Fachhochschule
Nordwestschweiz

Functional Programming

Algebraic Data Types

Daniel Kröni
University of Applied Sciences Northwestern Switzerland

Learning Targets

You can define your own data types.

You understand polymorphic types.

You can handle failure with Maybe and Either.

You understand the structure of recursive data types.

Content

- **Predefined Types**
- **Example: Bookstore**
  - Making your own types
- **type vs. data**
- **Record Syntax**
- **Polymorphic Types**
- **Recursive Types**

Data Types until now

A type is a set of related values

- Basic Types
  - `Bool` `{True, False}`
  - `Int` `{-2`^{63} `..` 2^{63}`-1}`
  - `Integer` `{-∞ .. +∞}`
  - `Char` `{'a' .. 'Z', '\n', '\t'}`
  - `Float` `{-∞ .. +∞, Nan} single precision`
  - `Double` `{-∞ .. +∞, Nan} double precision`
- Tuples (polymorphic)
  - `(a,b)` e.g. `(Bool,Int)` `{(True,0),(False,0),(True,1),...}`
- Lists (polymorphic)
  - `[a]` e.g. `[Int]` `{[],[0],[1], [0,1], ...}`
- Functions (polymorphic)
  - `a -> b` e.g. `[Int] -> Int` `{length, sum, product, ...}`

Algebraic Data Types

- BookInfo is a new type describing a book

```
data BookInfo = Book Int String deriving (Show)
```

Type Constructor

Components

Value Constructor

Deriving Clause

- The Type Constructor defines the name of the new type
- The Value Constructor is used to create a value of this type
- The Components define the fields' types
- The Deriving Clause automatically derives instances for the named classes (here only Show)

```
data BookInfo = Book Int String deriving (Show)
```

Algebraic Data Types

- **The Book value constructor is just a function which takes the components as arguments and creates a value of type BookInfo**

```
*> :t Book
Book :: Int -> String -> BookInfo
```

```
*> :t Book 123 "Real World Haskell"
Book 123 "Real World Haskell" :: BookInfo
```

  – The nice string representation is due to "deriving (Show)"

- **Pattern matching is used to access the components of a value constructor**

```
isbn :: BookInfo -> Int
isbn (Book isbnnr _) = isbnnr
```

  – Don't miss the parentheses!

Example: Online Bookstore

- **A type can have multiple value constructors**

```
data BillingInfo = CreditCard Float String String
                 | Invoice Float Int
                  deriving (Show)
```

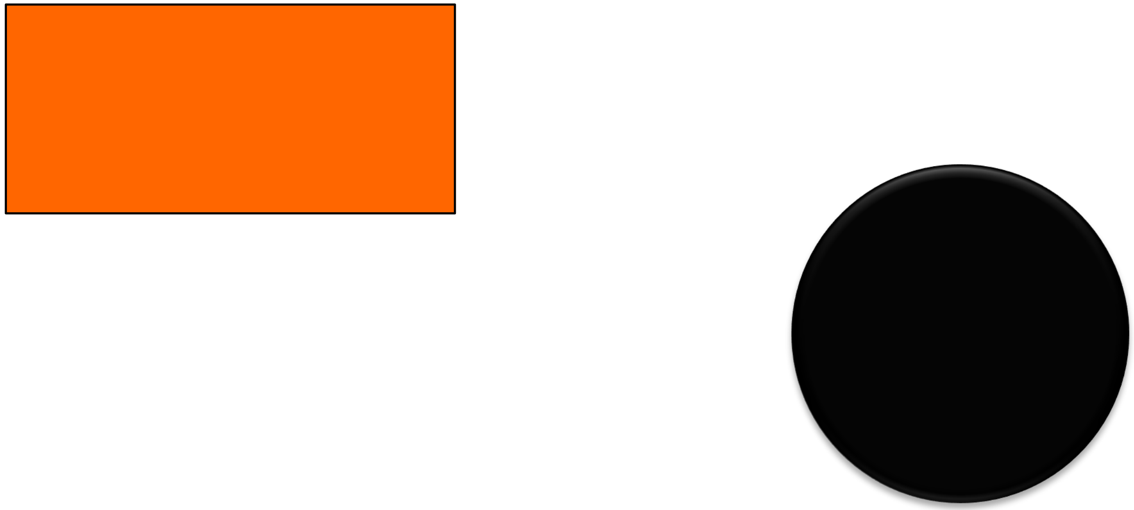
  – BillingInfo has two possible representations / value constructors

