Functional Programming

ADT

Daniil Berezun

danya.berezun@gmail.com

2022

Before we start...

you can and **should** interrupt me and **ask** **any** **questions**

Functional programming

Functional programming

Functional programming

a *style* of programming in which basic method of computation is function application

```
int counter = 0;
for (int i = 0; i < n; i++)
  counter += i;
```

variable assignment

```
sum [1..n]

fold (+) [1..n] 0
```

- function application
- declarative
- efficiency — compiler’s job

Functional programming

a *style* of programming in which basic method of computation is function application

```
int counter = 0;
for (int i = 0; i < n; i++)
  counter += i;
```

variable assignment

```
sum [1..n]

fold (+) [1..n] 0
```

- function application
- declarative
- efficiency — compiler's job

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

Types

What is a *type*?

Types

What is a *type*?

a *collection* of its values

Example: `Bool = True "+" False`

Strongly Statically Typed

- Safer
- Faster since no type checking in runtime
- Specifies function behaviour!

Specifies function behaviour

- `Int -> Bool`
- `Int -> Int -> Int`

Basic types

- `Bool`
- `Int`,`Integer`
- `Float`,`Double`
- `f :: a -> a`
- `[a]`
- `Char, String = [Char]`
- `:type <Expr>`

Types and Primitive Data

Basic types

- `Bool`
- `Int`,`Integer`
- `Float`,`Double`
- `f :: a -> a`
- `[a]`
- `Char`, `String = [Char]`
- `:type <Expr>`

Tuples

```
(42,"Hello!") :: (Int,String)

(True,'r',12312.123123)
```

- min tuple size: 2
- max tuple size: ≥ 15
  (62 in ghc)

Basic types

- Bool
- Int, Integer
- Float, Double
- f :: a -> a
- [a]
- Char, String = [Char]
- :type <Expr>

Tuples

```
(42,"Hello!") :: (Int,String)

(True,'r',12312.123123)
```

- min tuple size: 2
- max tuple size: ≥ 15 (62 in ghc)

Lists

```
[]
[1,2,3,4,9]
1 : [] ≡ [1]
ghci> ['H', 'e', 'l', 'l', 'o']
"Hello"
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> let b = [[1,2,3,4],[3,4,5,6,7]]
ghci> b
[[1,2,3,4],[3,4,5,6,7]]
ghci> b ++ [[1,1,1]]
[[1,2,3,4],[3,4,5,6,7],[1,1,1]]
ghci> head [5,4,3,2,1]
5
ghci> tail [5,4,3,2,1]
[4,3,2,1]
ghci> head []
*** Exception: Prelude.head: empty list
```

Lists provides no info about size, tuples does!

Make your own types: Algebraic Data Types

- a-la enum + structures in C
- ADT = union + product + exponential types

Trivial union types: enums (0-arity)

```
--      [1]      [2] [3] [2][3] [2] [3] [2]
data Cardinal = North | East | South | West
-- [1]: Type constructor
-- [2]: Data constructor
-- [3]: The pipe operator that separates data constructors
```

Make your own types: Algebraic Data Types

- a-la enum + structures in C
- ADT = union + product + exponential types

Trivial union types: enums (0-arity)

```
--       [1]       [2] [3] [2][3] [2] [3] [2]
data Cardinal = North | East | South | West
-- [1]: Type constructor
-- [2]: Data constructor
-- [3]: The pipe operator that separates data constructors
```

Pattern Matching

```
hasPole :: Cardinal -> Bool
hasPole x =
  if (x == North) || (x == South)
  then True
  else False
```

- a-la enum + structures in C
- ADT = union + product + exponential types

Trivial union types: enums (0-arity)

```haskell
--       [1]        [2]  [3]  [2][3]  [2]  [3]  [2]
data Cardinal = North | East | South | West
-- [1]: Type constructor
-- [2]: Data constructor
-- [3]: The pipe operator that separates data constructors
```

Pattern Matching

```haskell
hasPole :: Cardinal -> Bool
hasPole x =
  if (x == North) || (x == South)
  then True
  else False
```

```haskell
hasPole North = True
hasPole South = True
hasPole _     = False
```

