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 functions 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:

```haskell
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[1]: an object that can be used to generate random values of type `Int`.

[1]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

```haskell
-- 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

```haskell
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

```haskell
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:

```haskell
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

```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

```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.[2]

[2] 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

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

Three values of `Nat`:

```
Zero
```

```
Suc
 ↓
Zero
```

```
Suc
 ↓
Suc
 ↓
Zero
```

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

```haskell
-- 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

```haskell
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:[3]

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

[3]Not counting any terms with `undefined`.

Counting the elements of a type

How many elements are in the following types:[3]

- `Either Bool Answer` $2 + 3 = \mathbf{5}$
- `(Bool, Bool, Answer)`
- `Maybe (Bool, Bool)`

[3]Not counting any terms with `undefined`.

Counting the elements of a type

How many elements are in the following types:[3]

- `Either Bool Answer` $2 + 3 = \mathbf{5}$
- `(Bool, Bool, Answer)` $2 \times 2 \times 3 = \mathbf{12}$
- `Maybe (Bool, Bool)`

[3] Not counting any terms with `undefined`.

Counting the elements of a type

How many elements are in the following types:[3]

- `Either Bool Answer` $2 + 3 = \mathbf{5}$
- `(Bool, Bool, Answer)` $2 \times 2 \times 3 = \mathbf{12}$
- `Maybe (Bool, Bool)` $1 + (2 \times 2) = \mathbf{5}$

[3] 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?

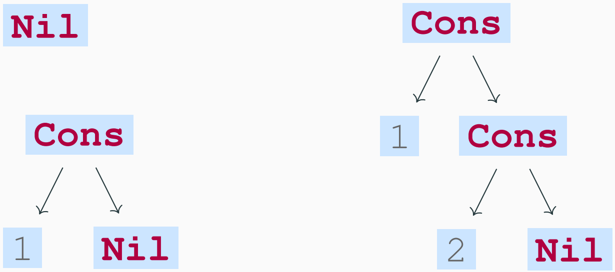
Answer.

```
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`:



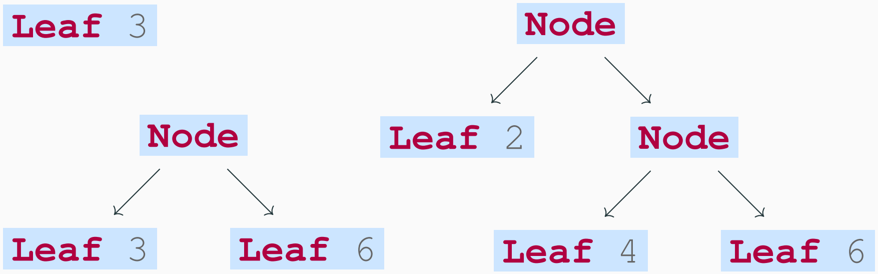
Example: Binary trees

```
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`

```haskell
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