# Why Types Matter

#### **IN ALMOST 5 REASONS**

Sebastiaan Visser - 2013

# Silk<sup>o</sup>

### http://silkapp.com

# Silk<sup>Q</sup>

### One product

### Started in 2009

### Grown from 4 to 10 people

Located in Amsterdam

### Why am I here?

### We love functional programming!

### Server: Haskell

### Client: JavaScript



http://haskell.org



Higher order

Lazy evaluation

Pure (no side effects)

Statically typed

Well suited for

Strict

Impure

Dynamically typed

Imperative

Hacking

**5 REASONS** 



# CORRECTNESS



# REFACTORING

### Types simplify refactoring

Particularly interesting:

Refactor into something more type safe.

### Employment

(Year, Year)

myProgram =
 do (f, t) <- getEmployment "sebas"
 printEmployment (t - f)</pre>

# \$ ./myProgram Employment duration: -4 years.

### Change representation?

#### (Year, Integer)



### Need to check every use site manually.

### Create a new type

#### data Employment = MkEm Year Year

normalize f t = MkEm (min f t) (max f t)

make :: Year -> Year -> Maybe Employment
make f t = if f < 2009
 then Nothing
 else Just (normalize f t)</pre>

module Employment
 (Employment, make, from, to)

from, to :: Employment -> Year

from (MkEm f \_) = f
to (MkEm \_ t) = t

**Opaque datatype** 

### Only export smart constructor make,

**Not** the original constructor MkEm.

Type error: Couldn't match expected type (Year, Year) with actual type Employment

# myProgram = do (f, t) <- getEmployment "sebas" printEmployment (t - f)</pre>

# myProgram = do e <- getEmployment "sebas" printEmployment (to e - from e)</pre>

# \$ ./myProgram Employment duration: 4 years.



### Fixed a bug,

### that will never occur again.

### Made only a local change, compiler points out use sites.



### What does this function do?

# foo :: [Bool] -> [Bool]

# foo :: [Bool] -> [Bool]

# The function can produce every single bit sequence.

# foo :: [a] -> [a]

### foo :: [a] -> [a]

### The function must reuse input.

reverse, empty, cycle, powerset, etc.

# foo :: ()

### Technically, not a function.

Singleton type 'unit', only one value ().

# foo :: a -> a

foo :: a -> a foo a = a

### Only one possible implementation.

id

### foo :: (a -> b) -> a -> b

## foo :: (a -> b) -> a -> b foo f a = f a

Function application, (\$) (specialization of id)

### foo :: (b -> c) -> (a -> b) -> a -> c

# foo :: (b -> c) -> (a -> b) -> a -> c foo f g a = f (g a)

## Function composition, (.)

# foo :: (a, b) -> (b, a)

# foo :: (a, b) -> (b, a) foo (a, b) = (b, a)

## Only one possible implementation.

swap

# reverse :: [a] -> [b]



## This is a lie!

## Can only produce the empty list.



## Maybe this?

foo :: a foo = foo Maybe this?

# undefined :: a

## Also called bottom, or $\bot$

# foo :: a -> b

## foo :: a -> b

## **Dangerous coercion!**

Provided by the compiler as unsafeCoerce.

## Theorems for free

**Curry-Howard isomorphism:** 

Types ⇔ Propositions

Implemention  $\Leftrightarrow$  Proofs



# Genericity

# Equality

### equalInt :: Int -> Int -> Bool



equalInt :: Int -> Int -> Bool

equalStr :: String -> String -> Bool

equalBool :: Bool -> Bool -> Bool

equalX :: X -> X -> Bool

## **Generic Equality**

## (==) :: a -> a -> Bool

Nope

#### (==) :: a -> a -> Bool

Free theorems say cannot be done!

Do all types have equality anyway?