- a-la enum + structures in C
- ADT = union + product + exponential types

Trivial union types: enums (0-arity)

```haskell
--       [1]       [2] [3] [2][3] [2] [3] [2]
data Cardinal = North | East | South | West
-- [1]: Type constructor
-- [2]: Data constructor
-- [3]: The pipe operator that separates data constructors
```

Pattern Matching

```haskell
hasPole :: Cardinal -> Bool
hasPole x =
  if (x == North) || (x == South)
  then True
  else False
```

```haskell
hasPole North = True
hasPole South = True
hasPole _     = False

hasPole x = case x of
  North -> True
  South -> True
  _     -> False
```

Make your own types: Algebraic Data Types

- a-la enum + structures in C
- ADT = union + product + exponential types

Trivial union types: enums (0-arity)

```haskell
--      [1]        [2] [3] [2][3] [2] [3] [2]
data Cardinal = North | East | South | West
-- [1]: Type constructor
-- [2]: Data constructor
-- [3]: The pipe operator that separates data constructors
```

Pattern Matching

```haskell
hasPole :: Cardinal -> Bool
hasPole x =
  if (x == North) || (x == South)
  then True
  else False
```

```haskell
hasPole x = x `elem` [South, North]
```

```haskell
hasPole North = True
hasPole South = True
hasPole _     = False

hasPole x = case x of
  North -> True
  South -> True
  _     -> False
```

Make your own types: Algebraic Data Types

- a-la enum + structures in C
- ADT = union + product + exponential types

Trivial union types: enums (0-arity)

```
--       [1]       [2] [3] [2][3] [2] [3] [2]
data Cardinal = North | East | South | West deriving Eq
-- [1]: Type constructor
-- [2]: Data constructor
-- [3]: The pipe operator that separates data constructors
```

Pattern Matching

```
hasPole :: Cardinal -> Bool
hasPole x =
  if (x == North) || (x == South)
  then True
  else False
```

```
hasPole x = x `elem` [South, North]
```

↑ **Instance of Eq**

```
hasPole North = True
hasPole South = True
hasPole _     = False

hasPole x = case x of
  North -> True
  South -> True
  _     -> False
```

Simple Product Type

```haskell
--    [1]      [2]          [3]
data Point = Point Double Double
-- [1]: Type constructor.
-- [2]: Data constructor.
-- [3]: Types wrapped.
```

Simple Product Type

```
--     [1]     [2]        [3]
data Point = Point Double Double
-- [1]: Type constructor.
-- [2]: Data constructor.
-- [3]: Types wrapped.
ghci> :type Point
Point :: Double -> Double -> Point
ghci> a = Point 3 4
ghci> a
Point 3.0 4.0
ghci> a = Point 3 4
ghci> b = Point 1 2
```

Simple Product Type

```
--     [1]      [2]        [3]
data Point = Point Double Double
-- [1]: Type constructor.
-- [2]: Data constructor.
-- [3]: Types wrapped.
ghci> :type Point
Point :: Double -> Double -> Point
ghci> a = Point 3 4
ghci> a
Point 3.0 4.0
ghci> a = Point 3 4
ghci> b = Point 1 2

dist (Point x1 y1) (Point x2 y2) =
  sqrt ((x1 - x2)^2 + (y1 - y2)^2)

ghci> dist a b
2.8284271247461903
```

Product types

Simple Product Type

```
--    [1]      [2]         [3]
data Point = Point Double Double
-- [1]: Type constructor.
-- [2]: Data constructor.
-- [3]: Types wrapped.
ghci> :type Point
Point :: Double -> Double -> Point
ghci> a = Point 3 4
ghci> a
Point 3.0 4.0
ghci> a = Point 3 4
ghci> b = Point 1 2

dist (Point x1 y1) (Point x2 y2) =
  sqrt ((x1 - x2)^2 + (y1 - y2)^2)

ghci> dist a b
2.8284271247461903
```

