
- `fmap (+1) (Just 1)`
- `fmap (+1) Nothing`
- `fmap (+1) (Right 2)`
- `fmap (+1) (Left 3)`
- `fmap (+1) (*2)`

Try it out in GHCi!

# Examples of functors

```
instance Functor [] where  
  fmap f xs = map f xs
```

```
instance Functor Maybe where  
  fmap f Nothing = Nothing  
  fmap f (Just x) = Just (f x)
```

```
instance Functor Tree where  
  fmap f (Leaf x) = Leaf (f x)  
  fmap f (Node l r) =  
    Node (fmap f l) (fmap f r)
```

## Either a is a functor

```
instance Functor (Either a)
-- fmap :: (b -> c)
--           -> Either a b
--           -> Either a c
fmap f (Left x) = Left x
fmap f (Right y) = Right (f y)
```

## ( $\rightarrow$ ) a is a functor

**Reminder:** ( $\rightarrow$ ) a b is the same as a  $\rightarrow$  b.

```
instance Functor (( $\rightarrow$ ) a)
  -- fmap :: (b  $\rightarrow$  c)
  --            $\rightarrow$  (a  $\rightarrow$  b)
  --            $\rightarrow$  (a  $\rightarrow$  c)
fmap f g = f . g
```

**Question.** Can you think of a type constructor that can **not** be made into an instance of **Functor**?

**Question.** Can you think of a type constructor that can **not** be made into an instance of **Functor**?

**Answer.** Here is an example:

```
newtype Endo a = Endo (a -> a)
instance Functor Endo where
    fmap f (Endo g) = Endo ???
```

More generally, if the type parameter **a** occurs *to the left of a function arrow*, the type cannot be made into a **Functor**.

# **Applicative functors**

---

# Reminder: subclasses in Haskell

A **subclass** is a family that extends the interface of another type class:

```
-- Ord is a subclass of Eq
class Eq a => Ord a where
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  x <= y = (x < y) || (x == y)
```

Each instance of the subclass must already be an instance of the base class.

# 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
```

# The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`:
- For lists:
- For `Tree`:
- For `Either a`:
- For `(->) a`:

# The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists:
- For `Tree`:
- For `Either a`:
- For `(->) a`:

# The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`:
- For `Either a`:
- For `(->) a`:

# The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`: `pure x = Leaf x`
- For `Either a`:
- For `(->) a`:

# The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`: `pure x = Leaf x`
- For `Either a`:
- For `(->) a`:

# The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`: `pure x = Leaf x`
- For `Either a`: `pure x = Right x`
- For `(->) a`:

# The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`: `pure x = Leaf x`
- For `Either a`: `pure x = Right x`
- For `(->) a`: `pure x = const x`

# The function $(\langle * \rangle)$

The function

$$(\langle * \rangle) :: f (a \rightarrow b) \rightarrow f a \rightarrow f b$$

combines two containers by applying functions in the first to values in the second one.

**Example.** For the **Maybe** functor, we have

$$\text{Just } f \quad \langle * \rangle \quad \text{Just } x \quad = \quad \text{Just } (f \ x)$$

$$\text{Nothing} \quad \langle * \rangle \quad \underline{\quad} \quad = \quad \text{Nothing}$$

$$\underline{\quad} \quad \langle * \rangle \quad \text{Nothing} \quad = \quad \text{Nothing}$$

# Applicative instance for lists

For lists, the function `(<*>)` iterates over all possible combinations of functions and values:

```
instance Applicative [] where
    pure x      = [x]
    fs <*> xs = 
        [f x | f <- fs, x <- xs]
```

**Example.** `[(1+), (2*)] <*> [10, 20] =`  
`[11, 21, 20, 40]`

**Remark.** This is not the only way to make lists into an applicative functor (see Weblab).

# Combining containers

We can combine two applicative containers by chaining `pure` and `(<*>)`:

```
zipA :: Applicative f
      => f a -> f b -> f (a,b)
zipA xs ys =
  pure (,) <*> xs <*> ys
```

## Examples.

- `zipA (Just 1) (Just 2) = Just (1,2)`
- `zipA (Just 1) Nothing = Nothing`

# Quiz question

**Question.** What is the result of

```
zipA [1,2] ['a','b'] ?
```

1. `([1,2], ['a','b'])`
2. `[(1,'a'),(2,'b')]`
3. `[(1,'a'),(1,'b'),(2,'a'),(2,'b')]`
4. `[(1,'a'),(2,'a'),(1,'b'),(2,'b')]`

# Discussion question

Can every instance of the **Functor** type class  
be made into an instance of the  
**Applicative** type class?

Can a given functor be made into an  
**Applicative** functor in *different ways*?

# What's next?

Next lecture: IO and Monads

To do:

- Read the book:
  - Today: 8.5, 12.1-12.2
  - Next lecture: 10.1-10.5, 12.3
- Start on week 3 exercises on Weblab

# IO and Monads

Lecture 6 of CSE 3100  
Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

# Lecture plan

- Effectful programming with the `IO` type
- The `Monad` typeclass
- `Monad` examples: `Maybe`, `Either`, `[]`

# The **IO** type

---

CODE WRITTEN IN HASKELL  
IS GUARANTEED TO HAVE  
NO SIDE EFFECTS.

...BECAUSE NO ONE  
WILL EVER RUN IT?



# Impure operations

**Recap.** A function is called **pure** if its behaviour is fully described by how it maps its inputs to its outputs.

What are examples of things that violate purity?

# Impure operations

**Recap.** A function is called **pure** if its behaviour is fully described by how it maps its inputs to its outputs.

What are examples of things that violate purity?

- Mutable variables
- Exceptions
- Input and output
- File system access
- Network communication

**Question.** How to write Haskell programs that interact with the world?

**Question.** How to write Haskell programs that interact with the world?

**Answer.** Use the builtin Haskell type `IO a`!

# What is **IO**?

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

## Examples.

- `putStrLn "Hello" :: IO ()`
- `getLine :: IO String`

Unlike other builtin types such as `[a]` or `(a, b)`, we cannot give a definition of **IO** ourselves: it is built into Haskell.

# Executing IO actions

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 executed except in specific cases:

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

# Separating the pure from the impure

Haskell programs are separated into a **pure** part (that does *not* use `IO`) and an **impure** (or *effectful*) part (that does use `IO`).

We can convert a pure value (of type `a`) to an impure one (of type `IO a`), but not the other way. Once a value is in `IO`, it is stuck there!

**Advice.** Most Haskell programs should only use `IO` in a small part of the program (~ 10%).

# IO primitives: terminal I/O

```
putChar    :: Char -> IO ()  
putStr     :: String -> IO ()  
putStrLn   :: String -> IO ()  
print      :: Show a => a -> IO ()
```

```
getChar    :: IO Char  
getLine    :: IO String  
readLn    :: Read a => IO a
```

# IO primitives: reading and writing files

```
data IOMode = ReadMode  
             | WriteMode  
             | ReadWriteMode  
             | AppendMode
```

```
openFile   :: FilePath -> IOMode  
           -> IO Handle  
  
hPutStrLn :: Handle -> String -> IO ()  
  
hGetLine  :: Handle -> IO String  
  
hIsEOF    :: Handle -> IO Bool  
  
hClose    :: Handle -> IO ()
```

# Other things you can do with `IO`

- Generate random numbers
- Read and write mutable variables (`IORef`)
- Throw and catch IO exceptions
- Write graphics on the display
- Listen to keyboard and mouse events
- Communicate over the network
- Start other processes
- ...

Basically: anything that's not a pure function

# do notation

Haskell has a special syntax for writing sequences of `IO` actions, known as `do` notation:

```
main :: IO ()  
main = do  
    putStrLn "What's your name?"  
    name <- getLine  
    putStrLn ("Hello, " ++ name ++ "!")
```

# Anatomy of a `do` block

```
f :: IO a  
f = do  
    v1 <- a1  
    ...  
    vn <- an  
    f v1 ... vn
```

- Each `vi <- ai` is a **statement** with an action `ai :: IO bi`.
- The result `vi` of each statement can be used as a *pure value of type* `bi` in the rest of the do block.
- The final line must be an **IO** action of type `IO a` (not a statement).

# The function `return` (1/2)

The function `return :: a -> IO a` turns a pure value into an `IO` action:

It is often used at the end of a `do` block:

```
f :: a -> IO (b, c)
```

```
f x = do
```

```
  y <- g x
```

```
  z <- h x
```

```
  return (y, z)
```

## The function `return` (2/2)

**Warning.** The `return` function is unrelated to `return` in imperative languages.

```
main = do
    putStrLn "Spam"
    return ()
    putStrLn " and eggs"
    return ()
-- Output: Spam and eggs
```

A `return` in the middle of a `do` block does not terminate the function!

# Conditional actions with `when`

```
when :: Bool -> IO () -> IO ()
```

executes an action only when the condition is  
**True**:

```
main = do
    putStrLn "Pick a password:"
    pwd <- getLine
    when (length pwd < 6) $
        putStrLn "Warning: too short"
    ... -- do some more stuff
```

# Running a list of actions with `sequence`

`sequence :: [IO a] -> IO [a]` runs a list of `IO` actions and collect the results.

```
main = do
    putStrLn "Pick three colors"
    colors <- sequence
        [getLine, getLine, getLine]
    putStrLn "The sorted colors are:"
    sequence
        (map putStrLn (sort colors))
return ()
```

# IO is a functor

`fmap f act` applies the pure function `f` to the result of the `IO a` action `act`, producing a new `IO b` action.

**Example.** If `parse :: String -> Expr` then `fmap parse getLine :: IO Expr`.

So we can view `act :: IO a` as a ‘box’ holding a value of type `a` that will be available at run-time.

# Applicative instance for IO

We could define an **Applicative IO** instance as follows:

```
instance Applicative IO where
    pure x = return x
    mf <*> mx = do
        f <- mf
        x <- mx
        return (f x)
```

This is not the real definition<sup>1</sup> but it gives the right idea.

<sup>1</sup> **do**-notation requires a **Monad** instance, which requires an

# Live programming question

Write a small interactive Haskell program with a function `main` that stores a list of items. It should have a simple menu with three options:

1. List all current items on the list in sequence
2. Add a new item to the list
3. Remove the item at a given position in the list

**Bonus:** Add an option for saving/loading the list from a file.

# The hidden backdoor: `unsafePerformIO`

The module `System.IO.Unsafe` provides a function `unsafePerformIO :: IO a -> a` that allows executing an `IO` action inside a pure function.

**Warning (!!).** Due to laziness, predicting if and when the action will be executed is very hard!

Only use `unsafePerformIO` for debugging or better understanding of laziness, but avoid using it in real programs.

# Tracing Haskell functions

A common use of unsafe IO is the function

```
trace :: String -> a -> a:
```

```
fib :: Int -> Int
fib 0 = trace ("fib 0") 1
fib 1 = trace ("fib 1") 1
fib n = trace ("fib " ++ show n)
        (fib (n-1) + fib (n-2))
```

By adding `trace` to your functions, you can get a better understanding of how Haskell functions are evaluated at runtime.

# Monads

---

# Monads: impossible to understand?

*A monad is just a monoid in the category of endofunctors, what's the problem?*

# ...or completely trivial?

You already know everything there is to know about monads!

- Examples: **Maybe**, **Either**, **IO**, ...
- Higher-order functions
- Type classes
- Functors and applicative functors
- **do**-notation

# The Maybe monad

---

# Walk the line<sup>2</sup>

```
type Birds = Int  
type Pole  = (Birds, Birds)
```

```
checkBalance :: Pole -> Maybe Pole  
checkBalance (x, y)
```

```
  | abs (x-y) < 4 = Just (x, y)  
  | otherwise      = Nothing
```

```
landL :: Birds -> Pole -> Maybe Pole  
landL n (x, y) = checkBalance (x+n, y)
```

```
landR :: Birds -> Pole -> Maybe Pole  
landR n (x, y) = checkBalance (x, y+n)
```



---

<sup>2</sup><http://learnyouahaskell.com/a-fistful-of-monads>

# Walk the line

```
landingSequence :: Pole -> Maybe Pole
landingSequence pole0 =
  case landL 1 pole0 of
    Nothing -> Nothing
    Just pole1 -> case landR 4 pole1 of
      Nothing -> Nothing
      Just pole2 -> case landL (-1) pole2 of
        Nothing -> Nothing
        Just pole3 -> case landR (-2) pole3 of
          Nothing -> Nothing
          Just pole4 -> Just pole4
```

Can't we do better??

# Walk the line

Can't we do better?? Yes we can!

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b  
Nothing >>= f = Nothing  
Just x >>= f = f x
```

```
landingSequence :: Pole -> Maybe Pole  
landingSequence pole0 =  
Just pole0  
>>= landL 1  
>>= landR 4  
>>= landL (-1)  
>>= landR (-2)
```

# The bind operator

The function `(>>=)` is called **bind**.