## Constraint

### (==) :: Eq a => a -> a -> Bool

## Type classes

# class Eq a where (==) :: a -> a -> Bool

## Type classes

instance Eq Bool where
True == True = True
False == False = True
= False

## Composability

notEq :: Eq a => a -> a -> Bool
notEq a b = not (a == b)

## Composability

lsEq :: Eq a => [a] -> [a] -> Bool
lsEq [] [] = True
lsEq (x:xs) (y:ys) = x == y && lsEq xs ys
lsEq \_ \_ \_ = False

## Super classes

# instance Eq a => Eq [a] where (==) = lsEq

## ghci> [[], [True, False]] == [[True]]

False

## Type classes

Eq, Ord, Show, Read, Random, Bounded, Enum, IsString, Functor, Num, Floating, Fractional, Json, Xml, Binary, ...

Lots more

## Deriving

# data User = User { name :: String , contact :: Either Twitter Email , age :: Int } deriving (Eq, Ord, Show)

## Either, Sum or +

#### data Eihter a b = Left a | Right b

# Tuple, Product or \*

#### **data** Tuple a b = Tuple a b

# Tuple, Product or \*

$$data(,) a b = (,) a b$$

data (a, b) = (a, b)

# **Deriving Generics**

#### data User = User

- { name :: String
- , contact :: Either Twitter Email
- , age :: Int

deriving (Eq, Ord, Show, Generics)