- **Pattern matching is used to distinguish the representations and introspect values**

```
amount :: BillingInfo -> Float
amount (CreditCard a _ _) = a
amount (Invoice a _)      = a
```

Fachhochschule Nordwestschweiz

Worksheet: Figures



type vs. data

- **The keyword type gives a new name to an existing type**

```
type Student = (String,Int)
```

  - Pro: Functions which are defined for the existing type can be reused
  - Cons: No protection from mixing up "incompatible" data

```
email :: Student -> String
email s = fst s
```

```
email ("MacBook", 999)
~> "MacBook"
```

- **The keyword data introduces a new type**

```
data Student = Student String Int
```

  - Pro: Mixing up incompatible types is prohibited by the compiler

```
email :: Student -> String
email (Student e _) = e
```

```
email ("MacBook", 999)
~> Type Error
```

Example: Data Type for a Person

- **A data type for a person**

```
data PersonType = Person String String String Int
```

  - requires many accessor functions

```
firstName :: PersonType -> String
firstName (Person f _ _ _) = f

lastName :: PersonType -> String
lastName (Person _ l _ _) = l

email :: PersonType -> String
email (Person _ _ e _) = e

yob :: PersonType -> Int
yob (Person _ _ _ y) = y
```

BORING

Record Syntax to the rescue

- **Record syntax allows to define a value constructor with named components**

```
data PersonType = Person {
  firstName :: String,
  lastName :: String,
  email :: String,
  yob :: Int
} deriving (Show)
```

  - Haskell automatically generates all the accessor functions for us:

```
*> firstName (Person "Haskell" "Curry" "unknown" 1900)
"Haskell"
```

  - Allows value creation in which the components are specified by name

```
Person { lastName = "Curry", firstName = "Haskell" ... }
```

Polymorphic Data Types

- **Example: Validation Framework**

```
validateEmail :: String -> ValidatedString
```

```
data ValidatedString = Ok String | Nok String
```

  – validateEmail checks its input and returns whether it is ok or malformed

```
validateAge :: Int -> ValidatedInt
```

```
data ValidatedInt = Ok Int | Nok String
```

  – validateAge checks its input and returns whether it is ok or malformed

BORING

Polymorphic Data Types

- **Solution: abstract away from the concrete types by parameterizing the type constructors with types**

```
data Validated a = Ok a | Nok String
```

Type Variable

  - a type variable stands for an arbitrary type

```
*> :t Ok "Good"
Ok "Good" :: Validated [Char]
*> :t Ok
Ok :: a -> Validated a
```

  - validateEmail has now the following type

```
validateEmail :: String -> Validated String
```

Worksheet: Maybe

Polymorphic Data Types

- **A function can produce multiple values of different types**

```
data PairType a b = PairData a b
```

  – This is like the tuple type (a,b) with a different name

- **A function can produce values of different types**

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

  – Either a value of type a wrapped in Left
    Or a value of type b wrapped in Right
  – When used for validation, Right contains the valid value and Left contains an error description per convention.

```
safeHead :: [a] -> Either String a
safeHead []     = Left "Head on empty list"
safeHead (x:xs) = Right x
```

Recursive Data Types

- A **list with elements of type e** is either empty or it is constructed of a head element of type e and and a tail which is again a **list with elements of type e**.

```
data List e = Empty | Cons e (List e)
```

  – Constructing a list

```
*Main> :t Cons 'a' (Cons 'b' (Cons 'c' Empty))
Cons 'a' (Cons 'b' (Cons 'c' Empty)) :: List Char
```

  – Processing lists

```
listSize :: List a -> Int
listSize Empty        = 0
listSize (Cons _ t) = 1 + listSize t
```

Recursive Natural Numbers

- **Natural Numbers can be defined as**

```
data Nat = Z | S Nat deriving Show
```

  - Z stands for Zero
  - S takes a Nat and returns its successor

- **Addition can be implemented as follows**

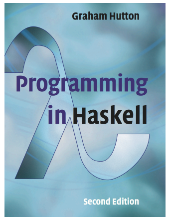
```
add :: Nat -> Nat -> Nat
add Z n      = n
add (S m) n = S (add m n)
```

- **Multiplication in terms of addition**

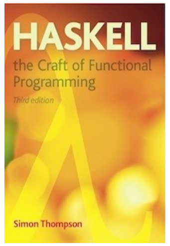
```
mul:: Nat -> Nat -> Nat
mul Z n      = Z
mul (S m) n = add n (mul m n)
```

Fachhochschule
Nordwestschweiz

Further Reading



Chapter 8



Chapter 14

Chapter 7

http://learnyouahaskell.com/types-and-typeclasses