04IN1023: Introduction to functional programming

Resit—WS 2013/14

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel, M.Sc. Andrei Varanovich
2 December 2013

Name, Vorname	
Matrikel-Nr.	
Email	.......................`@uni-koblenz.de`
Studiengang	☐ BSc Inf ☐ BSc CV ☐ ....
Prüfungsversuch	☐ 1 ☐ 2 ☐ 3

Hiermit bestätige ich, dass ich zur Klausur angemeldet und zugelassen bin.
Eine falsche Angabe wird als Täuschungsversuch gewertet.

Unterschrift: ________________

Korrekturabschnitt

Aufgabe	Punkte (0-2)	Zusatzpunkt?
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

1 “Simple algorithms”

Define a function that selects the odd indexes of a list of ints. Here is an illustration:

```
> odds []
[]
> odds [1]
[]
> odds [1,2]
[2]
> odds [1,2,3]
[2]
> odds [1,2,3,4]
[2,4]
```

Reference solution

```
-- Function signature not required
odds :: [Int] -> [Int]

odds [] = []
odds [x] = []
odds (x:y:r) = y : odds r
```

2 “Simple data models”

Declare data types for vector (line) images such. An *image* is defined as a list of lines. A *line* is defined as a list of points. A *point* is defined as a pair of floats.

Reference solution

```
type Image = [Line]
type Line = [Point]
type Point = (Float, Float)
```

3 “Local scope”

Consider the following code:

```
test x y z = smaller x y && smaller x z
smaller x y = x < y
```

Transform the code such that local scope is used for the definition of *smaller*, i.e., *smaller* becomes a local function of *test*. The local definition should only have a single argument.

Reference solution

```
test x y z = smaller y && smaller z
  where
    smaller q = x < q -- y could be used instead of q
```

4 "Parametric polymorphism"

Define a polymorphic function including its signature such that the odd elements of a given list are filtered. Consider the following illustration:

```
> odds [5,1,2,4]
[5,1]
```

Reference solution

```
odds :: Integral x => [x] -> [x]
odds [] = []
odds (x:xs) =
     (if odd x then [x] else [])
  ++ odds xs
```

5 "Higher-order functions"

Define a polymorphic function including its function signature for duplicating elements that meet a certain condition. The condition is given as an argument of a function (predicate) type. Consider the following illustration:

```
> duplicate odd [1,2,3,4,5]
[1,1,3,3,5,5]
```

Reference solution

```
duplicate :: (x -> Bool) -> [x] -> [x]
duplicate _ [] = []
duplicate f (x:xs) =
     (if f x then [x,x] else [])
  ++ duplicate f xs
```

6 "Monoids"

A monoid must meet the property of associativity. Does the following definition meet this property? *Please, be concise: 140 characters or less.*

```
-- Import not required
import Data.Monoid

instance Monoid Float
  where
    mempty = 0
    mappend = (+)
```

Reference solution

If we assume that addition on floats is associative, then the property is met.

7 “Functors”

Consider the following type of non-empty lists:

```
data List1 x = One x | More x (List1 x)
```

Describe an instance of the type class Functor with its member function *fmap*, as needed for the lists at hand.

Reference solution

```
instance Functor List1
  where
    fmap f (One x) = One (f x)
    fmap f (More x l) = More (f x) (fmap f l)
```

8 “Reasoning”

Consider the following property:

```
import Test.QuickCheck

prop_map xs = sum (map (+1) xs) > sum xs
```

This ‘property’ is not universally true. Give an application of the ‘property’ for which it returns *False*.

Reference solution

```
> prop_map []
False
```

9 “Lazy evaluation”

Consider the following function:

```
foo :: [Int]
foo = map (+1) ([0]++foo)
```

Consider the following illustration:

```
> take 5 foo
[1,2,3,4,5]
```

How does the example depend on laziness? *Please, be concise: 140 characters or less.*

Reference solution

The function *foo* denotes an infinite list. Its mere definition does not compute the list though. Its use via *take* explicitly quantifies the prefix demaned.

10 “Monads”

Complete the following instance:

```
instance Monad Maybe
  where
    return x = Just x
    Nothing >>= f = ...
    (Just x) >>= f = ...
```

Reference solution

```
instance Monad Maybe
  where
    return x = Just x
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
```