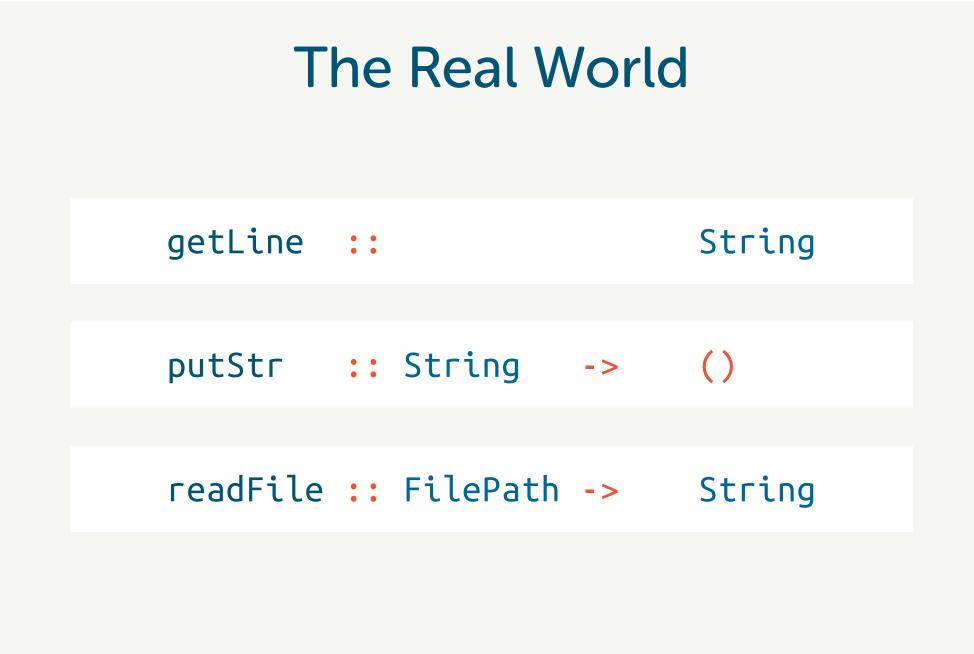
## **Deriving Generics**

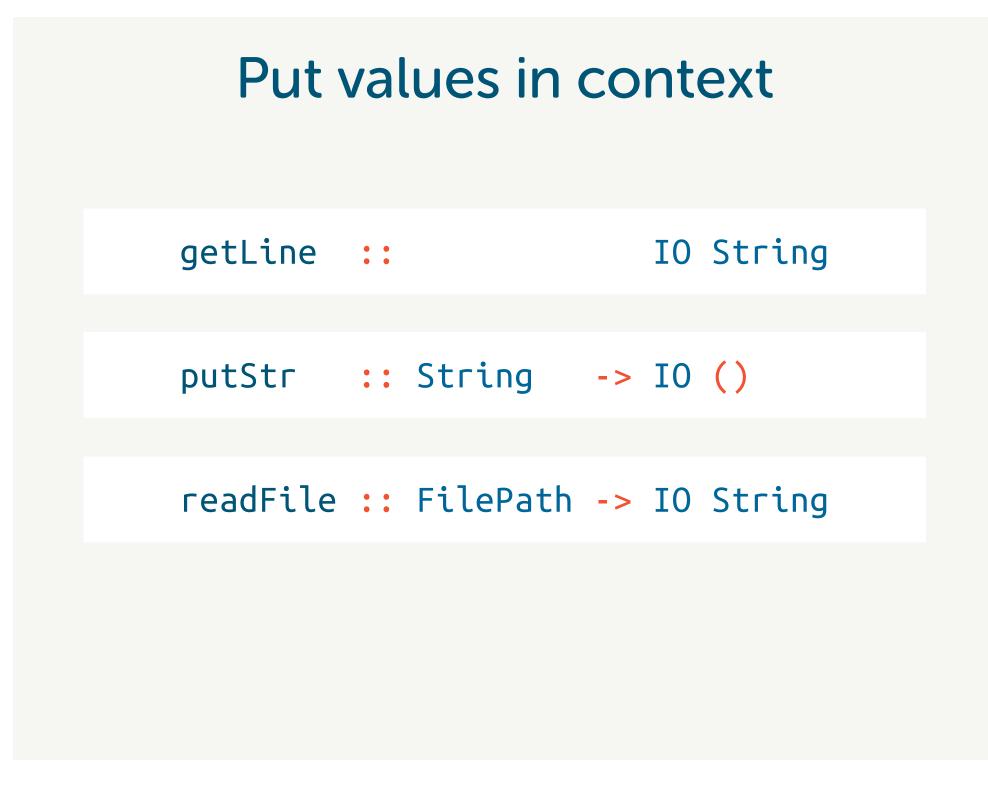
# Represents datatype algebraicly, using sums and products.

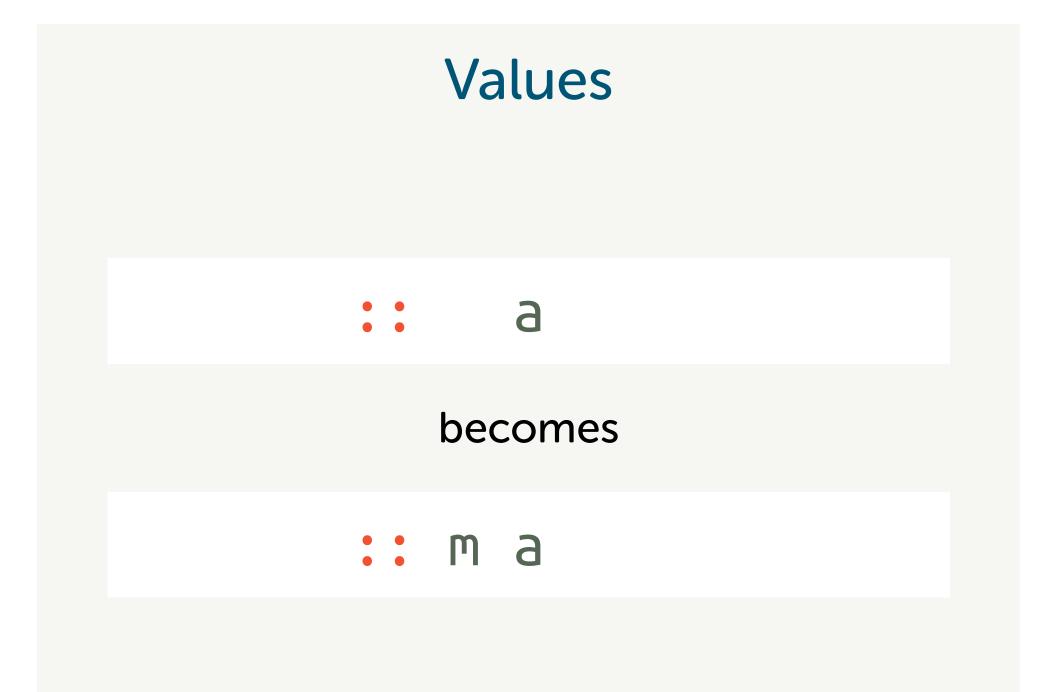
## Now we can derive Json, Xml, Binary, etc.