The general type of bind is:

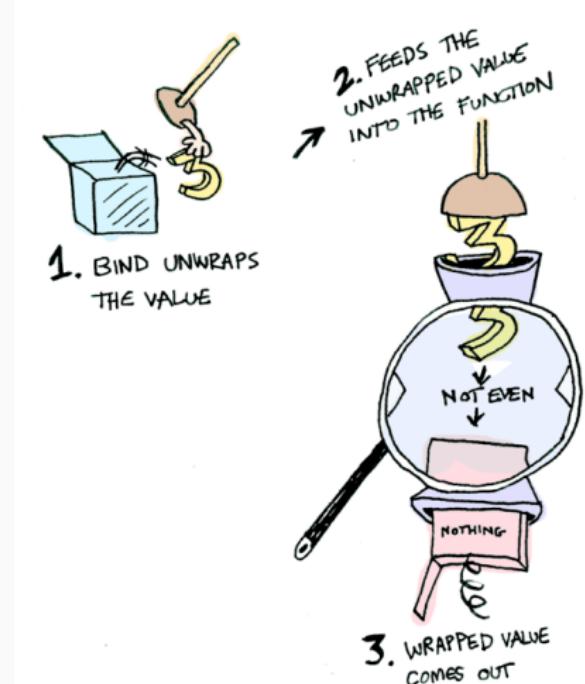
```
(>>=) :: m a -> (a -> m b) -> m b
```

where the type constructor `m` is a **monad**.

It is so useful, it made it into the Haskell logo:



# The bind operator<sup>3</sup>



<sup>3</sup>[https://adit.io/posts/2013-04-17-functors,\\_applicatives,\\_and\\_monads\\_in\\_pictures.html](https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html)

# The Monad class

---

# Functor, Applicative, and Monad

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

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

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

# Two ways to think about monads

You can think about a monadic type  $m\ a$  as...

- ... a **container** data structure that holds values of type  $a$ .
- ... a **computation** that can perform some side effects before returning a value of type  $a$ .

Both ways are valid and can be useful in different situations!

# 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.

# The sequencing operator (`>>`)

The sequencing operator `(>>)` executes one action after another, ignoring the output of the first.

```
(>>) :: Monad m => m a -> m b -> m b  
mx >> my = mx >>= (\_ -> my)
```

# More monadic functions

```
-- do the operation if the boolean is `True`  
when :: Applicative f => Bool -> f () -> f ()  
  
-- do the operation if the boolean is `False`  
unless :: Applicative f => Bool -> f () -> f ()  
  
-- run the actions from left to right.  
sequence :: Monad m => [m a] -> m [a]  
  
-- run the monadic function to each element  
-- in the list, combining the results.  
traverse :: Applicative f =>  
          (a -> f b) -> [a] -> f [b]
```

# do notation for monads

We can use do notation for any monad!

```
landingSequence :: Pole -> Maybe Pole
landingSequence pole0 = do
    pole1 <- landL 1 pole0
    pole2 <- landR 4 pole1
    pole3 <- landL (-1) pole2
    pole4 <- landR (-2) pole3
    return pole4
```

# Desugaring of **do** notation

## With **do**-notation

**do**

x **<-** f

g x

y **<-** h x

return (p x y)

## Without **do**-notation

f **>>=** (\x **->**

g x **>>** (

h x **>>=** (\y **->**

return (p x y)

)

)

)

# Monads in other languages

Several other languages use monads too:

- The `std::optional` library defines the **Maybe** monad in C++
- The `flatMap` function in Scala is precisely `(>>=)` for the list monad
- Promises in JavaScript (and other languages) form a monad<sup>4</sup>, with `.then()` acting as `(>>=)`.

---

<sup>4</sup>Almost, but not quite: see <https://hackernoon.com/functional-javascript-functors-monads-and-promises-679ce>

# The Promise monad in JavaScript

async/await is **do**-notation:<sup>5</sup>

```
async function getBalances() {  
    const accounts =  
        await web3.eth.accounts();  
    const balances =  
        await Promise.all(  
            accounts.map(web3.eth.getBalance));  
    return Ramda.zipObject(balances, accounts);  
}
```

---

<sup>5</sup><https://gist.github.com/MaiaVictor/bc0c02b6d1fbc7e3dbae838fb1376c80>

# Quiz: Which is which?

Monad

function

IO ()

action

Maybe

...is a ...

type

return 42

monad

(>>=)

type class

# What's next?

Next lecture: The **State** and **Parser** monads

To do:

- Read the book:
  - Today: 10.1-10.5, 12.3
  - Next lecture: 13.1-13.8
- Finish week 3 exercises on Weblab

# More monads

Lecture 7 of CSE 3100  
Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

# Lecture plan

- The **Either** monad
- The list monad
- The **State** monad
- The **Parser** monad
- The monad laws

# Recap: the **Monad** type class

**Monad** is a type class with functions `return` and `(>>=)` ('bind'):

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

A monadic action `x :: m a` can be seen as a **computation** that can perform some side effects before returning a value of type `a`.

Examples of monads: **Maybe**, **IO**.

# The **Either** monad

We can see **Either** as a generalization of **Maybe**:

- **Right** takes the role of **Just**
- **Left err** is a more informative version of **Nothing**

```
instance Monad (Either e) where
    return x = Right x
    Left err >>= f = Left err
    Right x >>= f = f x
```

# The list monad

We can see `[]` as another generalization of **Maybe**:

- `[x]` takes the role of `Just x`
- `[]` takes the role of `Nothing`
- Functions can have multiple outputs

`instance Monad [] where`

`return x = [x]`

`xs >>= f = [y | x <- xs, y <- f x]`

# Using the list monad

```
pairs :: [a] -> [b] -> [(a, b)]  
pairs xs ys = do x <- xs  
                  y <- ys  
                  return (x, y)
```

Compare this with list comprehensions:

```
pairs xs ys =  
  [ (x , y) | x <- xs  
            , y <- ys ]
```

You've been using the list monad all this time!

# The State monad

---

# Example: Generating random numbers

How to generate random numbers in Haskell?

```
-- chosen by fair dice roll  
-- guaranteed to be random.
```

```
randomNumber :: Int
```

```
randomNumber = 4
```

...not like that!

# Example: Generating random numbers

**Solution.** Write a pure function that takes a random seed as input.

```
import System.Random

randomNumber :: StdGen -> (Int, StdGen)
randomNumber = random

> randomNumber (mkStdGen 100)
(9216477508314497915, StdGen{...})
```

# Example: Generating random numbers

**Exercise.** Roll 3 six-sided dice & add results.

# Example: Generating random numbers

**Exercise.** Roll 3 six-sided dice & add results.

```
roll :: StdGen -> (Int, StdGen)
roll gen = let (x, newGen) = random gen
           in (x `mod` 6, newGen)
```

```
roll3 :: StdGen -> (Int, StdGen)
roll3 gen0 =
  let (die1, gen1) = roll gen0
      (die2, gen2) = roll gen1
      (die3, gen3) = roll gen2
  in (die1+die2+die3, gen3)
```

Can't we do better??

# The **State** monad

```
newtype State s a = State (s -> (a, s))
```

```
get :: State s s
```

```
get = State (\st -> (st, st))
```

```
put :: s -> State s ()
```

```
put st = State (\_ -> (((), st)))
```

# Rolling dice with **State**

```
randomInt :: State StdGen Int  
randomInt = State random
```

```
roll :: State StdGen Int  
roll = do  
    x <- randomInt  
    return (x `mod` 6)
```

```
roll3 :: State StdGen Int  
roll3 = do  
    die1 <- roll  
    die2 <- roll  
    die3 <- roll  
    return (die1+die2+die3)
```

# Functor and applicative for State

```
instance Functor (State s) where
    fmap f (State h) =
        State (\oldSt ->
            let (x, newSt) = h oldSt
            in (f x, newSt))
```

```
instance Applicative (State s) where
    pure x = State (\st -> (x, st))
    State g <*> State h =
        State (\oldSt ->
            let (f, newSt1) = g oldSt
                (x, newSt2) = h newSt1
            in (f x, newSt2))
```

# Binding the state

```
runState :: State s a -> s -> (a, s)
runState (State h) = h

instance Monad (State s) where
    return x = pure x

    State h >>= f =
        State (\oldSt ->
            let (x, newSt) = h oldSt
            in runState (f x) newSt)
```

# Reader and Writer monads

The **reader monad** gives access to an extra input of some type `r`:

```
newtype Reader r a = Reader (r -> a)
```

The **writer monad** allows writing some output of type `w`:

```
newtype Writer w a = Writer (w, a)
```

See exercises on Weblab!

# Monadic parsing

---

# What is a parser?

At a basic level, a parser turns strings into objects of some type:

```
type Parser a = String -> a
```

```
item :: Parser Char
```

```
item (x:[]) = x
```

```
item _      = error "Parse failed!"
```

Problems:

- No option for graceful failure
- Hard to compose parsers

# A better parser

```
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

*A parser of things  
is a function from strings  
to lists of pairs  
of things and strings!*

# A monadic parser

We wrap `Parser` in a `newtype` to make it into a monad:

```
newtype Parser a =
  Parser (String -> [(a, String)])
parse :: Parser a -> String -> [(a, String)]
parse (Parser f) = f

instance Functor Parser where ...
instance Applicative Parser where ...
instance Monad Parser where ...
```

See book for implementation of instances.

# Writing monadic parsers

We can now make use of **do** notation to write parsers:

```
three :: Parser (Char, Char)
three = do
    c1 <- item
    c2 <- item
    c3 <- item
    return (c1, c3)
```

# Writing monadic parsers

We can now make use of `do` notation to write parsers:

```
word :: Parser String
word = do
    c <- item
    if (isSpace c) then
        return ""
    else do
        cs <- word
        return (c:cs)
```

# Picky parsers

We can define a parser `empty` that always fails:

```
empty :: Parser a
empty = Parser []
```

This is useful to write parsers that only succeed when some property is satisfied:

```
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- char
           if p x then return x else empty
```

```
digit :: Parser Char
digit = sat isDigit
```

# Choosing between parses

We can combine two parsers by using the second one if the first one fails:

```
(<|>) :: Parser a -> Parser a -> Parser a
(Parser f <|> Parser g) = Parser
  (\inp -> case f inp of
    []      -> g inp
    result -> result)
```

-- Parsing an optional thing

```
maybeP :: Parser a -> Parser (Maybe a)
maybeP p = fmap Just p <|> pure Nothing
```

# Parsing several things

`some` repeats a parser one or more times.

`many` repeats a parser zero or more times.

```
some many :: Parser a -> Parser [a]
```

```
some x = pure (:) <*> x <*> many x
```

```
many x = some x <|> pure []
```

```
nat :: Parser Int
```

```
nat = do xs <- some digit  
           return (read xs)
```

# Live coding: parsing boolean expressions

**Assignment.** Develop a parser for the following grammar:

```
expr ::= atom
      | true
      | false
      | expr && expr
      | expr || expr
      | ~ expr
      | (expr)
```

where *atom* can be any string of letters.

# The Monad laws

---

# Type class laws

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

These laws are **not checked** by the compiler, but they form a contract between Haskell programmers.

So **check<sup>1</sup>** that your implementation satisfies the laws when implementing an instance!

---

<sup>1</sup>for example, using a QuickCheck property

# Example: Laws of **Eq**

- Reflexivity:  $x == x = \text{True}$
- Symmetry:  $(x == y) = (y == x)$
- Transitivity: If  
 $(x == y \&& y == z) = \text{True}$  then  
 $x == z = \text{True}$
- Substitututivity: If  $x == y = \text{True}$  then  
 $f\ x == f\ y = \text{True}^2$
- Negation:  $x /= y = \text{not } (x == y)$

---

<sup>2</sup>where `f :: a -> b` and `a` and `b` are both instances of **Eq**.

# The functor laws

`fmap f` applies `f` to each value stored in the container, but should *leave the structure of the container unchanged.*

This is expressed formally by the functor laws:

$$\text{fmap id} = \text{id}$$

$$\text{fmap (g . h)} = \text{fmap g . fmap h}$$

# A bogus instance of **Functor**

```
instance Functor Tree where
    fmap f (Leaf x)      = Leaf (f x)
    fmap f (Node l r)   =
        Node (fmap f r) (fmap f l)
```

# A bogus instance of **Functor**

```
instance Functor Tree where
```

```
fmap f (Leaf x) = Leaf (f x)
```

```
fmap f (Node l r) =
```

```
    Node (fmap f r) (fmap f l)
```

This does not satisfy the law `fmap id = id`:

