Defining and testing functions

Lecture 2 of CSE 3100

Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

The many restrictions of Haskell

- No mutable variables
- No while loops / for loops / ...
- No try/catch blocks
- No side effects
- No objects
- Not even goto!

The many restrictions of Haskell

- No mutable variables
- No while loops / for loops / ...
- No try/catch blocks
- No side effects
- No objects
- Not even goto!

How do we even write programs in such a language?

Lecture plan

- Tools for writing functional programs
  - Pattern matching
  - Recursion
- Writing reusable functions:
  polymorphic types and type classes
- Testing functions:
  QuickCheck

Pattern matching

Pattern matching

We can define functions by case analysis using pattern matching:

```
not :: Bool -> Bool
not True  = False
not False = True
```

Pattern matching is one of the most powerful and useful features of Haskell.
You will have to use it a lot!

Pattern variables and wildcards

A pattern variable matches any value that didn't match the previous patterns:

```
rank 1 = "first"
rank 2 = "second"
rank 3 = "third"
rank n = show n ++ "th"
```

A wildcard `_` is like a pattern value for which you don't care about the value:

```
isItTheAnswer 42 = True
isItTheAnswer _  = False
```

Matching on multiple arguments

```
xor :: Bool -> Bool -> Bool
xor True  False = True
xor False True  = True
xor _     _     = False
```

Side note: Order of clauses

```haskell
f True _    = 1
f _    True = 2
f _    _    = 3
```

vs.

```haskell
g _    True = 2
g True _    = 1
g _    _    = 3
```

Haskell will use the first clause that matches,
so `f True True = 1` but
`g True True = 2`!

Side note: operator syntax

Infix operations such as `(+)`, `(-)`, `(==)`, `(!!)`, ...are just regular Haskell functions:

- Name must consist of special characters only
- Must be parentesized when appearing by themselves

They can also be used as normal functions:

```
(+) 1 1 == 1 + 1
```

Three definitions of `(&&)`

Here are three definitions of the library function `(&&) :: Bool -> Bool -> Bool`

Three definitions of `(&&)`

Here are three definitions of the library function `(&&) :: Bool -> Bool -> Bool`

```
-- version 1
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

```
-- version 2
True && True = True
_    && _    = False
```

```
-- version 3
True  && b = b
False && _ = False
```

Question: Is there any difference in practice?

Pattern matching on lists

```haskell
isEmpty :: [a] -> Bool
isEmpty []      = True
isEmpty (x:xs) = False
```

- Any list is either `[]` or `x:xs`
- `[1,2,3]` is syntactic sugar for `1:2:3:[]`

Incomplete matches

```haskell
takeTwo (x1:x2:xs) = (x1,x2)
```

```
> takeTwo [6]
*** Exception: Non-exhaustive
patterns in function takeTwo
```

Tip. Add `-Wincomplete-patterns` to the `ghc-options` of your `.cabal` project to get a warning for incomplete patterns!

Using guards

We can use guards to add boolean conditions to a clause:

```
signum x
  | x < 0       = -1
  | x == 0      = 0
  | otherwise = 1
```

- Guards `| b` appear after the patterns (and before `=`)
- The condition `b` should be of type `Bool`
- `otherwise` is defined to be always `True`

Mixing guards and pattern matching

```haskell
capitalize :: String -> String
capitalize (c : cs)
  | isLower c = (toUpper c) : cs
capitalize cs = cs
```

Recursion

Recursion example: factorial

```haskell
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Evaluating `fac` step by step:

```
fac 3 = 3 * fac 2
      = 3 * (2 * fac 1)
      = 3 * (2 * (1 * fac 0))
      = 3 * (2 * (1 * 1))       = 6
```

Question. What happens if $n < 0$?

Recursion on lists

Recursion is not limited to numbers: we can also recurse over structured data such as lists:

```
product :: [Int] -> Int
product []       = 1
product (x:xs)   = x * product xs

zip :: [Int] -> [Int] -> [(Int,Int)]
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
zip _      _      = []
```

Why use recursion?

