Questions & Answers 27 Sept.

Functional Programming 2018/19

Alejandro Serrano

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Administrativia

Format of the exam

Similar to the previous exams in the web

- The exam is on **paper**
  - Some places (wrongly) state that it is digital
- Two types of questions
  - Open (85%): explain something or write some code
  - Multiple choice (15%): choose one answer
- There is a maximum amount of space per question
  - Only the things you write there count
  - Don't worry! There's way more space that you need!
- You can**not** bring any notes to the exam
- You have to answer in English
  - You can bring a (mother tongue) - English dictionary

Contents of the exam

Everything until today

- Basic types: lists, tuples, numbers
- User-defined data types
- Define functions by pattern matching
- Recursion on lists and other data types
- Define and use higher-order functions
- Write classes and instances
- Infer and check the type of an expression
- **No**: write functions using accumulators
- **No**: use functions to represent data

Contents of the exam

Do I need to know all the types by heart?

Writing and understanding type signatures is something you need to learn, and we need to test

- You have to know or deduce the type of simple functions such as `(++)`, `max`, `(==)`, and so forth
- Some higher-order functions are *very* important
  - `map :: (a -> b) -> [a] -> [b]`
  - `filter :: (a -> Bool) -> [a] -> [a]`
  - `foldr :: (a -> b -> b) -> b -> [a] -> b`

  In most cases you can deduce their type if you know what they do

Outcome of the exam

- You should expect your grades in about two weeks
- What happens if I fail the exam?
  - *Nothing*, your grade is the average with the final one
  - Remember, T = 0.3 × midterm + 0.7 × final
  - *Reflect* on your mistakes and *act* to fix them

Q&A session

The most repeated question

More examples of type inference

Let me answer some smaller questions before

Difference between `(:)` and `(++)`

Difference between `(:)` and `(++)`

Let us look at the types:

```
(:)  :: a   -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
```

- `(:)` puts an *element* in front of a *list*
- `(++)` puts a *list* in front of a *list*

Remember the following equivalence:

```
[x] ++ ys = x : ys
```

But the following is not even well-typed:

```
xs ++ [y] = xs : y   -- ERG ONWAAR!!
```

Difference between `foldr` and `foldl`

Difference between `foldr` and `foldl`

Not much, just “right” or “left” in their names

- `foldr` and `foldl` are about parenthesis and nesting
  - `foldr (+) 0 [1,2,3] = 1 + (2 + (3 + 0))`
  - `foldl (+) 0 [1,2,3] = ((0 + 1) + 2) + 3`

Difference between `foldr` and `foldl`

Not much, just “right” or “left” in their names

- `foldr` and `foldl` are about parenthesis and nesting
  - `foldr (+) 0 [1,2,3] = 1 + (2 + (3 + 0))`
  - `foldl (+) 0 [1,2,3] = ((0 + 1) + 2) + 3`

Several people asked me about `applyAll` as a fold

- Unfortunately, we do not have the time :(
- I am available tomorrow from 8.30 to 16.30, at BBG-5.70
- I am **not** available on Monday

What can we do with anonymous function / (lambda) abstractions? Is there something deep?

What can we do with anonymous function / (lambda) abstractions? Is there something deep?

They are just a handier way to write (small) functions

```
g = map f                    ==>   g = map (\x -> foo)
  where f x = foo
```

They have a limitation: you cannot do case distinction

```
g = map f                    ==>   g = map (\x -> ??)
  where f True  = foo
        f False = bar
```

We do **not** enforce a particular style about this

When do we use _ instead of a name?

1. We use it in *patterns*
   - At the left of = or ->
2. When we do not care about a value
   - When it is not used to the right of = or ->

```
map f [] = []   -- not using f
-- Better
map _ [] = []
```

What is the difference between `name@(...)`, `\z -> ...` and `let x = ... in ...`?

What is the difference between `name@(...)`, `\z -> ...` and `let x = ... in ...`?

In common: they introduce a new *name* into scope

- You can use that name in the `...` part

The difference lies in what they refer to

- `let x = ... in ...` gives a name to an expression which is part of a bigger expression

```
unit (x, y) = let norm = sqrt(x*x + y*y)
              in (x/norm, y/norm)
```

- The others refer to the *argument* of a function
  - `name@(...)` always appear at the left of the = symbol

What is the difference between `name@(...)`, `\z -> ...` and `let x = ... in ...`?

With pattern matching we choose a branch in a function and access the components of a value

- We can match also in an anonymous function!
  - But we can only match *one* pattern

```
norm (x,y) = ...          norm = \(x,y) -> ...

length []     = ...       length = \?? -> ...
length (x:xs) = ...
```

With `name@(...)` we give a name to the whole value and then we pattern match in its components

```
f lst@(x:xs) = ...  -- 'lst' is the whole list
   -- 'x' is the head of 'lst' and 'xs' is its tail
```

What is the difference between `(.)` and parentheses?

What is the difference between `(.)` and parentheses?

- We use parentheses to delineat *values*
  - Think about arguments and return values
- When we use `(.)` we think at the level of *functions*
  - We never "touch" the values involved

```
f xs = map (\x -> x * 2) (filter even xs)
--      ----------------  -----------
--          function #1    function #2

f    = map (*2) . filter even
```

We drop the `xs` because we are talking about functions

- Function `f` is `map ...` after `filter even`

When can I “drop” arguments?

Rule 1: you can drop the last argument if it is also the last argument in the right-hand side

```
parseTable xs = map words xs
-- becomes
parseTable    = map words
```

When can I “drop” arguments?