```
fmap id (Node (Leaf 1) (Leaf 2))
```

```
= Node (Leaf 2) (Leaf 1)
```

```
≠ id (Node (Leaf 1) (Leaf 2))
```

# The four laws of Applicative

pure id  $\langle * \rangle$  x = x

pure (f x) = pure f  $\langle * \rangle$  pure x

mf  $\langle * \rangle$  pure y

= pure (\g -> g y)  $\langle * \rangle$  mf

x  $\langle * \rangle$  (y  $\langle * \rangle$  z)

= (pure (. )  $\langle * \rangle$  x  $\langle * \rangle$  y)  $\langle * \rangle$  z

# The Monad Laws

---

# Monad law #1: Left identity

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

**Intuition.** We can remove `return` statements in the middle of a `do`-block.

`do`

...

y `<-` return x

foo y

`do`

...

f x

# Monad law #2: Right identity

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

**Intuition.** We can eliminate `return` at the end of a `do`-block.

`do`

...

`x <- mx`

`return x`

`do`

...

`mx`

# Monad law #3: The associativity law

$$(mx \gg= f) \gg= g$$
$$=$$
$$mx \gg= (\backslash x \rightarrow (f x \gg= g))$$

**Intuition.** We can ‘flatten’ nested `do`-blocks.

`do`
$$\begin{array}{c} y \leftarrow \text{do } x \leftarrow mx \\ \quad f x \\ g y \end{array}$$
`do`
$$\begin{array}{c} x \leftarrow mx \\ y \leftarrow f x \\ g y \end{array}$$

# What's next?

Next lecture: Laziness and infinite data

To do:

- Read the book:
  - Today: section 13.1-13.8
  - Next lecture: 15.1-15.5, 15.7
- Start on week 4 exercises on Weblab

# Laziness

## Lecture 8 of CSE 3100 Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

[*TODO*: insert joke about laziness.]

# Lecture plan

- Lazy evaluation
- Forcing strictness
- Infinite data structures
- Case study: computing primes

# Lazy evaluation

---

# Evaluation strategies

An **evaluation strategy** gives a general way to pick which subexpression 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

**Note.** There are many other evaluation strategies ‘in between’ these two extremes.

## Side note: innermost and outermost reduction

In Haskell, a lambda expression is a **black box**: its body will never be evaluated before it is applied.

Evaluation strategies that do evaluate under lambdas:

- **Innermost reduction** is call-by-value with evaluation under lambdas.
- **Outermost reduction** is call-by-name with evaluation under lambdas.

# Evaluating map

**Question.** How is

`head (map (1+) [1, 2, 3])` evaluated  
under call-by-value and call-by-name?

# Evaluating map

## Call-by-value

```
head (map (1+) (1:2:3:[]))  
--> head ((1+1):map (+1) (2:3:[]))  
--> head (2:map (+1) (2:3:[]))  
--> head (2:(2+1):map (+1) (3:[]))  
--> head (2:3:map (+1) (3:[]))  
--> head (2:3:(3+1):map (+1) ([]))  
--> head (2:3:4:map (+1) ([]))  
--> head (2:3:4:[])  
--> 2
```

# Evaluating `map`

## Call-by-name

```
head (map (1+) (1:2:3:[]))  
--> head ((1+1):map (1+) (2:3:[]))  
--> 1+1  
--> 2
```

## Two definitions of `fac`

`fac 0 = 1`

`fac n = n * fac (n - 1)`

`fac' n = acc 1 n`

**where**

`acc x 0 = x`

`acc x y = acc (x*y) (y-1)`

**Question.** How are `fac 3` and `fac' 3` evaluated under call-by-name and call-by-value? Which one is more efficient?

# Non-terminating programs

Some programs will go into an infinite loop with any evaluation strategy.

```
inf :: Integer
```

```
inf = 1 + inf
```

```
inf
```

```
--> 1 + inf
```

```
--> 1 + 1 + inf
```

```
--> 1 + 1 + 1 + inf
```

```
--> ...
```

# Non-terminating programs

Other programs will go into an infinite loop with call-by-value, but not with call-by-name:

-- with *call-by-value*

fst (0, inf)

--> fst (0, 1 + inf)

--> fst (0, 1 + 1 + inf)

--> ...

-- with *call-by-name*

fst (0, inf)

--> 0

# Avoiding useless work

For functions that don't (always) use their arguments,  
*call-by-value* will do useless work:

```
const 42 (2+3)
--> const 42 5 --> 42
```

For functions that use their arguments more than once,  
*call-by-name* will do useless work:

```
square (2+3)
--> (2+3) * (2+3)
--> 5 * (2+3) --> 5 * 5 --> 25
```

**Can we get the best of both worlds? Yes!**

# Lazy evaluation

Lazy evaluation (aka *call-by-need*) is a variant of call-by-name that avoids double evaluation.

Each function argument is turned into a thunk:

- The first time the argument is used, the thunk is evaluated and the result is stored in the thunk.
- The next time the value stored in the thunk is used.

# Lazy evaluation in a nutshell

*Under lazy evaluation, programs are evaluated **at most once** and **only as far as needed**.*

# Lazy evaluation in other languages

Haskell is a **lazy language**: all evaluation is lazy by default.

Most other languages are **eager** (aka *strict*), but still have some form of lazy evaluation:

- *Lazy 'and'/'or'* (almost all languages): `False && b` evaluates to `False` without evaluating `b`.
- *Iterators* (e.g. Java) can produce values on-demand.
- *Generator functions* (e.g. Python) can use `yield` to lazily return values.
- *lazy val* (in Scala) declares a value that is computed lazily.

# Advantages of lazy evaluation

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

# Pitfalls of lazy evaluation

- Creation and management of thunks has some **runtime overhead**.
- It is **hard to predict** the order of evaluation.<sup>1</sup>
- Big intermediate expressions sometimes lead to a **drastic increase in memory usage**.

---

<sup>1</sup>Usually not a problem, unless you use `unsafePerformIO`.

# Forcing strict evaluation

---

# Performance drawbacks of lazy evaluation

The number of steps is not the only thing that matters for performance: the **size of intermediate terms** is also important:

- For small expressions that evaluate to a large data structure, call-by-need is better  
`(replicate 10000000 "spam") !! 5`
- For big expressions that evaluate to a small value, call-by-value is better  
`foldl (+) 0 [1..10000000]`

# Summing a long list

Let's take a closer look:

```
foldl (+) 0 [1..100000000]
--> foldl (+) (0+1) [2..100000000]
--> foldl (+) ((0+1)+2) [3..100000000]
--> foldl (+) (((0+1)+2)+3) [4..100000000]
--> ...
```

What happens if you try to evaluate

`foldl (+) 0 [1..100000000]` in GHCi?

# The problem with large intermediate expressions

Recursive functions (like `foldl`) can create large intermediate expressions during evaluation, which is bad for performance:

- Each intermediate expression requires a new thunk to be allocated.
- Too large intermediate expressions cause stack overflows.

Maybe being a *little* less lazy would help?

# Forcing strict evaluation

Haskell provides a built-in function `seq`:

```
seq :: a -> b -> b
```

The expression `seq u v` will evaluate `u` before returning `v`.

```
(1+2) `seq` 5 --> 3 `seq` 5 --> 5
```

```
replicate 5 'c' `seq` 42  
--> 'c':(replicate 4 'c') `seq` 42  
--> 42
```

# Strict application

Using `seq`, we can define **strict application**:

```
(\$!) :: (a -> b) -> a -> b  
f \$! x = x `seq` f x
```

*"Please evaluate `x` before applying `f`!"*

```
square \$! (1+2)      [           ]  
--> x `seq` square x [ x := 1+2 ]  
--> x `seq` square x [ x := 3   ]  
--> square x         [ x := 3   ]  
--> x*x              [ x := 3   ]  
--> 9
```

# Forcing evaluation of multiple arguments

-- force evaluation of x:

(f \$! x) y

-- force evaluation of y:

(f x) \$! y

-- force evaluation of x and y:

(f \$! x) \$! y

# A strict version of `foldl`

We can define a version of `foldl` that is strict in its second argument:

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' (#) v [] = v
foldl' (#) v (x:xs) =
  (foldl' (#) $! (v # x)) xs
```

Now we can evaluate

```
foldl' (+) 0 [1..100000000]
```

without running out of memory!

# Infinite data structures

---

# An infinite list

```
ones :: [Int]
```

```
ones = 1 : ones
```

```
ones --> 1 : ones
```

```
      --> 1 : (1 : ones)
```

```
      --> 1 : (1 : (1 : ones))
```

```
      --> ...
```

```
head ones --> head (1 : ones)
```

```
      --> 1
```

# Infinite data structures

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

**Intuition.** An infinite list is a **stream** of data that produces as much elements as required by its context.

# Quiz question

**Question.** Which of the following defines the infinite list `evens = 0:2:4:6:...`?

1. `evens = 0 : 2 : tail evens`
2. `evens = 0 : map (+2) (tail evens)`
3. `evens = 0 : map (+2) evens`
4. `evens = map (+2) [0..]`

# Syntactic sugar for (infinite) lists

[m..] denotes the list of all integers starting from m:

```
> [1..]  
[1,2,3,4,5,6,7, {Interrupted}  
> zip [1..] "hallo"  
[(1,'h'),(2,'a'),(3,'l'),(4,'l'),(5,'o')]
```

In fact, [m..] is syntactic sugar for enumFrom m:

```
enumFrom :: (Enum a) => a -> [a]
```

# Infinite list of prime numbers

```
sieve (x:xs) =  
  let xs' = [ y | y <- xs, y `mod` x /= 0 ]  
  in x : sieve xs'
```

```
primes :: [Int]  
primes = sieve [2..]
```

```
> take 10 primes  
[2,3,5,7,11,13,17,19,23,29]  
> primes !! 10000  
104743  
> head (dropWhile (<2023) primes)  
2027
```

# Separating data and control

With infinite data structures, we can separately define:

- *what we want to compute (the data)*
- *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`, ...

# Functions for constructing infinite lists

```
repeat :: a -> [a]
repeat x = xs
  where xs = x : xs
```

```
cycle :: [a] -> [a]
cycle xs = xs'
  where xs' = xs ++ xs'
```

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

# Filtering infinite lists

**Warning.** Filtering an infinite list will loop forever, even if the result is finite:

```
> filter (<5) [1..]  
[1,2,3,4,<loop>]
```

Instead, use `takeWhile` to get an initial fragment of an infinite list:

```
> takeWhile (<5) [1..]  
[1,2,3,4]
```

# The tree labeling problem

Remember the datatype of labeled trees:

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

**Exercise.** Given a tree and an infinite list of labels

`xs :: [Int]`, define a function

`label :: [Int] -> Tree a -> Tree (Int, a)`

that labels the tree with `xs`, using each label at most once.

# Other infinite data structures

Any (recursive) datatype in Haskell can have infinite structures, not just lists:

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

```
infTree :: Int -> Tree Int  
infTree n = Node subTree n subTree  
where subTree = infTree (n+1)
```

See exercises on WebLab!

# **Case study: computing fast primes**

---

# Working with infinite ascending lists

**Exercise 1.** Define a function

```
merge :: Ord a => [a] -> [a] -> [a]
```

that merges two *ascending* infinite lists into one (removing duplicate entries).

```
> take 10 (merge [2,4..] [3,6..])  
[2,3,4,6,8,9,10,12,14,15]
```

**Exercise 2.** Define a function

```
(\\) :: Ord a => [a] -> [a] -> [a]
```

that takes two *ascending* infinite lists and returns the list of elements that are in the first but not in the second list.

# A faster way to calculate primes (1/5)

We could define the infinite list of prime numbers is to first define the infinite list **composites** of *non-prime* numbers:

```
primesV2 :: [Integer]  
primesV2 = [2..] \\ composites
```

**Question.** How to compute **composites** ?

# A faster way to calculate primes (2/5)

```
multiples = [ map (*n) [n..] | n <- [  
mergeAll (xs:xss) = merge xs (mergeAll  
composites = mergeAll multiples
```

This loops forever!

## A faster way to calculate primes (3/5)

We can fix the loop by using the fact that the smallest element is always in the first list:

```
multiples = [ map (*n) [n..] | n <- [2..] ]
xmerge (x:xs) ys = x : merge xs ys
mergeAll (xs:xss) = xmerge xs (mergeAll xss)
composites = mergeAll multiples
```

primesV2 is faster than primes !

## A faster way to calculate primes (4/5)

We can avoid a lot of work by only considering multiples of prime numbers in the calculation of `composites`:

```
primesV3 = [2..] \\ composites
```

**where**

```
composites = mergeAll primeMultiples
primeMultiples = [ map (p*) [p..]
                  | p <- primesV3 ]
```

```
> take 10 primesV3
<loop>
```

Oh no, it's looping again!

# A faster way to calculate primes (5/5)

To get the recursion started, we need to specify that **2** is the first prime number:

```
primesV3 = 2 : ([3..] \\ composites)
```

**where**

```
composites = mergeAll primeMultiples
primeMultiples =
    [ map (p*) [p..] | p <- primesV3 ]
```

```
> take 10 primesV3
[2,3,5,7,11,13,17,19,23,29]
```

This one is much faster than V1!

# What's next?

Next lecture: Getting started with Agda

To do:

- Read the book:
  - This lecture: sections 15.1-15.5, 15.7
  - Next lecture: section 1 of Agda lecture notes
- Finish week 4 exercises on Weblab
- Install Agda on your PC (see instructions on Brightspace)

# Introduction to Agda

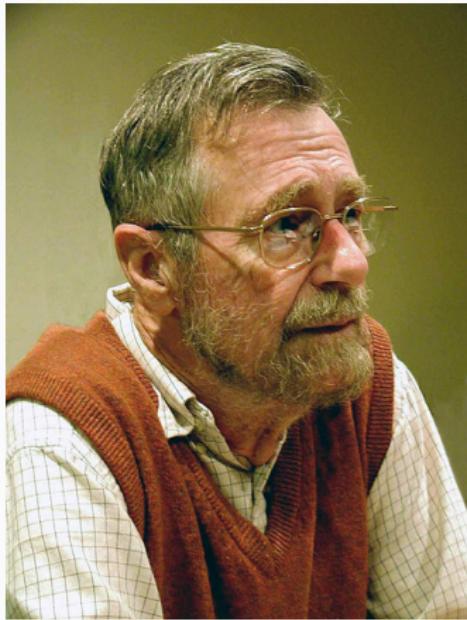
Lecture 11 of CSE 3100  
Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft



“Program testing can be used to show the presence of bugs, but never to show their absence!”