Polymorphic Data Types

```
data PPoint a = PPoint a a
distP (PPoint x1 y1) (PPoint x2 y2) =
  sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

Product types

Simple Product Type

```haskell
--     [1]      [2]         [3]
data Point = Point Double Double
-- [1]: Type constructor.
-- [2]: Data constructor.
-- [3]: Types wrapped.
ghci> :type Point
Point :: Double -> Double -> Point
ghci> a = Point 3 4
ghci> a
Point 3.0 4.0
ghci> a = Point 3 4
ghci> b = Point 1 2

dist (Point x1 y1) (Point x2 y2) =
  sqrt ((x1 - x2)^2 + (y1 - y2)^2)

ghci> dist a b
2.8284271247461903
```

Polymorphic Data Types

```haskell
data PPoint a = PPoint a a
distP (PPoint x1 y1) (PPoint x2 y2) =
  sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

```haskell
ghci> :kind Point
Point :: *
ghci> :kind PPoint
PPoint :: * -> *
ghci> :info (,)
type (,) :: * -> * -> *
data (,) a b = (,) a b
ghci> :t distP
distP :: Floating a => PPoint a
  -> PPoint a -> a
```

Union Types

```haskell
data Point = Point2D Double Double | Point3D Double Double Double

pointToList :: Point -> [Double]
pointToList (Point2D x y) = [x, y]
pointToList (Point3D x y z) = [x, y, z]
*Main> a = Point2D 3 4
*Main> b = Point3D 3 4 5
*Main> pointToList a
[3.0,4.0]
*Main> pointToList b
[3.0,4.0,5.0]
```

	Product types	Sum types
Example	`data (,) a b = (,) a b`	`data Bool = False | True`
Intuition	a **and** b	a **or** b

Built-in types *behaves* like ADTs

```
data Char = '\NUL' | ... | 'a'
  | 'b' | 'c' | 'd' | ...
  | '\1114111'
data Int = -9223372036854775808 |...
  | -2 | -1 | 0 | 1 | 2 | ...
  | 9223372036854775807
data Integer = ... | -2 | -1 | 0
  | 1 | 2 | ...

isAnswer :: Integer -> Bool
isAnswer 42 = True
isAnswer _  = False
```

More on Pattern Matching

Built-in types ***behaves*** like ADTs

```
data Char = '\NUL' | ... | 'a'
  | 'b' | 'c' | 'd' | ...
  | '\1114111'
data Int = -9223372036854775808 |...
  | -2 | -1 | 0 | 1 | 2 | ...
  | 9223372036854775807
data Integer = ... | -2 | -1 | 0
  | 1 | 2 | ...

isAnswer :: Integer -> Bool
isAnswer 42 = True
isAnswer _  = False
```

Pattern Matching Semantics

```
bar 1 2 = 3
bar 0 _ = 5

bar 0 7
bar 2 1
bar 1 (5-3)
bar 1 undefined
bar 0 undefined
```

More on Pattern Matching

Built-in types *behaves* like ADTs

```
data Char = '\NUL' | ... | 'a'
  | 'b' | 'c' | 'd' | ...
  | '\1114111'
data Int = -9223372036854775808 |...
  | -2 | -1 | 0 | 1 | 2 | ...
  | 9223372036854775807
data Integer = ... | -2 | -1 | 0
  | 1 | 2 | ...

isAnswer :: Integer -> Bool
isAnswer 42 = True
isAnswer _  = False
```

Pattern Matching Semantics

```
bar 1 2 = 3
bar 0 _ = 5

bar 0 7 -- fail, success
bar 2 1 -- fail, fail
bar 1 (5-3) -- success
bar 1 undefined -- diverge
bar 0 undefined -- success
```

More on Pattern Matching

Built-in types *behaves* like ADTs

```
data Char = '\NUL' | ... | 'a'
  | 'b' | 'c' | 'd' | ...
  | '\1114111'
data Int = -9223372036854775808 |...
  | -2 | -1 | 0 | 1 | 2 | ...
  | 9223372036854775807
data Integer = ... | -2 | -1 | 0
  | 1 | 2 | ...

isAnswer :: Integer -> Bool
isAnswer 42 = True
isAnswer _  = False
```