Rule 1: you can drop the last argument if it is also the last argument in the right-hand side

```
parseTable xs = map words xs
-- becomes
parseTable    = map words
```

Rule 2: you can drop the last argument if you can express the right-hand side as a composition of functions

```
f xs = map (\x -> x * 2) (filter even xs)
-- becomes
f    = map (*2) . filter even
```

When can I “drop” arguments?

In general, compoisition only works with *one* argument

- Rule 2 applies at most once
- You can use `uncurry` and put arguments in a tuple

When do I have variables in a `data`?

In other words, what is the difference between?

```
data Point1   = Pt1 Float Float
data Point2 a = Pt2 a a
```

When do I have variables in a `data`?

In other words, what is the difference between?

```
data Point1   = Pt1 Float Float
data Point2 a = Pt2 a a
```

- The second defines a *polymorphic* type
  - You can write `Point2 Float`, `Point2 Int`…
- Useful for “container” types
  - Lists, trees, `Maybes`

Enough waiting! We want to infer some types!

What is the type of `map . foldr`?

General rule: if `f :: A -> B` and `e :: A`, then `f e :: B`

Infix operator syntax comes to play here

- `map . foldr = (.) map foldr`
- The function `(.)` takes two arguments, `map` and `foldr`

```
(.)    :: (b -> c) -> (a -> b) -> a -> c
map    :: (a -> b) -> [a] -> [b]
foldr :: (a -> b -> b) -> b -> [a] -> b
```

1. Introduce new names to disambiguate
   - I tend to use `?n` to make it clear
   - Other people use Greek letters
   - I don't care, but make it clear in the exam

```
-- a, b and c in each type are unrelated
(.)    :: (?b -> ?c) -> (?a -> ?b) -> ?a -> ?c
map    :: (?d -> ?e) -> [?d] -> [?e]
foldr :: (?u -> ?v -> ?v) -> ?v -> [?u] -> ?v
```

1. Introduce new names to disambiguate
   - I tend to use `?n` to make it clear
   - Other people use Greek letters
   - I don't care, but make it clear in the exam

```
-- a, b and c in each type are unrelated
(.)    :: (?b -> ?c) -> (?a -> ?b) -> ?a -> ?c
map    :: (?d -> ?e) -> [?d] -> [?e]
foldr :: (?u -> ?v -> ?v) -> ?v -> [?u] -> ?v
```

2. Write equations to fit the type in the function with the types of the arguments

```
?b -> ?c = (?d -> ?e) -> [?d] -> [?e]
?a -> ?b = (?u -> ?v -> ?v) -> ?v -> [?u] -> ?v
```

3. Solve the equations
   - Remember about the implicit parenthesis for ->

```
?b -> ?c = (?d -> ?e) ->  [?d] -> [?e]
         = (?d -> ?e) -> ([?d] -> [?e])
===> ?b = ?d -> ?e
     ?c = [?d] -> [?e]

?a -> ?b = (?u -> ?v -> ?v) -> (?v -> [?u] -> ?v)
===> ?a = ?u -> ?v -> ?v
     ?b = ?v -> [?u] -> ?v

-- We have to equations for ?b
?b = ?d -> ?e = ?v -> ([?u] -> ?v)
===> ?d = ?v
     ?e = [?u] -> ?v
```

4. Obtain the result type
   - The remainder of the fn. without the given arguments

```
(.) :: (?b -> ?c) -> (?a -> ?b) -> ?a -> ?c
==> map . foldr :: ?a -> ?c
```

   - Substitute unknowns for their values

```
?a -> ?c
= -- Parentheses!
(?u -> ?v -> ?v) -> [?d] -> [?e]
= -- We can keep substituting
(?u -> ?v -> ?v) -> [?v] -> [[?u] -> ?v]
```

   - This type works for any `?u` and `?v`

```
map . foldr :: (u -> v -> v) -> [v] -> [[u] -> v]
```

How do I check that I am right?

Use the interpreter to ask for the type

```
> :t map . foldr
map . foldr
  :: Foldable t => (a1 -> a2 -> a2)
                -> [a2] -> [t a1 -> a2]
> :set -XTypeApplications
> :t map . foldr @[]
map . foldr @[] :: (a1 -> a2 -> a2)
                -> [a2] -> [[a1] -> a2]
```

The names `a1` and `a2` do not matter

- But the *shape* of the type must be the same

Rinse and repeat

What is the type of `map (map map)`?

The *result* of the inner `map map` is the *arg.* to the outer `map`

Rinse and repeat

What is the type of `map (map map)`?

The *result* of the inner `map map` is the *arg.* to the outer `map`

1. Introduce new names to disambiguate
   - Each `map` gets different names

```
map :: (?a -> ?b) -> [?a] -> [?b]   -- #1
map :: (?c -> ?d) -> [?c] -> [?d]   -- #2
map :: (?e -> ?f) -> [?e] -> [?f]   -- #3
```

2. Obtain the type of the inner `map map`
   - Pose and solve the equations

```
?c -> ?d = (?e -> ?f) -> [?e] -> [?f]
==> ?c = (?e -> ?f)
    ?d = [?e] -> [?f]
```

   - Obtain the result type

```
map map :: [?c] -> [?d]
        = [?e -> ?f] -> [[?e] -> [?f]]
```

3. Obtain the type of the whole expression
   - Pose and solve the equations

```
?a -> ?b = [?e -> ?f] -> [[?e] -> [?f]]
==> ?a = [?e -> ?f]
    ?b = [[?e] -> [?f]]
```

   - Obtain the result type

```
map (map map) :: [?a] -> [?b]
              = [[?e -> ?f]] -> [[[?e] -> [?f]]]
```

4. The type works for any `?e` and `?f`

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