- Often the most natural way to write functional programs
- Recursion + list comprehensions completely remove the need for traditional loops
- We can prove properties of recursive functions by induction

Implementing recursive functions

A 4-step plan for implementing a function:

1. Write down the type
2. Enumerate the cases
3. Define the base case(s)
4. Define the recursive case(s)

Example: Implementing insertion sort

Step 1: write down the type

```
isort :: [Int] -> [Int]
isort xs = _
```

Example: Implementing insertion sort

Step 2: enumerate the cases

```
isort :: [Int] -> [Int]
isort []     = _
isort (x:xs) = _
```

Example: Implementing insertion sort

Step 3: define base cases

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = _
```

Example: Implementing insertion sort

Step 4: define recursive cases

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = insert x (isort xs)
```

Example: Implementing insertion sort

Step 1: write down the type

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = insert x (isort xs)
  where
    insert :: Int -> [Int] -> [Int]
    insert x ys = _
```

Example: Implementing insertion sort

Step 2: enumerate the cases

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = insert x (isort xs)
  where
    insert :: Int -> [Int] -> [Int]
    insert x []     = _
    insert x (y:ys)
      | x <= y      = _
      | otherwise   = _
```

Example: Implementing insertion sort

Step 3: define base cases

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = insert x (isort xs)
  where
    insert :: Int -> [Int] -> [Int]
    insert x []     = [x]
    insert x (y:ys)
      | x <= y      = _
      | otherwise   = _
```

Example: Implementing insertion sort

Step 4: define recursive cases

```haskell
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = insert x (isort xs)
  where
    insert :: Int -> [Int] -> [Int]
    insert x []       = [x]
    insert x (y:ys)
      | x <= y        = x:y:ys
      | otherwise     = y:(insert x ys)
```

Polymorphic types

Recap: Haskell Types

Typing annotations: `x :: a`

Basic types `Bool`, `Int`, `Integer`, `Float`, `Double`, `Char`, `String`, ...

List type `[a]` [1]

Tuple types `(a,b)`, `(a,b,c)`, ...

Function types `a -> b`, `a -> b -> c`, ...

[1] `String = [Char]`

Question

What other types could be given to these functions?

- `length :: [Int] -> Int`
- `concat :: [[Int]] -> [Int]`
- `isSorted :: [Int] -> Bool`

Polymorphic functions

Many functions can be given several types, for example `length :: [Int] -> Int`, `length :: [Bool] -> Int`, ...

These functions are given a polymorphic type:

```
length :: [a] -> Int
```

Unlike generics in Java/C#/..., we do not need to give the type since Haskell can infer it for us:

```
> length [2,3,5,7]
4
```

Some polymorphic functions on tuples

```
fst  :: (a,b) -> a
snd  :: (a,b) -> b
swap :: (a,b) -> (b,a)
```

Question. Can you guess what they do from their types? Is there anything *else* that they could possibly do?

Some polymorphic functions on lists

```haskell
(:)    :: a -> [a] -> [a]
head   :: [a] -> a          -- partial!
tail   :: [a] -> [a]        -- partial!
(++)   :: [a] -> [a] -> [a]
(!!)   :: [a] -> Int -> a -- partial!
take   :: Int -> [a] -> [a]
drop   :: Int -> [a] -> [a]
zip    :: [a] -> [b] -> [(a,b)]
unzip  :: [(a,b)] -> ([a],[b])
```

Quiz question

Question. Which of the following equations is true for all Haskell lists `xs :: [a]`?

1. `[]:xs == [[],xs]`
2. `xs:xs == [xs,xs]`
3. `[[]] ++ xs == xs`
4. `[[]] ++ [xs] == [[],xs]`

Type classes

Polymorphic functions with constraints

Question: What should be the type of `double x = x + x`?

- `double :: Int -> Int` is too restrictive (it also works for `Float`!)
- `double :: a -> a` is too general (it doesn't work for `Bool`!)

Polymorphic functions with constraints

Question: What should be the type of `double x = x + x`?

- `double :: Int -> Int` is too restrictive (it also works for `Float`!)
- `double :: a -> a` is too general (it doesn't work for `Bool`!)

Solution: Add a constraint `Num a =>`:

```
double :: Num a => a -> a
```

Type classes in Haskell

`Num` is an example of a type class: a collection of types that support a common interface.

```
class Num a where
  (+)    :: a -> a -> a
  (-)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a
  abs    :: a -> a
  fromInteger :: Integer -> a
```

Question. Is this the same as interfaces in Java?

The Eq class

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

The `Ord` class

```
class Eq a => Ord a where
  (<)  : a -> a -> Bool
  (<=) : a -> a -> Bool
  (>)  : a -> a -> Bool
  (>=) : a -> a -> Bool
  max  : a -> a -> a
  min  : a -> a -> a
```

`Ord` is an example of a subclass: any instance of `Ord` must also be an instance of `Eq`.

Other useful type classes

`Show` is for printing values:

`show 123 == "123"`

`Read` is for parsing values:

`read "123" == 123`

`Integral` is for integral division:

``9 `div` 2 == 4``, ``9 `mod` 2 == 1``

`Fractional` is for floating-point division:

`9.0 / 2.0 == 4.5`

Discussion

Question. Can we define a Haskell function
`isSorted :: Ord a => [a] -> Bool`
that checks if a given list is sorted *without* using pattern matching or list comprehensions, only using functions from the Prelude?