– Edsger W. Dijkstra

# Lecture plan

- A brief overview of formal verification, dependent types, and Agda
- Syntax differences between Agda and Haskell
- Interactive programming in Agda
- Types as first-class values
- Total functional programming

# When testing is just not enough

**Question.** In what situations might testing not be enough to ensure software works correctly?

# When testing is just not enough

**Question.** In what situations might testing not be enough to ensure software works correctly?

- ... failure is **very costly** (e.g. spacecraft, medical equipment, self-driving cars)
- ... the software is **difficult to update** (e.g. embedded software)
- ... it is **security-sensitive** (e.g. banking, your private chats)
- ... errors are **hard to detect** or **not apparent until much later** (e.g. compilers, concurrent systems)

# Formal verification

Formal verification is a collection of techniques for proving correctness of programs with respect to a certain formal specification.

These techniques often rely on ideas from formal logic and mathematics to ensure a very high degree of trustworthiness.

# Forms of formal verification

- Model checking systematically explores all possible executions of a program.
- Deductive verification uses external or internal tools to analyse and prove correctness of a program.
- Lightweight formal methods automatically verify a class of properties such as type safety or memory safety.

# Why dependent types?

Dependent types are a form of deductive verification that is **embedded in the programming language**.

## Advantages.

- No different syntax to learn or tools to install
- Tight integration between IDE and type system
- Express invariants of programs in their types
- Use same syntax for programming and proving

Formally verifying a program should not be more difficult than writing the program in the first place!

# The Agda language



Agda is a **purely functional** programming language similar to Haskell.

Unlike Haskell, it has full support for **dependent types**.

It also supports **interactive programming** with help from the type checker.

# Building your own Agda

An important goal of Agda is to **experiment with new language features.**

As a consequence, many common language features are not built into Agda, but they can be defined.

**Example.** `if/then/else` is not built into Agda but can be defined as a function.

While we could import these from the standard library, here we will instead **build them ourselves** from the ground up.

# Hello, Agda

---

# Installing Agda

## **Binary release.** (Linux/WSL/VM)

```
sudo apt install agda
```

## **From source.** (Cabal/Stack)

```
cabal install Agda or
```

```
stack install Agda
```

## **Via the VS Code plugin.**

Install the `agda-mode` plugin and enable the Agda Language Server in the settings.

# Installing an editor for Agda

The following editors have support for Agda:

- **VS Code:** Install the agda-mode plugin
- **Emacs:** Plugin is distributed with Agda (run `agda-mode setup`)
- **Atom:** `https://atom.io/packages/agda-mode`
- **Vim:** `https://github.com/derekelkins/agda-vim`

# A first Agda program

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

This program:

- Defines a datatype `Greeting` with one constructor `hello`.
- Defines a function `greet` of type `Greeting` that returns `hello`.

# Loading an Agda file

You can **load** an Agda file by pressing `Ctrl+c` followed by `Ctrl+l`.

Once the file is loaded (and there are no errors), other commands become available:

`Ctrl+c Ctrl+d` Infer type of an expression.

`Ctrl+c Ctrl+n` Evaluate an expression.

# Syntax of Agda vs. Haskell

---

# Basic syntax differences

**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 (+)

# Unicode syntax

Agda allows **unicode characters** in its syntax:

- $\rightarrow$  can be used instead of `->`
- $\lambda$  can be used instead of `\`
- Other symbols can also be used as (parts of) names of functions, variables, or types:  
 $\times, \Sigma, \top, \perp, \equiv, \langle, \rangle, \circ, \dots$

# Entering unicode

Editors with Agda support will replace  
LaTeX-like syntax (e.g. \to) with unicode:

→	\to
λ	\lambda
×	\times
Σ	\Sigma
⊤	\top
⊥	\bot
≡	\equiv
...	

# Quiz question

**Question.** Which is NOT a valid name for an Agda function?

1.  $1+1=2$
2. foo bar
3.  $\lambda \rightarrow \times \Sigma$
4. if\_then\_else\_

# Declaring new datatypes

To declare a datatype in Agda, we need to give the **full type** of each constructor:

```
data Bool : Set where  
  true : Bool  
  false : Bool
```

We also need to specify that **Bool** itself has type **Set** (see later).

# Defining functions by pattern matching

Just as in Haskell, we can define new functions by **pattern matching**:

`not : Bool → Bool`

`not true = false`

`not false = true`

`_||_ : Bool → Bool → Bool`

`false || false = false`

`_ || _ = true`

# The type of natural numbers

```
data Nat : Set where
```

```
    zero : Nat
```

```
    suc : Nat → Nat
```

```
one    = suc zero
```

```
two    = suc one
```

```
three  = suc two
```

```
four   = suc three
```

```
five   = suc four
```

# Builtin support for numbers

Writing numbers with `zero` and `suc` is annoying and inefficient. We can enable Agda's support for machine integers as follows:

```
{-# BUILTIN NATURAL Nat #-}
```

Agda will then convert between numerals and `zero/suc` representation automatically:

`one'` = 1

`two'` = 2

`three'` = 3

# Functions on natural numbers

`isEven : Nat → Bool`

`isEven zero = true`

`isEven (suc zero) = false`

`isEven (suc (suc x)) = isEven x`

`_+_ : Nat → Nat → Nat`

`zero + y = y`

`(suc x) + y = suc (x + y)`

# Priority of infix operators

You can specify the priority and associativity of an operator with `infixl` or `infixr`:

```
infixl 10 _+_
```

```
infixl 20 _*_
```

```
myNumber = 1 + 2 * 3 + 4
```

- Parsed as `((1 + (2 * 3)) + 4)`
- With `infixr`, it would be parsed as `1 + ((2 * 3) + 4)` instead.

# Integers in Agda

**Question.** How would you define a type of integers in Agda?

# Integers in Agda

**Question.** How would you define a type of integers in Agda?

**Answer.** Here is one possibility:

```
data Int : Set where
  pos : Nat → Int
  zero : Int
  neg : Nat → Int
```

where **pos**  $n$  represents the number  $1 + n$  and  
**neg**  $n$  represents  $-(1 + n)$ .

# Interactive programming in Agda

---

# Holes in programs

A **hole** is a part of a program that is not yet complete. A hole can be created by writing ? or { !! } and loading the file (`Ctrl+c Ctrl+l`).

New commands for files with holes:

`Ctrl+c Ctrl+,` Give information about the hole

`Ctrl+c Ctrl+c` Case split on a variable

`Ctrl+c Ctrl+space` Give a solution for the hole

# Demo session

**Exercise.** Define the following Agda functions:

- `maximum : Nat → Nat → Nat`
- `_*_ : Nat → Nat → Nat`
- `_≤_ : Nat → Nat → Bool`

# Summary of interactive commands

<b>Ctrl+c Ctrl+l</b>	Load the file
<b>Ctrl+c Ctrl+d</b>	Deduce type of an expression
<b>Ctrl+c Ctrl+n</b>	Normalise an expression
<b>Ctrl+c Ctrl+,</b>	Get information about the hole
<b>Ctrl+c Ctrl+c</b>	Case split on a variable
<b>Ctrl+c Ctrl+space</b>	Give a solution for the hole

These commands will become more and more useful, so start using them now!

# **Types as first-class values**

---

# The type Set

In Agda, types such as `Nat` and `(Bool → Bool)` are themselves expressions of type `Set`.

We can pass around and return values of type `Set` just like values of any other type.

**Example.** Defining a type alias as a function:

`MyNat : Set`

`MyNat = Nat`

`myFour : MyNat`

`myFour = suc (suc (suc (suc zero)))`

# Polymorphic functions in Agda

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

`id : (A : Set) → A → A`

`id A x = x`

For example, we have `id Nat zero : Nat` and `id Bool true : Bool`.

# Hidden arguments

To avoid repeating the type at which we apply a polymorphic function, we can declare it as a **hidden argument** using curly braces:

```
id : {A : Set} → A → A
```

```
id x = x
```

Now we have `id zero` : Nat and `id true` : Bool.

# If/then/else as a function

We can define if/then/else in Agda as follows:

```
if_then_else_ : {A : Set} →
```

```
  Bool → A → A → A
```

```
(if true then x else y) = x
```

```
(if false then x else y) = y
```

This is an example of a mixfix operator.

**Example usage.**

```
test : Nat → Nat
```

```
test x = if (x ≤ 9000) then o else 42
```

# Polymorphic datatypes

Just like we can define polymorphic functions, we can also define **polymorphic datatypes** by adding a parameter ( $A : \text{Set}$ ):

```
data List (A : Set) : Set where
  []  : List A
  _∷_ : A → List A → List A
infixl 5 _∷_
```

**Note.** Agda does not have built-in support for list syntax  $[1, 2, 3]$ . Instead, we have to write  $1 :: 2 :: 3 :: []$ .

# A tuple type in Agda

Agda does not have a builtin type of tuples  $(x, y)$ , but we can define the product type  $A \times B$ :

```
data _×_ (A B : Set) : Set where  
  _,_ : A → B → A × B
```

$\text{fst} : \{A\ B : \text{Set}\} \rightarrow A \times B \rightarrow A$

$\text{fst}\ (x, y) = x$

$\text{snd} : \{A\ B : \text{Set}\} \rightarrow A \times B \rightarrow B$

$\text{snd}\ (x, y) = y$

# No pattern matching on Set

It is not allowed to pattern match on arguments of type Set:

- Not valid Agda code:

```
sneakyType : Set → Set
```

```
sneakyType Bool = Nat
```

```
sneakyType Nat = Bool
```

One reason for this is that Agda (like Haskell) erases all types during compilation.

# **Total functional programming**

---

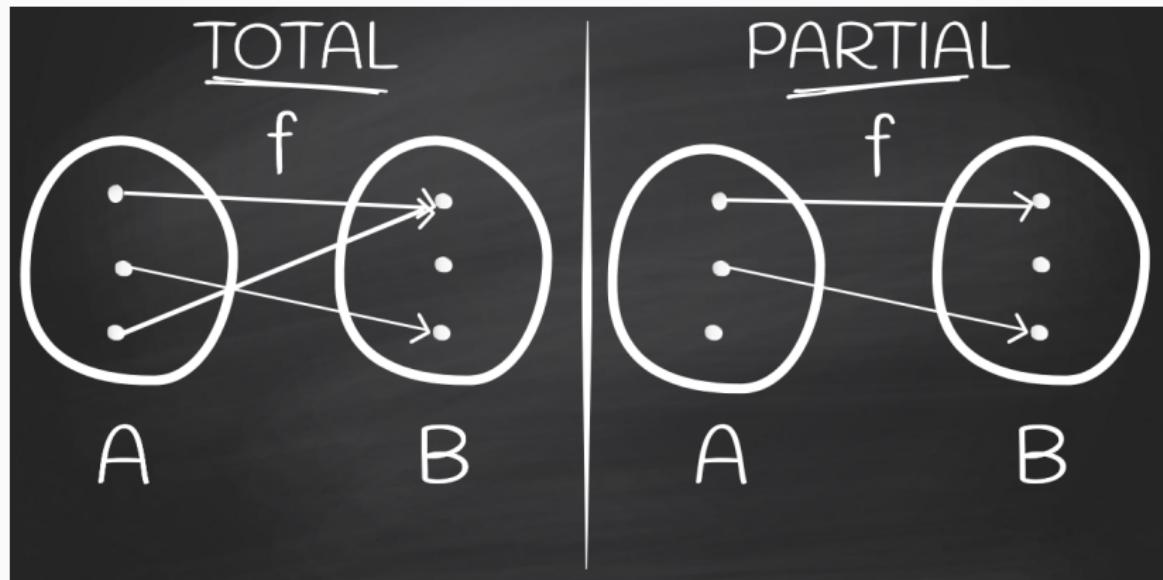
# Total functional programming

In contrast to Haskell, 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.**

# Totality in mathematics<sup>1</sup>



<sup>1</sup>Source: <https://kowainik.github.io/posts/totality>

# Why should we care about totality?

Some reasons to write total programs:

- Better guarantees of correctness
- Spend less time debugging infinite loops
- Easier to refactor without introducing bugs
- Less need to document valid inputs

Totality is also crucial for working with  
**dependent types** and using Agda as a **proof**  
assistant (see coming lectures).

# Coverage checking

Agda performs a **coverage check** to ensure all definitions by pattern matching are complete:

```
pred : Nat → Nat
```

```
pred (suc x) = x
```

Incomplete pattern matching for pred.

Missing cases: pred zero

# Termination checking

Agda performs a **termination check** to ensure all recursive definitions are terminating:

```
inf : Nat → Nat
```

```
inf x = 1 + inf x
```

Termination checking failed for the following functions: inf

Problematic calls: inf x

# To solve or not to solve the halting problem

**Question.** Isn't it impossible to determine whether a function is terminating? Or does Agda solve the halting problem?

# To solve or not to solve the halting problem

**Question.** Isn't it impossible to determine whether a function is terminating? Or does Agda solve the halting problem?

**Answer.** No, Agda only accepts functions that are **structurally recursive**, and rejects all other functions.

# Structural recursion

Agda only accepts functions that are **structurally recursive**: the argument of each recursive call must be a subterm of the argument on the left of the clause.

$f : \text{Nat} \rightarrow \text{Nat}$

$f(\text{suc } (\text{suc } x)) = f \text{ zero}$

$f(\text{suc } x) = f(\text{suc } (\text{suc } x))$

$f \text{ zero} = \text{zero}$

The function  $f$  is terminating but not structurally recursive, so it is rejected.

