Functional Programming

More about lists

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2017-2018

Lists recap

Zero or more values

```
[]    [1]    [True, False]    ["a", "bunch", "of", "flowers"]
```

All have the same type

```
[True, False] -- good
[1, "two", False] -- bad, type error
```

Order matters

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

List syntax

```
(1 : (2 : (3 : [])))
==
1 : 2 : 3 : []
==
[1,2,3]
```

Strings are lists of characters

```
"Hearts" == ['H','e','a','r','t','s']
```

Defining a list datatype

The values of type [a] are ...

- either `[]`, the empty list
- or `x:xs` where `x` has type `a` and `xs` has type `[a]`
  ":" is pronounced "cons"

```
data List a = ...
```

Defining a list datatype

The values of type [a] are ...

- either `[]`, the empty list
- or `x:xs` where `x` has type `a` and `xs` has type `[a]`
  ":" is pronounced "cons"

```
data List a = ...
```

Corresponding definition

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

- New: `List` is a parametric datatype with type parameter `a`
- Many functions on lists are also parametric (i.e., **polymorphic**)

Polymorphic functions on lists

```
length :: [a] -> Int
(++)   :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
take   :: Int -> [a] -> [a]
zip    :: [a] -> [b] -> [(a,b)]

map    :: (a -> b)    -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

Prelude functions on lists

Functions on specific lists

```
and, or           :: [Bool] -> Bool
words, lines      :: String -> [String]
unwords, unlines  :: [String] -> String
```

Prelude functions on lists

Functions on specific lists

```
and, or           :: [Bool] -> Bool
words, lines      :: String -> [String]
unwords, unlines  :: [String] -> String
```

Overloaded functions on lists

```
sum, product      :: Num a => [a] -> a
elem              :: Eq a => a -> [a] -> Bool
sort              :: Ord a => [a] -> [a]
```

Some examples . . .

- append, reverse
- sum, product
- take, drop, splitAt
- zip, unzip
- insert, isort, qsort
- QuickCheck: collect, classify

Quicksort!

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort bigger
  where
    smaller = filter (<= x) xs
    bigger  = filter (> x) xs
```

An unfortunate QuickCheck — ghci interaction

Two properties

```
prop_take_drop n xs = take n xs ++ drop n xs == xs
nonprop_take_drop n xs = drop n xs ++ take n xs == xs
```

An unfortunate QuickCheck — ghci interaction

Two properties

```
prop_take_drop n xs = take n xs ++ drop n xs == xs
nonprop_take_drop n xs = drop n xs ++ take n xs == xs
```

Testing ...

```
*Main> quickCheck prop_take_drop
+++ OK, passed 100 tests.
*Main> quickCheck nonprop_take_drop
+++ OK, passed 100 tests.
```

An unfortunate QuickCheck — ghci interaction

Two properties

```
prop_take_drop n xs = take n xs ++ drop n xs == xs
nonprop_take_drop n xs = drop n xs ++ take n xs == xs
```

Testing ...

```
*Main> quickCheck prop_take_drop
+++ OK, passed 100 tests.
*Main> quickCheck nonprop_take_drop
+++ OK, passed 100 tests.
```

Oops! what went wrong?

```
prop_take_drop :: Eq a => Int -> [a] -> Bool
nonprop_take_drop :: Eq a => Int -> [a] -> Bool
```

- The properties have polymorphic types, but...
- QuickCheck does not work with polymorphic types!

Ghci "helps"

- Instead of indicating the problem, ghci chooses a more specific **default type**
- In this case, it plugs the unit type for a
- QuickCheck tests

```
prop_take_drop :: Eq a => Int -> [()] -> Bool
nonprop_take_drop :: Eq a => Int -> [()] -> Bool
```

- Order does not matter when all elements are the same...

Force ghci to be unhelpful

- Use type signatures
- Disable defaulting

```
*Main> :set -XNoExtendedDefaultRules
```

- Restrict types used in defaulting

```
*Main> default (Integer, Double)
```

Break Time — Questions?