Pattern Matching Semantics

```
bar 1 2 = 3
bar 0 _ = 5

bar 0 7 -- fail, success
bar 2 1 -- fail, fail
bar 1 (5-3) -- success
bar 1 undefined -- diverge
bar 0 undefined -- success
```

Exhaustive?

```
repl :: String -> String
repl " " = " "
repl (x:xs) = x:x:repl xs
ghci> repl "a"
"aa*** Exception: ...
Non-exhaustive patterns ...

-fwarn-incomplete-patterns (-W, -Wall)
```

Maybe

```
ghci> :info Maybe
type Maybe :: * -> *
data Maybe a = Nothing | Just a

ghci> head []
*** Exception: Prelude.head: empty
  list
```

Maybe

```
ghci> :info Maybe
type Maybe :: * -> *
data Maybe a = Nothing | Just a

ghci> head []
*** Exception: Prelude.head: empty
  list

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

ghci> safeHead []
Nothing
ghci> safeHead [1, 2, 3]
Just 1
```

Standard ADTs

Maybe

```
ghci> :info Maybe
type Maybe :: * -> *
data Maybe a = Nothing | Just a

ghci> head []
*** Exception: Prelude.head: empty
  list

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

ghci> safeHead []
Nothing
ghci> safeHead [1, 2, 3]
Just 1
```

Either

```
ghci> :info Either
type Either :: * -> * -> *
data Either a b = Left a | Right b

safeHead :: [a] -> Either String a
safeHead []    =
  Left "safeHead: empty list"
safeHead (x:_) = Right x

ghci> safeHead []
"safeHead: empty list"
ghci> safeHead [1, 2, 3]
Right 1
```

- **Distinctness**

$$\forall j \neq i . C_i(x) \neq C_j(y)$$

- Injectivity

$$C_{ij}(x_1, \ldots, x_{n_{ij}} = C_{ij}(y_1, \ldots, y_{n_{ij}}) \Rightarrow \forall k . x_k = y_k$$

- Exhaustiveness

$$x \text{ of some ADT } \Rightarrow \exists i, n . x = C_i(y_1, \ldots, y_n)$$

- Selection

$$\exists s_i^k : s_i^k(C_k(x_{k_1}, \ldots, x_{k_n})) = x_{k_i}$$

Distinctness

- Start from different constructors ⇒ different values

```
neqLists []     (_:_)   = True
neqLists (_:_)  []      = True
neqLists (x:xs) (y:ys) =
  (x /= y) || (neqLists xs ys)
neqLists []     []      = False
```

- Distinctness

$$\forall j \neq i. C_i(x) \neq C_j(y)$$

- **Injectivity**

$$C_{ij}(x_1, \ldots, x_{n_{ij}}) = C_{ij}(y_1, \ldots, y_{n_{ij}}) \Rightarrow \forall k. x_k = y_k$$

- Exhaustiveness

$$x \text{ of some ADT } \Rightarrow \exists i, n. x = C_i(y_1, \ldots, y_n)$$

- Selection

$$\exists s_i^k : s_i^k(C_k(x_{k_1}, \ldots, x_{k_n})) = x_{k_i}$$

Injectivity

- Same values ⇒ start from the same constructor, and arguments are equal pairwise

```
eqLists []       []       = True
eqLists []       (_:_)    = False
eqLists (_:_)    []       = False
eqLists (x:xs) (y:ys) =
  (x == y) && eqLists xs ys
```

- Distinctness

$$\forall j \neq i. C_i(x) \neq C_j(y)$$

- Injectivity

$$C_{ij}(x_1, \ldots, x_{n_{ij}}) = C_{ij}(y_1, \ldots, y_{n_{ij}}) \Rightarrow \forall k. x_k = y_k$$