# Quiz question

**Question.** Which of these are possible to be defined in Agda?

- A data type with infinitely many elements
- A function that loops forever
- A function that pattern matches on types
- A function with infinitely many cases

# Discussion question

**Question.** Is it possible to implement a function of type  $\{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \text{Nat} \rightarrow A$  in Agda?

# What's next?

Next lecture: Dependent types

To do:

- Read the lecture notes:
  - This lecture: section 1 of Agda lecture notes
  - Next lecture: section 2 of Agda lecture notes
- Install Agda on your computer
- Start on Agda exercises on Weblab

# Dependent types

Lecture 12 of CSE 3100

Functional Programming

---

Jesper Cockx

Q3 2023-2024

Technical University Delft

# Lecture plan

- What's a dependent type?
- Dependent function types
- The Vector and Fin types
- Well-typed syntax

# What's a dependent type?

---

# Cooking with dependent types (1/3)

Suppose we are implementing a cooking assistant that can help with preparing three kinds of food:

```
data Food : Set where
  pizza : Food
  cake  : Food
  bread : Food
```

We want to implement a function

`amountOfCheese : Food → Nat` that computes how much cheese is needed.

**Problem:** How can we make sure this function is never called with argument `cake`?

# Cooking with dependent types (2/3)

**Solution.** We can make the type `Food` more precise making it into an **indexed datatype**:

```
data Flavour : Set where
    cheesy    : Flavour
    chocolatey : Flavour
```

```
data Food : Flavour → Set where
    pizza   : Food cheesy
    cake    : Food chocolatey
    bread : {f: Flavour} → Food f
```

This defines two types `Food cheesy` and `Food chocolatey`.

# Cooking with dependent types (3/3)

We can now rule out invalid inputs by using the more precise type `Food` `cheesy`:

```
amountOfCheese : Food cheesy → Nat
```

```
amountOfCheese pizza = 100
```

```
amountOfCheese bread = 20
```

The coverage checker of Agda knows that `cake` is not a valid input!

# Dependent type theory (1972)



A **dependent type** is a family of types, depending on a term of a **base type**.

Per  
Martin-Löf

# Dependent type theory (1972)



Per  
Martin-Löf

A **dependent type** is a family of types, depending on a term of a **base type**.

**Example.** **Food** is a dependent type indexed over the base type **Flavour**.

# Dependent types vs. parametrized types

**Question.** What is the difference between a dependent type such as `Food f` and a parametrized type such as `Maybe a`?

# Dependent types vs. parametrized types

**Question.** What is the difference between a dependent type such as `Food f` and a parametrized type such as `Maybe a`?

**Answer.** All types `Maybe a` have the same constructors (`Nothing` and `Just`) for all values of `a` is, while `Food f` has different constructors depending on `f`.

# The **Vec** type

---

# Vectors: lists that know their length

`Vec A n` is the type of **vectors** with exactly  $n$  arguments of type `A`:

```
myVec1 : Vec Nat 4
```

```
myVec1 = 1 :: 2 :: 3 :: 4 :: []
```

```
myVec2 : Vec Nat 0
```

```
myVec2 = []
```

```
myVec3 : Vec (Bool → Bool) 2
```

```
myVec3 = not :: id :: []
```

# Definition of the `Vec` type

`Vec A n` is a dependent type indexed over the base type `Nat`:

```
data Vec (A : Set) : Nat → Set where
  []  : Vec A 0
  _∷_ : {n : Nat} →
         A → Vec A n → Vec A (suc n)
```

This has two constructors `[]` and `_∷_` like `List`, but the constructors specify the length in their types.

# Parameters vs. indices

The argument ( $A : \text{Set}$ ) in the definition of `Vec` is a **parameter**, and has to be *the same in the type of each constructor*.

The argument of type `Nat` in the definition of `Vec` is an **index**, and must be *determined individually for each constructor*.

# Quiz question

**Question.** How many elements are there in the type `Vec Bool 3`?

# Quiz question

**Question.** How many elements are there in the type `Vec Bool 3`?

**Answer.** 8 elements:

- `true :: true :: true :: []`
- `true :: true :: false :: []`
- `true :: false :: true :: []`
- `true :: false :: false :: []`
- `false :: true :: true :: []`
- `false :: true :: false :: []`
- `false :: false :: true :: []`
- `false :: false :: false :: []`

# Type-level computation

During type-checking, Agda will **evaluate** expressions in types:

myVec4 : Vec Nat (2 + 2)

myVec4 = 1 :: 2 :: 3 :: 4 :: []

Every type is equal to its normal form:

Vec Nat (2 + 2) is the same type as Vec Nat 4.

Since Agda is total, every type has a unique normal form!

# Checking the length of a vector

Constructing a vector of the wrong length in any way is a **type error**:

```
myVec5 : Vec Nat 0
```

```
myVec5 = 1 :: 2 :: []
```

suc \_n\_46 != zero of type Nat  
when checking that the inferred  
type of an application

Vec Nat (suc \_n\_46)

matches the expected type

Vec Nat 0

# Dependent function types

A **dependent function type** is a type of the form  $(x : A) \rightarrow B x$  where the *type* of the output depends on the *value* of the input.

## Example.

```
zeroes : (n : Nat) → Vec Nat n
```

```
zeroes zero    = []
```

```
zeroes (suc n) = 0 :: zeroes n
```

E.g. `zeroes 3` has type `Vec Nat 3` and evaluates to `0 :: 0 :: 0 :: []`.

# Quiz question

**Question.** What's 'dependent' about a dependent function type?

1. The value of the output depends on the value of the input
2. The value of the output depends on the type of the input
3. The type of the output depends on the value of the input
4. The type of the output depends on the type of the input

# Concatenation of vectors

We can pattern match on `Vec` just like on `List`:

$$\text{mapVec} : \{A\ B : \text{Set}\} \{n : \text{Nat}\} \rightarrow \\ (A \rightarrow B) \rightarrow \text{Vec } A\ n \rightarrow \text{Vec } B\ n$$
$$\text{mapVec } f\ [] = []$$
$$\text{mapVec } f\ (x :: xs) = fx :: \text{mapVec } f\ xs$$

**Note.** The type of `mapVec` specifies that the output has the same length as the input.

# A safe head function

By making the input type of a function more precise, we can rule out certain cases **statically** (= during type checking):

```
head : {A : Set}{n : Nat} → Vec A (suc n) → A  
head (x :: xs) = x
```

Agda knows the case for `head []` is impossible!  
(just like for `amountOfCheese cake`)

# A safe tail function

**Question.** What should be the type of `tail` on vectors with the following definition?

```
tail (x :: xs) = xs
```

# A safe tail function

**Question.** What should be the type of `tail` on vectors with the following definition?

```
tail (x :: xs) = xs
```

**Answer.**

```
tail : {A : Set} {n : Nat} → Vec A (suc n) → Vec A n  
tail (x :: xs) = xs
```

# Live coding

**Exercise.** Define a function `zipVec` that only accepts vectors of the same length.

# The **Fin** type

---

# A safe lookup

By combining `head` and `tail`, we can get the 1st, 2nd, 3rd,... element of a vector with at least that many elements.

How can we define a function `lookupVec` that get the element at position  $i$  of a `Vec A n` where  $i < n$ ?

**Note.** We want to get an element of  $A$ , not of `Maybe A!`

# The Fin type

We need a type of indices that are *safe* for a vector of length  $n$ , i.e. numbers between 0 and  $n - 1$ .

This is the type `Fin n` of finite numbers:

```
zero3 one3 two3 : Fin 3
```

```
zero3 = zero
```

```
one3 = suc zero
```

```
two3 = suc (suc zero)
```

# Definition of the Fin type

```
data Fin : Nat → Set where
  zero : {n : Nat} → Fin (suc n)
  suc  : {n : Nat} → Fin n → Fin (suc n)
```

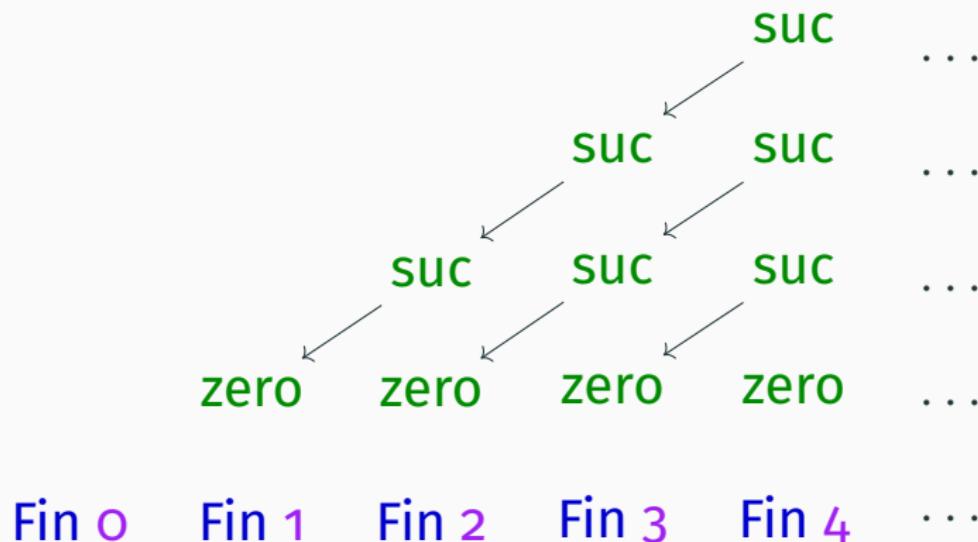
# An empty type

$\text{Fin } n$  has  $n$  elements, so in particular  $\text{Fin } 0$  has zero elements: it is an **empty type**.

This means there are *no valid indices* for a vector of length  $0$ .

**Note.** Unlike in Haskell, we cannot even construct an expression of  $\text{Fin } 0$  using undefined or an infinite loop.

# The family of Fin types



# A safe lookup (1/5)

`lookupVec : {A : Set} {n : Nat} →`  
`Vec A n → Fin n → A`

`lookupVec xs i =` {! !}

# A safe lookup (2/5)

```
lookupVec : {A : Set} {n : Nat} →  
  Vec A n → Fin n → A  
lookupVec (x :: xs) i = {! !}
```

# A safe lookup (3/5)

lookupVec : {A : Set} {n : Nat} →  
  Vec A n → Fin n → A

lookupVec (x :: xs) zero = {! !}

lookupVec (x :: xs) (suc i) = {! !}

# A safe lookup (4/5)

```
lookupVec : {A : Set} {n : Nat} →  
  Vec A n → Fin n → A  
lookupVec (x :: xs) zero    = x  
lookupVec (x :: xs) (suc i) = {! !}
```

# A safe lookup (5/5)

`lookupVec : {A : Set} {n : Nat} →`

`Vec A n → Fin n → A`

`lookupVec (x :: xs) zero = x`

`lookupVec (x :: xs) (suc i) = lookupVec xs i`

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

# Live coding exercise (1/2)

Define a datatype `Expr` of expressions of a small programming language with:

- Number literals  $0, 1, 2, \dots$
- Arithmetic expressions  $e_1 + e_2$  and  $e_1 * e_2$
- Booleans `true` and `false`
- Comparisons  $e_1 < e_2$  and  $e_1 == e_2$
- Conditionals `if`  $e_1$  `then`  $e_2$  `else`  $e_3$

`Expr` should be a *dependent type* indexed over the type `Ty` of possible types of this language:

```
data Ty : Set where
```

```
  tInt  : Ty
```

```
  tBool : Ty
```

## Live coding exercise (2/2)

Next, write a function `El : Ty → Set` that interprets a type of this language as an Agda type.

Finally, define `eval : {t : Ty} → Expr t → El t` that evaluates a given expression to an Agda value.

# Dependent types: Summary

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.

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

# What's next?

**Next lecture:** Using Agda as a theorem prover

**To do:**

- Read the lecture notes:
  - This lecture: section 2 of Agda lecture notes
  - Next lecture: section 3 of Agda lecture notes
- Do Weblab exercises on dependent types
- Continue hacking on the project and asking questions on TU Delft Answers

# The Curry-Howard Correspondence

Lecture 11 of CSE 3100  
Functional Programming

---

Jesper Cockx  
Q3 2023-2024  
Technical University Delft



*“Every good idea will be discovered twice:  
once by a logician and once  
by a computer scientist.”*

– Philip Wadler

# Lecture plan

- The Curry-Howard correspondence between type systems and logic
- Classical logic vs. constructive logic
- Curry-Howard for predicate logic
- Proof by induction in Agda

# The Curry-Howard Correspondence

---

# Goal of today's lecture

**Goal.** We want to write **proofs** that our program satisfies certain properties, and have the compiler **check** that these proofs are correct.

## Examples.

- For any  $x : \text{Nat}$ ,  $x + x$  is an even number.
- The length of  $\text{map } f \text{ xs}$  is equal to the length of  $\text{xs}$ .
- $\text{reverse} = \text{foldr } (\lambda x \text{ xs} \rightarrow \text{xs} \text{ ++ } x) []$  and  $\text{reverse}' = \text{foldl } (\lambda \text{xs } x \rightarrow x :: \text{xs}) []$  always return the same result.

