NICCOLÒ
VETTORELLO

28/01/2021

1.1

2) first part

(→) is the arrow type. It takes two parameter types a and b, and represents a function (a→b)

NICCOLÒ
VETORELLO

28/01/2021

4)

```
foldr :: (a -> b -> b) -> b -> [a]
foldr _ v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

2.1

```
filter p xs = foldr g [] xs
    where g = \x fl -> if (p x) then x:fl
                       else fl
```

2)

second part

```
-- fmap :: (b->c) -> (a -> b) -> (a -> c)
fmap = (.)
```

```
fmap id x = id . x = x = id (x)
```

OK

↑ definition of fmap

↑ id is the identity element with respect to composition

↑ definition of id

2.2

NICCOLO'
VEITORELLO

28/01/2021

3) continued

executes f to obtain an mb

Monads are more flexible than applicatives

consider the following functions

```
ifM :: m Bool -> m a -> m a -> m a
ifM c t e = do
  v <- c
  if v then t else e
```

```
ifA :: a Bool -> ap -> ap -> ap
ifA c t e = pure (\c t e -> if c then t else e)
  <*> c <*> t <*> e
```

if we consider the type Maybe we have

```
ifM (Just True) (Just 1) Nothing : Just 1
```

but also

```
ifA (Just True) (Just 1) Nothing = Nothing
```

is less flexible, if one parameter is Nothing then all is Nothing

Niccolò
Vettorello
28/01/2021

5) continued

fmap :: (a → b → c → d) → f a → f b → f c → f d

the applicative is characterized by

pure :: a → q a

which «boxes» a value into an applicative *

and by

<*> :: q(a → b) → q a → q b

which abstracts the application

Monads are a more flexible extension of applicatives, and have

return :: a → m a

↑ same purpose of pure

(>>=) :: m a → (a → m b) → m b

which binds the value contained in ma into and

* fmap can be expressed in terms of pure and <*>

fmap f l = pure f <*> l

NICCOLO'
VETTORELLO

28/01/2021

5.1

3)

a functor is a generalization of a type on which a function can be mapped.
It's characterized by a function

```
fmap :: (a -> b) -> f a -> f b
```

which maps a function with one parameter on the elements of the functor. For example, the list type is an istance of functor with

```
fmap = map
```

An applicative is a generalization of this concept, where we don't restrict ourselves to have a function with only one parameter. In a sense an applicative represent the fallowing "generalization" of fmap

```
fmap :: a -> f a
fmap : (a -> b) -> f a -> f b
```

NICCOLO'
VETTORELLO

28/01/2021

1) Thesis [illegible] add n m = add m n

BC n = Zero

add Zero m = m by definition of add

6.1

add m Zero = m by property 1

OK

IC n = Succ x

by definition of add

add (Succ x) m = Succ (add x m)

6.2

= Succ (add m x) = add m (Succ m)

by inductive hypothesis

by property 2

OK

Indice dei commenti