- **Exhaustiveness**

$$x \text{ of some ADT } \Rightarrow \exists i, n. x = C_i(y_1, \ldots, y_n)$$

- Selection

$$\exists s_i^k : s_i^k(C_k(x_{k_1}, \ldots, x_{k_n})) = x_{k_i}$$

Exhaustiveness

- Values of some ADT starts from one of constructors listed in the type definition only

```
evenLength [] = True
evenLength (_:_:tl) = evenLength tl

ghci>:set -fwarn-incomplete-patterns
...: warning: [-Wincomplete-patterns]
  Pattern match(es) are non-exhaustive
  In an equation for ‘evenLength’:
    Patterns not matched: [_]
 |
 | evenLength [] = True
 | ^^^^^^^^^^^^^^^^^^^^...
```

Formal ADT Axioms

- Distinctness

$$\forall j \neq i. C_i(x) \neq C_j(y)$$

- Injectivity

$$C_{ij}(x_1, \ldots, x_{n_{ij}}) = C_{ij}(y_1, \ldots, y_{n_{ij}}) \Rightarrow \forall k. x_k = y_k$$

- Exhaustiveness

$$x \text{ of some ADT } \Rightarrow \exists i, n. x = C_i(y_1, \ldots, y_n)$$

- **Selection**

$$\exists s_i^k : s_i^k(C_k(x_{k_1}, \ldots, x_{k_n})) = x_{k_i}$$

Selection

- One can select an (sub)-element via pattern-matching

```
somefunct [] =
  -- no argument of empty list
  ...
somefunct (h:tl) =
 ... h ... tl ... h ...
```

ADT definition can be recursive

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

Nil :: [a]
Cons 1 Nil :: [Int]
Cons 2 (Cons 1 Nil) :: [Int]
```

type and newtype

```haskell
type FirstName1 = String

newtype FirstName2 = FirstName2 String
```

Recursive types

ADT definition can be recursive

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

Nil :: [a]
Cons 1 Nil :: [Int]
Cons 2 (Cons 1 Nil) :: [Int]
```

Have **same** constructors

New constructor

type and newtype

```haskell
type FirstName1 = String

newtype FirstName2 = FirstName2 String
```

Recursive types

ADT definition can be recursive

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

Nil :: [a]
Cons 1 Nil :: [Int]
Cons 2 (Cons 1 Nil) :: [Int]
```

Have **same** constructors

New constructor

type and newtype

```
type FirstName1 = String

newtype FirstName2 = FirstName2 String
```

Example

```
unF1 :: FirstName1 -> String
unF1 = id

unF2 :: FirstName2 -> String
unF2 (FirstName2 s) = s

ghci> unF1 "a"
"a"
ghci> unF1 (FirstName2 "a")
"a"
```

Recursive types

ADT definition can be recursive

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

Nil :: [a]
Cons 1 Nil :: [Int]
Cons 2 (Cons 1 Nil) :: [Int]
```

Have **same** constructors

New constructor

type and newtype

```
type FirstName1 = String

newtype FirstName2 = FirstName2 String
```

Example

```
unF1 :: FirstName1 -> String
unF1 = id

unF2 :: FirstName2 -> String
unF2 (FirstName2 s) = s

ghci> unF1 "a"
"a"
ghci> unF1 (FirstName2 "a")
"a"
```

newtype

- Type safety
- Exactly one constructor and one field
- Can't be recursive

Exponential types (functions)

```
data Endom a = Endom (a -> a)
appEndom :: Endom a -> a -> a
appEndom (Endom f) = f

ghci> e = Endom (\n -> 2 * n + 1)
ghci> :t e
e :: Num a => Endom a
```

- $|a \to b| = |a|^{|b|}$
- Functions — first class values

Why it is called Algebraic?