# Formal verification with dependent types

**Reminder.** Formal verification is the process proving correctness of programs with respect to a certain formal specification.

Our goal is to use Agda as a proof assistant for doing formal verification.

To do this, we first need to answer the question: what exactly is a proof?

# What even is a proof? (1/3)

Traditionally, a proof is a **sequence of statements** where each statement is a direct consequence of previous statements.

**Example.** A proof that if (1)  $A \Rightarrow B$  and (2)  $A \wedge C$ , then  $B \wedge C$ :

- (3)  $A$  (follows from 2)
- (4)  $B$  (modus ponens with 1 and 3)
- (5)  $C$  (follows from 2)
- (6)  $B \wedge C$  (follows from 4 and 5)

# What even is a proof? (2/3)

We can make the dependencies of a proof more explicit by writing it down as a **proof tree**.

**Example.** Here is the same proof that if (1)  $A \Rightarrow B$  and (2)  $A \wedge C$ , then  $B \wedge C$ :

$$\begin{array}{c} A \Rightarrow B^{(1)} \qquad \frac{A \wedge C^{(2)}}{A} \qquad \frac{A \wedge C^{(2)}}{C} \\ \hline B \\ \hline B \wedge C \end{array}$$

# What even is a proof? (3/3)

To represent these proofs in a programming language, we can annotate each node of the tree with a **proof term**:

$$\frac{p : A \Rightarrow B}{\frac{\frac{q : A \wedge C}{\text{fst } q : A}}{\frac{\frac{q : A \wedge C}{\text{snd } q : C}}{(p (\text{fst } q), \text{snd } q) : B \wedge C}}}$$

# What even is a proof? (3/3)

To represent these proofs in a programming language, we can annotate each node of the tree with a **proof term**:

$$\begin{array}{c} p : A \Rightarrow B \\ \hline p (\text{fst } q) : B \end{array} \quad \begin{array}{c} q : A \wedge C \\ \hline \text{fst } q : A \end{array} \quad \begin{array}{c} q : A \wedge C \\ \hline \text{snd } q : C \end{array} \\ \hline (p (\text{fst } q), \text{snd } q) : B \wedge C$$

Hmm, these proof terms start to look a lot like functional programs...

# The Curry-Howard correspondence



Haskell B. Curry

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

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

# What is conjunction $A \wedge B$ ?

What do we know about the proposition  $A \wedge B$  (A and B)?

- To prove  $A \wedge B$ , we need to provide a proof of A and a proof of B.
- Given a proof of  $A \wedge B$ , we can get proofs of A and B

# What is conjunction $A \wedge B$ ?

What do we know about the proposition  $A \wedge B$  (A and B)?

- To prove  $A \wedge B$ , we need to provide a proof of A and a proof of B.
- Given a proof of  $A \wedge B$ , we can get proofs of A and B

⇒ The type of proofs of  $A \wedge B$  is the **type of pairs**  $A \times B$

# What is implication $A \Rightarrow B$ ?

What do we know about the proposition  $A \Rightarrow B$  (A implies B)?

- To prove  $A \Rightarrow B$ , we can assume we have a proof of A and have to provide a proof of B
- From a proof of  $A \Rightarrow B$  and a proof of A, we can get a proof of B

# What is implication $A \Rightarrow B$ ?

What do we know about the proposition  $A \Rightarrow B$  (A implies B)?

- To prove  $A \Rightarrow B$ , we can assume we have a proof of A and have to provide a proof of B
- From a proof of  $A \Rightarrow B$  and a proof of A, we can get a proof of B

⇒ The type of proofs of  $A \Rightarrow B$  is the **function type**  $A \rightarrow B$

# Proof by implication (Modus ponens)

Modus ponens says that if  $P$  implies  $Q$  and  $P$  is true, then  $Q$  is true.

**Question.** How can we prove this in Agda?

# Proof by implication (Modus ponens)

Modus ponens says that if  $P$  implies  $Q$  and  $P$  is true, then  $Q$  is true.

**Question.** How can we prove this in Agda?

**Answer.**

`modusPonens : {P Q : Set} → (P → Q) × P → Q`

`modusPonens (f , x) = fx`

# What is disjunction $A \vee B$ ?

What do we know about the proposition  $A \vee B$  ( $A$  or  $B$ )?

- To prove  $A \vee B$  we need to provide a proof of  $A$  or a proof of  $B$ .
- If we have:
  - a proof of  $A \vee B$
  - a proof of  $C$  assuming a proof of  $A$
  - a proof of  $C$  assuming a proof of  $B$then we have a proof of  $C$ .

# What is disjunction $A \vee B$ ?

What do we know about the proposition  $A \vee B$  ( $A$  or  $B$ )?

- To prove  $A \vee B$  we need to provide a proof of  $A$  or a proof of  $B$ .
- If we have:
  - a proof of  $A \vee B$
  - a proof of  $C$  assuming a proof of  $A$
  - a proof of  $C$  assuming a proof of  $B$then we have a proof of  $C$ .

⇒ The type of proofs of  $A \vee B$  is the **sum type**  
**Either A B**

# Proof by cases

Proof by cases says that if  $P \vee Q$  is true and we can prove  $R$  from  $P$  and also prove  $R$  from  $Q$ , then we can prove  $R$ .

**Question.** How can we prove this in Agda?

# Proof by cases

Proof by cases says that if  $P \vee Q$  is true and we can prove  $R$  from  $P$  and also prove  $R$  from  $Q$ , then we can prove  $R$ .

**Question.** How can we prove this in Agda?

**Answer.**

cases : { $P\ Q\ R : \text{Set}$ }

→ Either  $P\ Q \rightarrow (P \rightarrow R) \times (Q \rightarrow R) \rightarrow R$

cases (left  $x$ ) ( $f, g$ ) =  $f x$

cases (right  $y$ ) ( $f, g$ ) =  $g y$

# Quiz question

**Question.** Which Agda type represents the proposition “*If (P implies Q) then (P or R) implies (Q or R)*”?

1.  $(\text{Either } P \ Q) \rightarrow \text{Either } (P \rightarrow R) \ (Q \rightarrow R)$
2.  $(P \rightarrow Q) \rightarrow \text{Either } P \ R \rightarrow \text{Either } Q \ R$
3.  $(P \rightarrow Q) \rightarrow \text{Either } (P \times R) \ (Q \times R)$
4.  $(P \times Q) \rightarrow \text{Either } P \ R \rightarrow \text{Either } Q \ R$

# What is truth?

What do we know about the proposition ‘true’?

- To prove ‘true’, we don’t need to provide anything
- From ‘true’, we can deduce nothing

# What is truth?

What do we know about the proposition ‘true’?

- To prove ‘true’, we don’t need to provide anything
- From ‘true’, we can deduce nothing

⇒ The type of proofs of truth is the *unit type*  $\top$  with one constructor  $\text{tt}$ :

```
data  $\top$  : Set where  
  tt :  $\top$ 
```

# What is falsity?

What do we know about the proposition ‘false’?

- There is no way to prove ‘false’
- From a proof  $t$  of ‘false’, we get a proof  
absurd  $t$  of any proposition  $A$

# What is falsity?

What do we know about the proposition ‘false’?

- There is no way to prove ‘false’
- From a proof  $t$  of ‘false’, we get a proof  
absurd  $t$  of any proposition  $A$

⇒ The type of proofs of falsity is the **empty type**  $\perp$  with no constructors:

`data ⊥ : Set where`

# Principle of explosion

The principle of explosion<sup>1</sup> says that if we assume a false statement, we can prove any proposition  $P$ .

**Question.** How can we prove this in Agda?

---

<sup>1</sup>Also known as *ex falso quodlibet* = from falsity follows anything.

# Principle of explosion

The principle of explosion<sup>1</sup> says that if we assume a false statement, we can prove any proposition  $P$ .

**Question.** How can we prove this in Agda?

**Answer.**

`absurd : {P : Set} → ⊥ → P`

`absurd ()`

---

<sup>1</sup>Also known as *ex falso quodlibet* = from falsity follows anything.

# Curry-Howard for propositional logic

We can translate from the language of **logic** to the language of **types** according to this table:

Propositional logic		Type system
proposition	$P$	type
proof of a proposition	$p : P$	program of a type
conjunction	$P \times Q$	pair type
disjunction	$\text{Either } P Q$	either type
implication	$P \rightarrow Q$	function type
truth	$\top$	unit type
falsity	$\perp$	empty type

# Derived notions

**Negation.** We can encode  $\neg P$  (“not  $P$ ”) as the type  $P \rightarrow \perp$ .

**Equivalence.** We can encode  $P \Leftrightarrow Q$  (“ $P$  is equivalent to  $Q$ ”) as  $(P \rightarrow Q) \times (Q \rightarrow P)$ .

# An exercise in translation

**Exercise.** Translate the following statements to types in Agda, and prove them by constructing a program of that type:

1. If  $P$  implies  $Q$  and  $Q$  implies  $R$ , then  $P$  implies  $R$
2. If  $P$  is false and  $Q$  is false, then (either  $P$  or  $Q$ ) is false.
3. If  $P$  is both true and false, then any proposition  $Q$  is true.

# Three layers of Curry-Howard

1. Propositions are *types*

# Three layers of Curry-Howard

1. Propositions are *types*
2. Proofs are *programs*

# Three layers of Curry-Howard

1. Propositions are *types*
2. Proofs are *programs*
3. Simplifying a proof is *evaluating* a program

# Three layers of Curry-Howard

1. Propositions are *types*
2. Proofs are *programs*
3. Simplifying a proof is *evaluating* a program

**Example.** An indirect proof of  $A \rightarrow A$  evaluates to direct proof:

$$\begin{aligned} & \lambda x \rightarrow (\lambda y \rightarrow \text{fst } y) (x, x) \\ \longrightarrow & \quad \lambda x \rightarrow \text{fst } (x, x) \\ \longrightarrow & \quad \lambda x \rightarrow x \end{aligned}$$

# **Classical vs. constructive logic**

---

# Non-constructive statements

In classical logic we can prove certain ‘non-constructive’ statements:

- $P \vee (\neg P)$  (excluded middle)
- $\neg\neg P \Rightarrow P$  (double negation elimination)

However, Agda uses a **constructive logic**: a proof of  $A \vee B$  gives us a **decision procedure** to tell whether  $A$  or  $B$  holds.

When  $P$  is unknown, it’s impossible to decide whether  $P$  or  $\neg P$  holds, so the excluded middle is **unprovable** in Agda.

# From classical to constructive logic

Consider the proposition  $P$  (“ $P$  is true”) vs.  $\neg\neg P$  (“It would be absurd if  $P$  were false”).

Classical logic can't tell the difference between the two, but constructive logic can.

**Theorem.**  $P$  is provable in classical logic if and only if  $\neg\neg P$  is provable in constructive logic.  
(proof by Gödel and Gentzen)

# Predicate logic

---

# Curry-Howard beyond simple types

*“Every good idea will be discovered twice:  
once by a logician and once by a computer  
scientist.”*

– Philip Wadler

# Curry-Howard beyond simple types

- Classical logic corresponds to  
**continuations** (e.g. Lisp)

*“Every good idea will be discovered twice:  
once by a logician and once by a computer  
scientist.”*

– Philip Wadler

# Curry-Howard beyond simple types

- Classical logic corresponds to **continuations** (e.g. Lisp)
- Linear logic corresponds to **linear types** (e.g. Rust)

*“Every good idea will be discovered twice:  
once by a logician and once by a computer  
scientist.”*

– Philip Wadler

# Curry-Howard beyond simple types

- Classical logic corresponds to **continuations** (e.g. Lisp)
- Linear logic corresponds to **linear types** (e.g. Rust)
- Predicate logic corresponds to **dependent types** (e.g. Agda)

*“Every good idea will be discovered twice:  
once by a logician and once by a computer  
scientist.”*

– Philip Wadler

# Proving things about programs

So far, we have encoded logical propositions as types and proofs as programs of these types, but there is no interaction yet between the ‘program part’ and the ‘proof part’ of Agda.

**Question.** Can we use the ‘proof part’ to **prove things about** the ‘program part’?

# Proving things about programs

So far, we have encoded logical propositions as types and proofs as programs of these types, but there is no interaction yet between the ‘program part’ and the ‘proof part’ of Agda.

**Question.** Can we use the ‘proof part’ to **prove things about** the ‘program part’?

**Answer.** Yes, we can define propositions that depend on (the output of) programs by using dependent types!

# Defining predicates

**Question.** How would you define a type that expresses that a given number  $n$  is even?

# Defining predicates

**Question.** How would you define a type that expresses that a given number  $n$  is even?

```
data IsEven : Nat → Set where
```

```
  e-zero : IsEven zero
```

```
  e-suc2 : {n : Nat} →
```

```
    IsEven n → IsEven (suc (suc n))
```

```
6-is-even : IsEven 6
```

```
6-is-even = e-suc2 (e-suc2 (e-suc2 e-zero))
```

```
7-is-not-even : IsEven 7 → ⊥
```

```
7-is-not-even = e-suc2 (e-suc2 (e-suc2 ())))
```

# Defining predicates

To define a predicate  $P$  on elements of type  $A$ , we can define  $P$  as a **dependent datatype** with base type  $A$ :

```
data P : A → Set where
```

```
  C1 : ⋯ → P a1
```

```
  C2 : ⋯ → P a2
```

```
  ⋮
```

# A predicate for being true

We can define a predicate `IsTrue` that allows us to use boolean functions as predicates.

```
data IsTrue : Bool → Set where  
  is-true : IsTrue true
```

- If  $b = \text{true}$ , then `IsTrue b` has exactly one element `is-true`
- If  $b = \text{false}$ , then `IsTrue b` has no elements: it is an empty type

# Using the `IsTrue` predicate

`_=Nat_` : `Nat → Nat → Bool`

`zero =Nat zero = true`

`(suc x) =Nat (suc y) = x =Nat y`

`_ =Nat _ = false`

`length-is-3 : IsTrue (length (1 :: 2 :: 3 :: [])) =Nat 3)`

`length-is-3 = is-true`

# Universal quantification

What do we know about the proposition  
 $\forall(x \in A). P(x)$  ('for all  $x$  in  $A$ ,  $P(x)$  holds')?