```
isSorted [1,3,6,10] == True
isSorted [5,6,1,3] == False
isSorted [] == True
```

Property-based testing with QuickCheck

Unit testing

Writing unit tests is important but also boring and difficult:

- *Boring* because you need a lot of unit tests
- *Difficult* because it is very easy to miss cases

What if we could generate test cases automatically?

Unit testing

Writing unit tests is important but also boring and difficult:

- *Boring* because you need a lot of unit tests
- *Difficult* because it is very easy to miss cases

What if we could generate test cases automatically?

Enter property-based testing.

Property-based testing

Instead of writing individual test cases, we can write down properties of our programs and generate test cases from those.

```
prop_abs_symmetric :: Int -> Int -> Bool
prop_abs_symmetric x y =
  abs (x - y) == abs (y - x)

prop_reverse_idempotent :: [Int] -> Bool
prop_reverse_idempotent xs =
  reverse (reverse xs) == xs
```

Random testing of properties

To test a property, we can simply generate many inputs randomly and check if the property holds for all of them.

Randomly testing the property `prop_abs_symmetric`:

```
abs (0.0 - 2.7)         == abs (2.7 - 0.0)
abs ((-0.7) - (-0.9))   == abs ((-0.9) - (-0.7))
abs (6.5 - (-4.0))      == abs ((-4.0) - 6.5)
abs (2.0 - 7.7)         == abs (7.7 - 2.0)
abs ((-19.0) - 2.8)     == abs (2.8 - (-19.0))
  ...
```

Advantages of property-based testing

- You spend less time writing tests: a single property replaces many tests
- You get better coverage: test lots of combinations you'd never try by hand
- You spend less time on diagnosing errors: failing tests can be minimized

The QuickCheck library for Haskell

QuickCheck is a Haskell library for writing property-based tests.

It was introduced in 1999 by Koen Claessen and John Hughes.[2]

It has been ported to many other languages: C, C++, Java, JavaScript, Python, Scala, …[3]

---

[2] K. Claessen and J. Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*

[3] `https://en.wikipedia.org/wiki/QuickCheck`

Installing QuickCheck

Install QuickCheck with Cabal:

```
> cabal install QuickCheck
```

Or in a Stack project, add the following to the list of dependencies in `package.yaml`:

```
- QuickCheck >= 2.14
```

Basic usage of QuickCheck

```haskell
import Test.QuickCheck

dist :: Int -> Int
dist x y = abs (x - y)

prop_dist_self :: Int -> Bool
prop_dist_self x = dist x x == 0

prop_dist_sym :: Int -> Int -> Bool
prop_dist_sym x y = dist x y == dist y x

prop_dist_pos :: Int -> Int -> Bool
prop_dist_pos x y = dist x y > 0
```

Running QuickCheck from GHCi

```
> quickCheck prop_dist_self
+++ OK, passed 100 tests.
> quickCheck prop_dist_sym
+++ OK, passed 100 tests.
> quickCheck prop_dist_pos
*** Failed! Falsified (after 1 test):
0
0
```

Running QuickCheck tests in batch

```
main = do
  quickCheck prop_dist_self
  quickCheck prop_dist_sym
  quickCheck prop_dist_pos
```

```
> runghc Distance.hs
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
*** Failed! Falsified (after 1 test):
0
0
```

Anatomy of a QuickCheck test

```
prop_isort_isSorted :: [Int] -> Bool
prop_isort_isSorted xs =
  isSorted (isort xs)
```

- Name starts with `prop_` (convention, but required on Weblab)
- Type of argument must be one for which we can generate arbitrary values
- Return type must be `Bool` or `Property`

Finding minimal counterexamples

If QuickCheck finds a counterexample, it will apply shrinking to find a counterexample that is as small as possible.

```
prop_all_sorted :: [Int] -> Bool
prop_all_sorted xs = isSorted xs
```

Running `quickCheck prop_all_sorted` will always return either `[1,0]` or `[0,-1]`.

Shrinking inputs

QuickCheck determines how to shrink a counterexample based on its type:

- `Int`: try number closer to 0
- `Bool`: try `False` instead of `True`
- `(a,b)`: shrink one of the components
- `[a]`: either shrink one value in the list, or delete a random element

What's next?

Next lecture: Algebraic data types

To do:

- Read the book:
  - Today: 3.7-3.9, 4.1-4.4, 6.1-6.6, QuickCheck notes
  - Next lecture: 8.1-8.4, 8.6
- Finish week 1 exercises on WebLab
- Ask & answer questions on TU Delft Answers