Data.Void	**Void**	0
`()`	`()`	1
Bool	`data Bool = False | True`	1+1
Maybe	`data Maybe a = Nothing | Just a`	1+|a|
Either	`data Either a b = Left a | Right b`	|a|+|b|
Tuple	`(a, b)`	|a| * |b|
Function	`a -> b`	$\|b\|^{\|a\|}$
2D or 3D point	`data Point a = Point2D a a | Point3D a a a`	$\|a\|^2 + \|a\|^3$

Example: Arithmetic Expressions

Variant 1

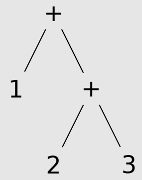
```
data Expr =
   | Const Int
   | VarCalledX
   | Plus     Expr Expr
   | Asterisk Expr Expr
   | Dash     Expr Expr
   | Slash    Expr Expr
```

Variant 2

```
data Op = Plus | Asterisk
   | Dash | Slash
data Expr =
  | Const Int
  | VarCalledX
  | BinOp Op Expr Expr
```

Looks exactly like AST

```
Plus (Const 1,
  Plus (Const 2,
        Const 3))
```



ADT: What can go Wrong?

should be **safe by construction**

```haskell
data Expr = Plus Expr Expr | Const Int | ...
x = Plus (Const 1, Plus (Const 2, Const 3))
                -- vs
data Expr = Plus [Expr] | Const Int | ...
y = Plus [ Const 1, Const 2, Const 3 ]
```

ADT: What can go Wrong?

should be **safe by construction**

```
data Expr = Plus Expr Expr | Const Int | ...
x = Plus (Const 1, Plus (Const 2, Const 3))
               -- vs
data Expr = Plus [Expr] | Const Int | ...
y = Plus [ Const 1, Const 2, Const 3 ]
z = Plus [ ]  -- semantics?
```

ADT: What can go Wrong?

should be safe by construction

```
data Expr = Plus Expr Expr | Const Int | ...
x = Plus (Const 1, Plus (Const 2, Const 3))
                -- vs
data Expr = Plus [Expr] | Const Int | ...
y = Plus [ Const 1, Const 2, Const 3 ]
z = Plus [ ]  -- semantics?
```

data should have a unique representation

```
data Expr =
  | Plus Expr Expr
  | Oper Op [Expr]
  | ...
```

Usual Union Data Definition

```
data SimplePerson = SimplePerson String String String Int String
firstPerson = SimplePerson "Alan" "Smith" "asmith@gmail.com" 42 "Lawyer"
incomplete  = SimplePerson "Michael" "Smith" "msmith@gmail.com" 42
complete    = incomplete "Dancer"
```

Usual Union Data Definition

```haskell
data SimplePerson = SimplePerson String String String Int String
firstPerson = SimplePerson "Alan" "Smith" "asmith@gmail.com" 42 "Lawyer"
incomplete  = SimplePerson "Michael" "Smith" "msmith@gmail.com" 42
complete    = incomplete "Dancer"
```

Records

- are an extension of union ADT that allow fields to be named:

```haskell
data Person = Person
  { age :: Int
  , name :: String
  }
ghci> a = Person 3 "a"
ghci> a
Person {age = 3, name = "a"}
ghci> age a
3
ghci> b = a {name = "BB"} --"update"
ghci> b
Person {age = 3, name = "BB"}
```

```haskell
ghci> grouwUp person =
  person {age = age person + 1}
ghci> c = growUp a
ghci> c
Person {age = 4, name = "a"}
```

Usual Union Data Definition

```haskell
data SimplePerson = SimplePerson String String String Int String
firstPerson = SimplePerson "Alan" "Smith" "asmith@gmail.com" 42 "Lawyer"
incomplete  = SimplePerson "Michael" "Smith" "msmith@gmail.com" 42
complete    = incomplete "Dancer"
```

Records

- are an extension of union ADT that allow fields to be named:

```haskell
data Person = Person
  { age :: Int
  , name :: String
  }
ghci> a = Person 3 "a"
ghci> a
Person {age = 3, name = "a"}
ghci> age a
3
ghci> b = a {name = "BB"} --"update"
ghci> b
Person {age = 3, name = "BB"}
```

```haskell
ghci> grouwUp person =
  person {age = age person + 1}
ghci> c = growUp a
ghci> c
Person {age = 4, name = "a"}
```

RecordWildCards

```haskell
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) =
  map toLower name
```