- To prove  $\forall(x \in A). P(x)$ , we assume we have an unknown  $x \in A$  and prove that  $P(x)$  holds.
- If we have a proof of  $\forall(x \in A). P(x)$  and a concrete  $a \in A$ , then we know  $P(a)$ .

# Universal quantification

What do we know about the proposition  
 $\forall(x \in A). P(x)$  ('for all  $x$  in  $A$ ,  $P(x)$  holds')?

- To prove  $\forall(x \in A). P(x)$ , we assume we have an unknown  $x \in A$  and prove that  $P(x)$  holds.
- If we have a proof of  $\forall(x \in A). P(x)$  and a concrete  $a \in A$ , then we know  $P(a)$ .

$\Rightarrow \forall(x \in A). P(x)$  corresponds to the **dependent function type**  $(x : A) \rightarrow P x$ .

# Universal quantification

**Example.** We can state and prove that for any number  $n : \text{Nat}$ , double  $n$  is even:

`double : Nat → Nat`

`double zero = zero`

`double (suc m) = suc (suc (double m))`

`double-even : (n : Nat) → IsEven (double n)`

`double-even n = {! !}`

# Universal quantification

**Example.** We can state and prove that for any number  $n : \text{Nat}$ , double  $n$  is even:

$\text{double} : \text{Nat} \rightarrow \text{Nat}$

$\text{double zero} = \text{zero}$

$\text{double}(\text{suc } m) = \text{suc}(\text{suc}(\text{double } m))$

$\text{double-even} : (n : \text{Nat}) \rightarrow \text{IsEven}(\text{double } n)$

$\text{double-even zero} = \boxed{\text{! !}}$

$\text{double-even}(\text{suc } m) = \boxed{\text{! !}}$

# Universal quantification

**Example.** We can state and prove that for any number  $n : \text{Nat}$ , double  $n$  is even:

`double : Nat → Nat`

`double zero = zero`

`double (suc m) = suc (suc (double m))`

`double-even : (n : Nat) → IsEven (double n)`

`double-even zero = e-zero`

`double-even (suc m) = {! !}`

# Universal quantification

**Example.** We can state and prove that for any number  $n : \text{Nat}$ , double  $n$  is even:

`double : Nat → Nat`

`double zero = zero`

`double (suc m) = suc (suc (double m))`

`double-even : (n : Nat) → IsEven (double n)`

`double-even zero = e-zero`

`double-even (suc m) = e-suc2 {! !}`

# Universal quantification

**Example.** We can state and prove that for any number  $n : \text{Nat}$ , double  $n$  is even:

`double : Nat → Nat`

`double zero = zero`

`double (suc m) = suc (suc (double m))`

`double-even : (n : Nat) → IsEven (double n)`

`double-even zero = e-zero`

`double-even (suc m) = e-suc2 (double-even m)`

# **Existential quantification**

---

# Existential quantification

What do we know about existential quantification  $\exists(x \in A). P(x)$  (“there exists  $x \in A$  such that  $P(x)$ ”)?

- To prove  $\exists(x \in A). P(x)$ , we need to provide some  $v \in A$  and a proof of  $P(v)$ .
- From a proof of  $\exists(x \in A). P(x)$ , we can get some  $v \in A$  and a proof of  $P(v)$ .

$\Rightarrow$  The proposition  $\exists(x \in A). P(x)$  corresponds to the type of pairs  $(v, p)$  where the type of  $p$  depends on the value of  $v$ .

# Dependent pairs

The type  $\Sigma^2$  is defined as follows:

```
data  $\Sigma$  (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x →  $\Sigma$  A B
```

Projections from a dependent pair:

```
fst $\Sigma$  : {A : Set}{B : A → Set} →  $\Sigma$  A B → A
fst $\Sigma$  (x , y) = x
```

```
snd $\Sigma$  : {A : Set}{B : A → Set} →
  (z :  $\Sigma$  A B) → B (fst $\Sigma$  z)
snd $\Sigma$  (x , y) = y
```

---

<sup>2</sup>Write \Sigma to enter.

# Proving an existential statement

**Example.** Prove that there exists a number  $n$  such that  $n + n = 12$ :

```
half-a-dozen : Σ Nat (λ n → IsTrue ((n + n) =Nat 12))  
half-a-dozen = 6 , is-true
```

Here the second component **is-true** has type  $\text{IsTrue}((6 + 6) =\text{Nat } 12)$ .

# Induction in Agda

In general, a **proof by induction on natural numbers** in Agda looks like this:

```
proof : (n : Nat) → P n
```

```
proof zero    = ...
```

```
proof (suc n) = ...
```

- **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  $n : P n$** .

# An example: $n$ is equal to itself

Let's prove that any number is equal to itself:

```
n-equals-n : (n : Nat) → IsTrue (n =Nat n)  
n-equals-n n = is-true
```

This code results in an error:

true != n =Nat n of type Bool

**Question.** What did we do wrong?

# An example: $n$ is equal to itself

**Answer.** Since `_=Nat_` is defined by pattern matching and recursion, the proof must do the same:

`n>equals-n : (n : Nat) → IsTrue (n =Nat n)`

`n>equals-n zero = is-true`

`n>equals-n (suc m) = n>equals-n m`

# Proving things about programs

**General rule of thumb:** A proof about a function often follows the same structure as that function:

- To prove something about a function by pattern matching, the proof will also use pattern matching (= **proof by cases**)
- To prove something about a recursive function, the proof will also be recursive (= **proof by induction**)

# On the need for totality

To ensure the proofs we write are correct, we rely on the totality of Agda:

- The coverage checker ensures that a proof by cases covers all cases.
- The termination checker ensures that inductive proofs are well-founded.

# Induction on lists in Agda

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

```
proof : {A : Set} (xs : List A) → P xs
```

```
proof []      = ...
```

```
proof (x :: xs) = ...
```

- **proof []** is the **base case**
- **proof (x :: xs)** is the **inductive case**

The inductive case can use the **induction hypothesis** `proof xs : P xs.`

# Live programming exercise

**Assignment.** Write down the Agda type expressing the statement that for any function  $f$  and list  $xs$ , `length (map f xs)` is equal to `length xs`.

Then, prove it by implementing a function of that type.

# What's next?

**Next lecture:** Equational reasoning about functional programs

To do:

- Read the lecture notes:
  - This lecture: section 3 of Agda lecture notes
  - Next lecture: section 4 of Agda lecture notes
- Do Weblab exercises on Curry-Howard

# Equational Reasoning about Functional Programs

Lecture 12 of CSE 3100  
Functional Programming

---

Jesper Cockx  
Q3 2023-2024  
Technical University Delft



*“Beware of bugs in the  
above code; I have only  
proved it correct, not tried  
it.”*

– Donald Knuth

# Lecture plan

- The identity type
- Equational reasoning in Agda
- Applications of equational reasoning:
  - Proving type class laws
  - Verifying optimizations
  - Verifying compiler correctness

# Recap: the Curry-Howard correspondence



Haskell B. Curry

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

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

# Recap: 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	<b>Either</b> $P Q$	either type
implication	$P \rightarrow Q$	function type
truth	$\top$	unit type
falsity	$\perp$	empty type
universal quantification	$(x : A) \rightarrow P x$	dependent function type
existential quantification	$\Sigma A (\lambda x \rightarrow P x)$	dependent pair type

# **The identity type**

---

# The identity type

The type `IsTrue` encodes the property of being equal to `true : Bool`:

```
data IsTrue : Bool → Set where
  is-true : IsTrue true
```

We can generalize this to the property of two elements of some type  $A$  being equal:

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

# Using the identity type

If  $x$  and  $y$  are equal,  $x \equiv y$  has one constructor  
**refl**:

one-plus-one :  $1 + 1 \equiv 2$

one-plus-one = **refl**

If  $x$  and  $y$  are not equal,  $x \equiv y$  is an empty type:

zero-not-one :  $0 \equiv 1 \rightarrow \perp$

zero-not-one ()

# Application of the identity type: Writing test cases

One use case of the identity type is for writing test cases:

`test1 : length (42 :: []) ≡ 1`

`test1 = refl`

`test2 : length (map (1 + _) (0 :: 1 :: 2 :: [])) ≡ 3`

`test2 = refl`

The test cases are run each time the file is loaded!

# Proving correctness of functions

We can use the identity type to prove the correctness of functional programs.

**Example.** Prove that  $\text{not}(\text{not } b) \equiv b$  for all  $b : \text{Bool}$ :

$\text{not-not} : (b : \text{Bool}) \rightarrow \text{not}(\text{not } b) \equiv b$

$\text{not-not true} = \text{refl}$

$\text{not-not false} = \text{refl}$

# Quiz question

**Question.** What is the type of the Agda expression  $\lambda b \rightarrow (b \equiv \text{true})$ ?

1.  $\text{Bool} \rightarrow \text{Bool}$
2.  $\text{Bool} \rightarrow \text{Set}$
3.  $(b : \text{Bool}) \rightarrow \text{IsTrue } b$
4.  $(b : \text{Bool}) \rightarrow b \equiv \text{true}$

# Pattern matching on `refl`

If we have a proof of  $x \equiv y$  as input, we can **pattern match** on the constructor `refl` to show Agda that  $x$  and  $y$  are equal:

```
castVec : {A : Set} {m n : Nat} →
```

```
  m ≡ n → Vec A m → Vec A n
```

```
castVec refl xs = xs
```

When you pattern match on `refl`, Agda applies **unification** to the two sides of the equality.

# Symmetry of equality

Symmetry states that if  $x$  is equal to  $y$ , then  $y$  is equal to  $x$ :

`sym : {A : Set} {x y : A} → x ≡ y → y ≡ x`

`sym refl = refl`

# Congruence

Congruence states that if  $f : A \rightarrow B$  is a function and  $x$  is equal to  $y$ , then  $f x$  is equal to  $f y$ :

```
cong : {A B : Set} {x y : A} →  
  (f : A → B) → x ≡ y → fx ≡ fy  
cong f refl = refl
```

# **Equational reasoning**

---

# Equational reasoning

In school, we learned how to prove equations by chaining basic equalities:

$$\begin{aligned} & (a + b) (a + b) \\ = & a (a + b) + b (a + b) \\ = & a^2 + ab + ba + b^2 \\ = & a^2 + ab + ab + b^2 \\ = & a^2 + 2ab + b^2 \end{aligned}$$

This style of proving is called **equational reasoning**.

# Equational reasoning about functional programs

Equational reasoning is well suited for proving things about **pure** functions:

```
head (replicate 100 "spam")
= head ("spam" : replicate 99 "spam")
= "spam"
```

Because there are no side effects, everything is explicit in the program itself.

# Equational reasoning in Agda

Consider the following definitions:

```
[_] : {A : Set} → A → List A
```

```
[x] = x :: []
```

```
reverse : {A : Set} → List A → List A
```

```
reverse [] = []
```

```
reverse (x :: xs) = reverse xs ++ [x]
```

**Goal.** Prove that `reverse [x] = [x]`.

# Example ‘on paper’

```
reverse [ x ]
=      { definition of [__] }
      reverse (x :: [])
=      { applying reverse (second clause) }
      reverse [] ++ [ x ]
=      { applying reverse (first clause) }
      [] ++ [ x ]
=      { applying _++_ }
      [ x ]
```

# Example in Agda

```
reverse-singleton : {A : Set} (x : A) → reverse [ x ] ≡ [ x ]  
reverse-singleton x =
```

```
begin
```

```
  reverse [ x ]
```

```
  =⟨⟩ - definition of [__]
```

```
  reverse (x :: [])
```

```
  =⟨⟩ - applying reverse (second clause)
```

```
  reverse [] ++ [ x ]
```

```
  =⟨⟩ - applying reverse (first clause)
```

```
  [] ++ [ x ]
```

```
  =⟨⟩ - applying _++_
```

```
  [ x ]
```

```
end
```

# 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 `=⟨⟩`, 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!

# Behind the scenes

Each proof by equational reasoning can be desugared to `refl` (and `trans`).

## Example.

```
reverse-singleton : {A : Set} (x : A) →  
  reverse [ x ] ≡ [ x ]  
reverse-singleton x = refl
```

However, proofs by equational reasoning are much easier to read and debug.