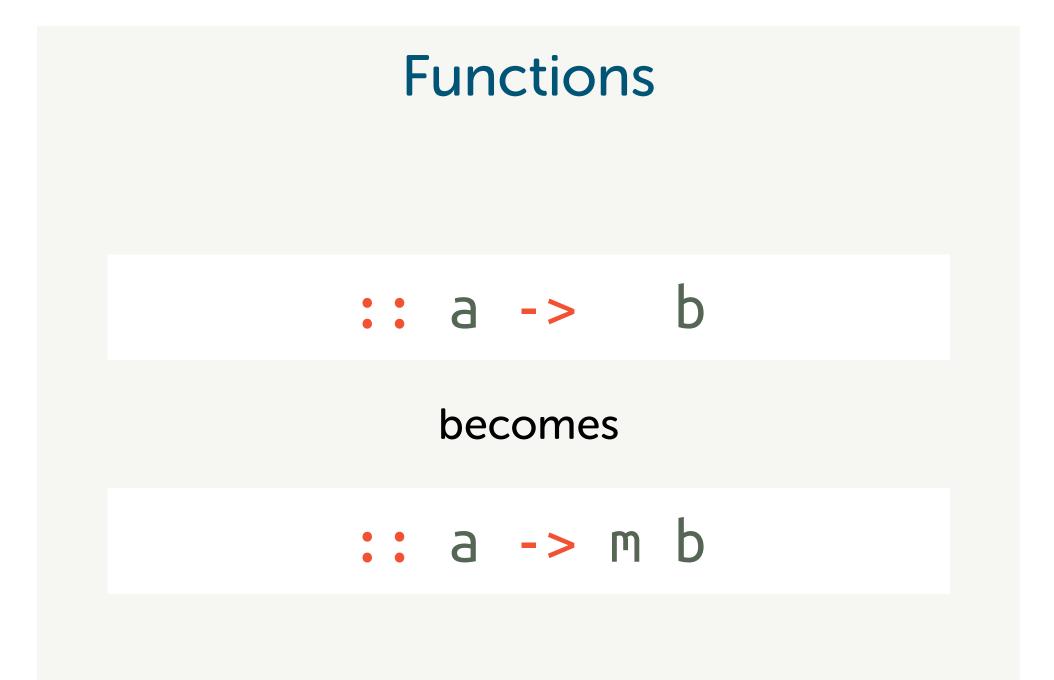
5.

# Effects









Effects

IO, ST, Cont, Identity, Maybe, Either, [], State, Reader, Writer, Random, Parser, Async, Par, STM, ...

Lots more

## Running

#### runState :: State v a -> v -> (a, v)



#### Most effects can be run

runState :: State v a -> v -> (a, v)

runReader :: Reader v a -> v -> (a )

runWriter :: Writer v a -> (a, v)

### What about IO?

#### runIO :: IO a -> a

### unsafePerformIO

#### runIO :: IO a -> a

Effects escape into purity!

## main :: IO ()

#### The RTS interprets your top level **10**.

## Composability

Parse to a list of values or failures from a socket in parallel?