Records

Usual Union Data Definition

```haskell
data SimplePerson = SimplePerson String String String Int String
firstPerson = SimplePerson "Alan" "Smith" "asmith@gmail.com" 42 "Lawyer"
incomplete  = SimplePerson "Michael" "Smith" "msmith@gmail.com" 42
complete    = incomplete "Dancer"
```

Records

- are an extension of union ADT that allow fields to be named:

```haskell
data Person = Person
  { age :: Int
  , name :: String
  }
ghci> a = Person 3 "a"
ghci> a
Person {age = 3, name = "a"}
ghci> age a
3
ghci> b = a {name = "BB"} --"update"
ghci> b
Person {age = 3, name = "BB"}
```

```haskell
ghci> grouwUp person =
  person {age = age person + 1}
ghci> c = growUp a
ghci> c
Person {age = 4, name = "a"}
```

RecordWildCards

```haskell
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) =
  map toLower name
lowerCaseName (Person { .. }) =
  map toLower name
f (Person {age = 3, ..})= name++"b"
```

More on Records: common field labels

common field labels

```
data Point a = Point2D {xCord :: a, yCord :: a}
| Point3D {xCord :: a, yCord :: a, zCord :: a}
ghci> p1 = Point2D 1.0 1.0
ghci> p2 = Point3D 2.0 2.0 2.0
ghci> xCord p1
1.0
ghci> xCord p2
2.0
```

- Field labels has global scope
- Thus, common labels may be wihtin one data type only

```
data Point1D a = Point1D {xCord :: a}
ghci>
Error: Multiple declarations of 'xCord'.
```

Example: Network package

```haskell
data ConnectionState =
  Connecting | Connected | Disconnected
data ConnectionInfo = ConnectionInfo
  { state :: ConnectionState,
    server :: InetAddr,
    last_ping_time :: Maybe Time,
    last_ping_id :: Maybe Int,
    session_id :: Maybe String,
    when_initiated :: Maybe Time,
    when_disconnected :: Maybe Time
  }
```

Example: Network package

```haskell
data ConnectionState =
  Connecting | Connected | Disconnected
data ConnectionInfo = ConnectionInfo
  { state :: ConnectionState,
    server :: InetAddr,
    last_ping_time :: Maybe Time,
    last_ping_id :: Maybe Int,
    session_id :: Maybe String,
    when_initiated :: Maybe Time,
    when_disconnected :: Maybe Time
  }
```

```haskell
-- better
data ConnectingD = ConnectingD
  { whenInitiated :: Time }
data ConnectedD = ConnectedD
  { lastPing  :: Maybe (Time, Int),
    sessionId :: String }
data DisconnectedD =
  DisconnectedD {
    whenDisconnected :: Time }
```

Example: Network package

```haskell
data ConnectionState =
  Connecting | Connected | Disconnected
data ConnectionInfo = ConnectionInfo
  { state :: ConnectionState,
    server :: InetAddr,
    last_ping_time :: Maybe Time,
    last_ping_id :: Maybe Int,
    session_id :: Maybe String,
    when_initiated :: Maybe Time,
    when_disconnected :: Maybe Time
  }
```

```haskell
-- better
data ConnectingD = ConnectingD
  { whenInitiated :: Time }
data ConnectedD = ConnectedD
  { lastPing  :: Maybe (Time, Int),
    sessionId :: String }
data DisconnectedD =
  DisconnectedD {
    whenDisconnected :: Time }
```

```haskell
-- but ugly
data ConnectionState =
  Connecting ConnectingD
  | Connected ConnectedD
  | Disconnected DisconnectedD
data ConnectionInfo = ConnectionInfo
  { state  :: ConnectionState,
    server :: InetAddr
  }
```