# **Proof by case analysis and induction**

---

# Equational reasoning + case analysis

We can use equational reasoning in a proof by **case analysis** (i.e. pattern matching):

not-not :  $(b : \text{Bool}) \rightarrow \text{not}(\text{not } b) \equiv b$

not-not **false** =

begin

**not**(**not** **false**)

=⟨⟩ – applying the inner not

**not** **true**

=⟨⟩ – applying not

**false**

end

not-not **true** = {!!} – similar to above

# Equational reasoning + induction

We can use equational reasoning in a proof by induction:

add-n-zero :  $(n : \text{Nat}) \rightarrow n + \text{zero} \equiv n$

add-n-zero zero =  $\{\!\!\}$  – easy exercise

add-n-zero (suc  $n$ ) =

begin

$(\text{suc } n) + \text{zero}$

=⟨ ⟩

– applying +

$\text{suc } (n + \text{zero})$

=⟨ cong suc (add-n-zero  $n$ ) ⟩ – using IH

$\text{suc } n$

end

Here we have to provide an explicit proof that  
 $\text{suc } (n + \text{zero}) = \text{suc } n$  (between the  $=\langle$  and  $\rangle$ ).

# Live coding (live proving?)

**Exercise.** State and prove associativity of addition on natural numbers:

$$x + (y + z) = (x + y) + z$$

**Hint.** If you get stuck, try to work instead backwards from the goal you want to reach!

# **Application 1: Proving type class laws**

---

# Reminder: functor laws

Remember the two **functor laws** from Haskell:

- $\text{fmap id} = \text{id}$
- $\text{fmap}(f . g) = \text{fmap } f . \text{fmap } g$

In Haskell we could only verify these laws by hand for each instance, but in Agda we can **prove** that they hold.

# First functor law for List (base case)

```
map-id : {A : Set} (xs : List A) → map id xs ≡ xs
map-id [] =
begin
  map id []
=⟨⟩ – applying map
[]
end
```

# First functor law for List (inductive case)

map-id ( $x :: xs$ ) =

begin

  map id ( $x :: xs$ )

  =  $\langle \rangle$

– applying map

  id  $x :: \text{map id } xs$

  =  $\langle \rangle$

– applying id

$x :: \text{map id } xs$

  =  $\langle \text{cong}(x :: \_) (\text{map-id } xs) \rangle$  – using IH

$x :: xs$

end

# More live proving

**Exercise.** Prove the second functor law for [List](#).

First, we need to define function composition:<sup>1</sup>

$$\begin{aligned}\underline{\circ} : \{A\ B\ C : \text{Set}\} \rightarrow \\ (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C) \\ f \circ g = \lambda x \rightarrow f(g x)\end{aligned}$$

Now we can prove that

$$\text{map } (f \circ g) x = (\text{map } f \circ \text{map } g) x.$$

---

<sup>1</sup>Unicode input for  $\circ$ : \circ

# **Application 2: Verifying optimizations**

---

# Reminder: working with accumulators

A slow version of `reverse` in  $O(n^2)$ :

```
reverse : {A : Set} → List A → List A
```

```
reverse [] = []
```

```
reverse (x :: xs) = reverse xs ++ [ x ]
```

A faster version of `reverse` in  $O(n)$ :

```
reverse-acc : {A : Set} → List A → List A → List A
```

```
reverse-acc [] ys = ys
```

```
reverse-acc (x :: xs) ys = reverse-acc xs (x :: ys)
```

```
reverse' : {A : Set} → List A → List A
```

```
reverse' xs = reverse-acc xs []
```

How can we be sure they are equivalent? **By proving it!**

# Equivalence of reverse and reverse'

```
reverse'-reverse : {A : Set} →  
  (xs : List A) → reverse' xs ≡ reverse xs  
reverse'-reverse xs =  
begin  
  reverse' xs  
  =⟨⟩          - def of reverse'  
  reverse-acc xs []  
  =⟨ reverse-acc-lemma xs [] ⟩ - (see next slide)  
  reverse xs ++ []  
  =⟨ append-[] (reverse xs) ⟩ - using append- []  
  reverse xs  
end
```

# Proving the lemma (base case)

```
reverse-acc-lemma : {A : Set} → (xs ys : List A)
  → reverse-acc xs ys ≡ reverse xs ++ ys
reverse-acc-lemma [] ys =
begin
  reverse-acc [] ys
  =⟨⟩ - definition of reverse-acc
    ys
  =⟨⟩ - unapplying ++
    [] ++ ys
  =⟨⟩ - unapplying reverse
    reverse [] ++ ys
end
```

# Proving the lemma (inductive case)

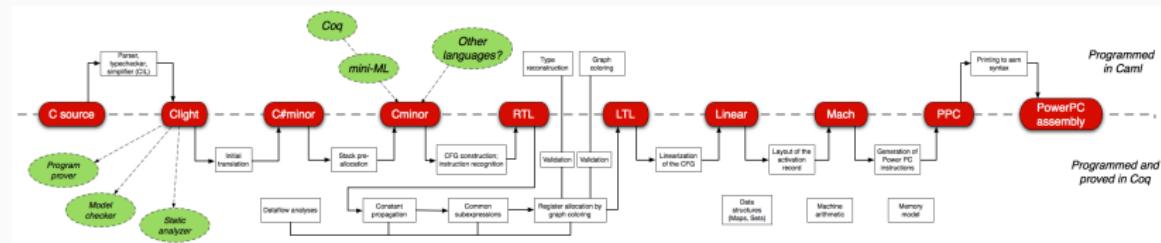
```
reverse-acc-lemma (x :: xs) ys =
begin
  reverse-acc (x :: xs) ys
=⟨⟩                                - def of reverse-acc
  reverse-acc xs (x :: ys)
=⟨ reverse-acc-lemma xs (x :: ys) ⟩
  reverse xs ++ (x :: ys)      - ^ using IH
=⟨⟩                                - unapplying ++
  reverse xs ++ ([ x ] ++ ys)
=⟨ sym (append-assoc (reverse xs) [ x ] ys) ⟩
  (reverse xs ++ [ x ]) ++ ys - ^ associativity of ++
=⟨⟩                                - unapplying reverse
  reverse (x :: xs) ++ ys
end
```

# **Application 3: Proving compiler correctness**

---

# Real-world application: The CompCert C compiler

CompCert is an optimizing compiler for C code, which is **formally proven to be correct** according to the semantics of the C language, using the dependently typed language Coq.



To learn more: <https://compcert.org/>

# A simple expression language

```
data Expr : Set where
```

```
  valE : Nat → Expr
```

```
  addE : Expr → Expr → Expr
```

– Example expr:  $(2 + 3) + 4$

```
expr : Expr
```

```
expr = addE (addE (valE 2) (valE 3)) (valE 4)
```

```
eval : Expr → Nat
```

```
eval (valE x)      = x
```

```
eval (addE e1 e2) = eval e1 + eval e2
```

# Evaluating expressions using a stack

```
data Op : Set where
```

```
  PUSH : Nat → Op
```

```
  ADD  : Op
```

```
Stack = List Nat
```

```
Code  = List Op
```

– Example code for  $(2 + 3) + 4$

```
code : Code
```

```
code = PUSH 2 :: PUSH 3 :: ADD  
      :: PUSH 4 :: ADD :: []
```

# Executing compiled code

Given a list of instructions and an initial stack,  
we can execute the code:

`exec : Code → Stack → Stack`

`exec []`                   `s`                   `= s`

`exec (PUSH x :: c) s`                   `= exec c (x :: s)`

`exec (ADD :: c) (m :: n :: s)`                   `= exec c (n + m :: s)`

`exec (ADD :: c) _`                   `= []`

# Compiling expressions

**Goal.** Compile an expression to a list of stack instructions.

**A first attempt.**

`comp : Expr → Code`

`comp (valE x) = [ PUSH x ]`

`comp (addE e1 e2) =`

`comp e1 ++ comp e2 ++ [ ADD ]`

**Problem.** This is very inefficient ( $O(n^2)$ ) due to the repeated use of `_++_`!

# Compiling with an accumulator

**Problem.** This is very inefficient ( $O(n^2)$ ) due to the repeated use of `_++_`!

Instead, we can use an **accumulator** for the already generated code:

`comp' : Expr → Code → Code`

`comp' (valE x) c = PUSH x :: c`

`comp' (addE e1 e2) c =`

`comp' e1 (comp' e2 (ADD :: c))`

`comp : Expr → Code`

`comp e = comp' e []`

# Proving correctness of `comp`

We want to prove that executing the compiled code has the same result as evaluating the expression directly:

```
comp-exec-eval : (e : Expr) → exec (comp e) [] ≡ [ eval e ]  
comp-exec-eval e =  
begin  
  exec (comp e) []  
  =⟨ comp'-exec-eval e [] [] ⟩ - (see next slide)  
  exec [] (eval e :: [])  
  =⟨ ⟧ - applying exec for []  
  eval e :: []  
  =⟨ ⟧ - unapplying []  
  [ eval e ]  
end
```

# Proving correctness of `comp'` (`valE` case)

`comp'-exec-eval : (e : Expr) (s : Stack) (c : Code)`

$\rightarrow \text{exec} (\text{comp}' e c) s \equiv \text{exec } c (\text{eval } e :: s)$

`comp'-exec-eval (valE x) s c =`

`begin`

`exec (comp' (valE x) c) s`

$= \langle \rangle$  – applying `comp'`

`exec (PUSH x :: c) s`

$= \langle \rangle$  – applying `exec` for `PUSH`

`exec c (x :: s)`

$= \langle \rangle$  – unapplying `eval` for `valE`

`exec c (eval (valE x) :: s)`

`end`

# Proving correctness of comp' (addE case)

```
comp'-exec-eval (addE e1 e2) s c =
begin
  exec (comp' (addE e1 e2) c) s
=⟨⟩ - def of comp'
  exec (comp' e1 (comp' e2 (ADD :: c))) s
=⟨ comp'-exec-eval e1 s (comp' e2 (ADD :: c)) ⟩ - IH
  exec (comp' e2 (ADD :: c)) (eval e1 :: s)
=⟨ comp'-exec-eval e2 (eval e1 :: s) (ADD :: c) ⟩ - IH
  exec (ADD :: c) (eval e2 :: eval e1 :: s)
=⟨⟩ - applying exec for ADD
  exec c (eval e1 + eval e2 :: s)
=⟨⟩ - unapplying eval for addE
  exec c (eval (addE e1 e2) :: s)
end
```

# Equational reasoning: summary

Equational reasoning is a simple but powerful technique to reason about pure functional programs.

We can write Agda proofs in equational reasoning style by using the combinators `begin`, `end`, `_ =⟨⟩_`, and `_ =⟨_⟩_`.

Equational reasoning combines well with case analysis (= pattern matching) and induction (= recursion).

# What's next?

**Final lecture:** Course recap + preparation for exam

**To do:**

- Read the lecture notes:
  - This lecture: section 4 of Agda lecture notes
- Do exercises on equational reasoning in Weblab
- Send me topics or questions for the final lecture!

# 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<sup>1</sup>

**Laziness** Expressions are only evaluated when required

**Purity** Functions do not have side effects

---

<sup>1</sup>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:

$$f = \text{foldr} \ (\#) \ v$$

# 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

Without do-notation

do

x <- f

f >>= (\x ->

g x

g x >> (

y <- h x

h x >>= (\y ->

return (p x y)

return (p x y)

)

)

)

# Lecture 7: Monadic parsing

```
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:

$b : \text{Bool}$  instead of  $b :: \text{Bool}$ .

**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  $\mathbf{1 + 1}$  instead.

**Infix operators** are indicated by underscores:  
 $\mathbf{\_+\_}$  instead of  $(+)$

# Lecture 9: Types as first-class values

In Agda, types such as `Nat` and `(Bool → Bool)` are themselves expressions of type `Set`.

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

`id : (A : Set) → A → A`

`id A x = x`

# 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.

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

# Lecture 10: A safe lookup

`lookupVec : {A : Set} {n : Nat}`

$\rightarrow \text{Vec } A \ n \rightarrow \text{Fin } n \rightarrow A$

`lookupVec (x :: xs) zero = x`

`lookupVec (x :: xs) (suc i) = lookupVec xs i`

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 ( $A \wedge B$ ,  $\neg A$ ,  $A \Rightarrow B$ , ...) 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	<b>Either</b> $P Q$	either type
implication	$P \rightarrow Q$	function type
truth	$\top$	unit type
falsity	$\perp$	empty type
universal quantification	$(x : A) \rightarrow P x$	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:

```
proof : (n : Nat) → P n
```

```
proof zero    = ...
```

```
proof (suc n) = ...
```

- **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:

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

Transitivity:

$$\begin{aligned} \text{trans} : & \{A : \text{Set}\} \{x y z : A\} \\ & \rightarrow x \equiv y \rightarrow y \equiv z \rightarrow x \equiv z \end{aligned}$$

Congruence:

$$\begin{aligned} \text{cong} : & \{A B : \text{Set}\} \{x y : A\} \\ & \rightarrow (f : A \rightarrow B) \rightarrow x \equiv y \rightarrow fx \equiv fy \end{aligned}$$

# 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 `=⟨⟩` or `=⟨ proof ⟩`, 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.<sup>2</sup>

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

---

<sup>2</sup>No Haskell plugin, sorry.

# Official course survey



[https://evasys-survey.tudelft.nl/  
evasys/online.php?p=744SX](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<sup>3</sup> (10-16 May)
- Summer School on Advanced Functional Programming in Utrecht<sup>4</sup> (3-7 July)

---

<sup>3</sup><https://wiki.portal.chalmers.se/agda/Main/AIMXXXVI>

<sup>4</sup><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!