ParT . ParserT . ListT . EitherT Err . IO

### Idioms

Category, Arrow, Functor, Applicative, Alternative, Monad, MonadPlus, Foldable, Traversable, ... class Functor f where
 fmap :: (a -> b) -> f a -> f b

# lengths :: Tree String -> Tree Int lengths = fmap length

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

mkUser :: String -> Int -> User

pStr :: Parser String
pInt :: Parser Int

pUser :: Parser User
pUser = pure mkUser <\*> pStr <\*> pInt

class Foldable f where
fold :: (a -> a -> a) -> f a -> a

# minTree :: Tree Int -> Int minTree = fold min

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

getLine :: IO String
print :: String -> IO ()

echo :: IO ()
echo = getLine >>= print

# echo :: IO () echo = do ln <- getLine print ln</pre>

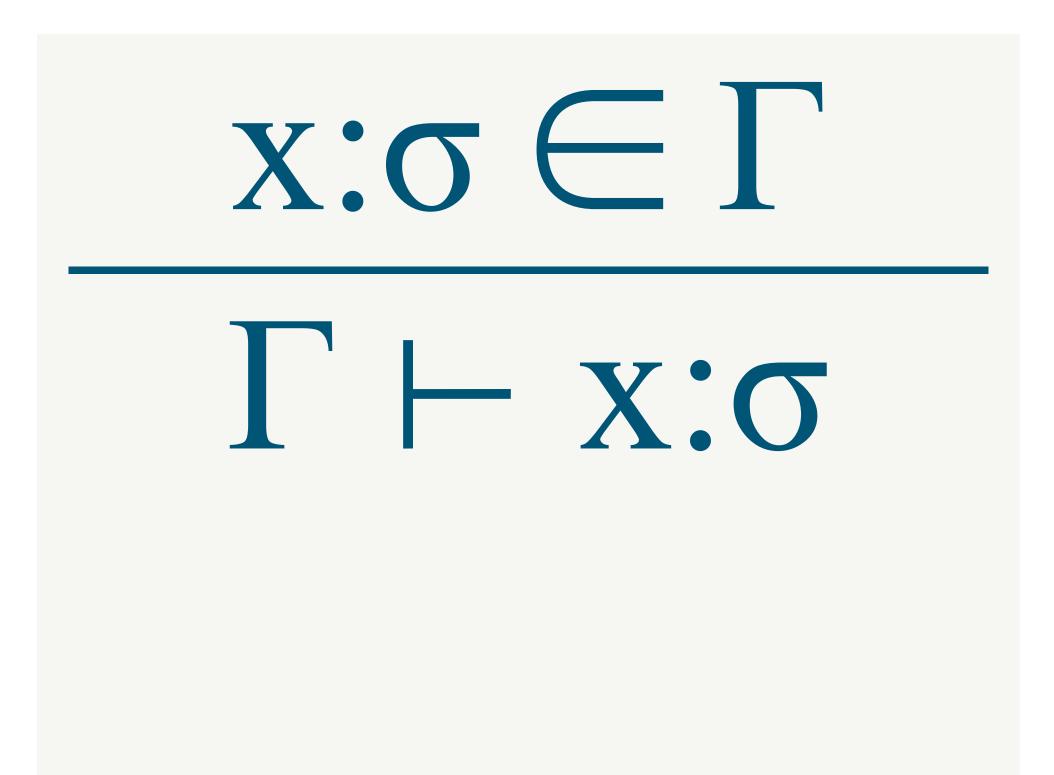
# class Functor f => Traversable f where mapM :: Monad m => (a -> m b) -> f a -> m (f b)

# fetch :: [Request] -> ParT IO [Response] fetch = mapM Http.request

fetch :: [Request] -> ParT IO [Response]
fetch = mapM Http.request

type Matrix a = Vector (Vector a)

transpose :: Matrix a -> Matrix a
transpose = mapM id



#### silkapp.com @silkapp github.com/silkapp





#### fvisser.nl @sfvisser

github.com/sebastiaanvisser