Example: Network package

```haskell
data ConnectionState =
  Connecting | Connected | Disconnected
data ConnectionInfo = ConnectionInfo
  { state :: ConnectionState,
    server :: InetAddr,
    last_ping_time :: Maybe Time,
    last_ping_id :: Maybe Int,
    session_id :: Maybe String,
    when_initiated :: Maybe Time,
    when_disconnected :: Maybe Time
  }
```

```haskell
-- +/- good
data ConnectionState =
  Connecting  { whenInitiated :: Time }
  | Connected { lastPing ::Maybe(Time,Int),
                sessionId::String }
  | Disconnected { whenDisconnected::Time }
data ConnectionInfo = ConnectionInfo
  { state  :: ConnectionState,
    server :: InetAddr
  }
```

```haskell
-- better
data ConnectingD = ConnectingD
  { whenInitiated :: Time }
data ConnectedD = ConnectedD
  { lastPing  :: Maybe (Time, Int),
    sessionId :: String }
data DisconnectedD =
  DisconnectedD {
    whenDisconnected :: Time }
```

```haskell
-- but ugly
data ConnectionState =
  Connecting ConnectingD
  | Connected ConnectedD
  | Disconnected DisconnectedD
data ConnectionInfo = ConnectionInfo
  { state  :: ConnectionState,
    server :: InetAddr
  }
```

Pure function

Pure Functions

Pure function

- Deterministic
  - No use of global mutable data
  - No dependence on date, time, rand and so on
- No side effects
  - No input/output
  - Purely or impurely call impure functions

Pure Functions

Pure function

- Deterministic
  - No use of global mutable data
  - No dependence on date, time, rand and so on
- No side effects
  - No input/output
  - Purely or impurely call impure functions

Remarks

- Is a property of a *function*, not a language
- Easier to optimize: e.g. inlining
- We can compute function call only ones (aka memoization)

Pure Functions

Pure function

- Deterministic
  - No use of global mutable data
  - No dependence on date, time, rand and so on
- No side effects
  - No input/output
  - Purely or impurely call impure functions

Remarks

- Is a property of a *function*, not a language
- Easier to optimize: e.g. inlining
- We can compute function call only ones (aka memoization)

Mathematical function **vs** pure function

- Non-termination
- Crash

	Math	Programming
Always returns a result	function	total function
May not return a result	partial function	function

Datatype Cardinality

A number of *different* terms (values) that populate the type.

Datatype Cardinality

A number of *different* terms (values) that populate the type.

Observational equivalence

Two terms are observationally equivalent if they are indistinguishable (there is no effective procedure to distinguish them) on the basis of their observable implications

Observational Equivalence

Datatype Cardinality

A number of *different* terms (values) that populate the type.

Observational equivalence

Two terms are observationally equivalent if they are indistinguishable (there is no effective procedure to distinguish them) on the basis of their observable implications

Guess type cardinality (pure functions)

```
?> () -> Int
?> Int -> ()
?> Void -> Int
```

```
?> Int -> Void
?> a -> ()
?> Void -> a
```

Guess

What these *pure* functions can do? What about cardinality?

```haskell
? a -> a
? [a] -> [a]
? [a] -> Bool
? (a -> b) -> [a] -> [b]
? (a -> a -> Bool) -> [a] -> [a]
? (a -> a -> Ordering) -> [a] -> [a] where data Ordering = LT | EQ | GT
? [a] -> [b] -> [(a,b)]
? Maybe a -> Maybe b -> (a -> b -> c) -> Maybe c
? Int -> a -> [a]
? [a] -> Int -> a
```

Guess

What these *pure* functions can do? What about cardinality?

```
? a -> a                                                      id x = x
? [a] -> [a]
? [a] -> Bool
```

```
isEmpty [] = True
isEmpty _  = False
```

```
? (a -> b) -> [a] -> [b]
```

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

```
? (a -> a -> Bool) -> [a] -> [a]                              filter
? (a -> a -> Ordering) -> [a] -> [a] where data Ordering = LT | EQ | GT
? [a] -> [b] -> [(a,b)]                                       zip
? Maybe a -> Maybe b -> (a -> b -> c) -> Maybe c
? Int -> a -> [a]
```

```
replicate 0 _ = []
replicate n x = x : replicate (n-1) x
```

```
? [a] -> Int -> a
```

```
(x:_ ) !! 0 = x
(_:xs) !! n = xs !! (n-1)
```

again: **type specifies behaviour**

Questions?