FUNCTIONAL PROGRAMMING MT2018

SHEET 2

GABRIEL MOISE

3.1

As the class of ordered types is already declared, we can use it in order to write an instance declaration for ordered lists:

instance Ord a => Ord [a] where

(<) , (<=) , (>) , (>=) :: [a] -> [a] -> Bool

[] <= \_ = True

\_ <= [] = False

(x:xs) <= (y:ys) = (x < y) || ( (x==y) && (xs <= ys) )

The other 3 operators can be defined the same way as in the example from the sheet, as they eventually get defined using only (<=).

We started by defining the limit cases, and we started with [] <= \_ as if we get in the case of comparing two empty lists, we will first obtain True and we will stop. Then we treated case \_ <= [] to eventually get to comparing the first elements from each lists and from there applying the reasoning recursively. We can apply operators between elements because the class of ordered types is already declared in the text of the problem.

The program works because we sort two lists after comparing their first elements and in case of equality, by comparing the next elements, so the first elements of the remaining lists after we drop their heads.

3.2

So, we have h x y = f ( g x y ).

First of all, we need to understand how f, g, h look like in terms of domain and codomain.

Because function application is left-associative, we can use brackets to make everything clearer:

( h x ) y = f ( ( g x ) y )

So, h is a function that gets an a as parameter and returns a function that takes an a as a parameter and returns another a. Basically, h is declared like this:

h :: a -> ( a -> a )

Now we will talk about g, who is a function that behaves exactly like h, so:

g :: a -> ( a -> a )

Finally, f takes the value that g returns and returns a value in order to be equal to the one ( h x ) y returns. So,

f :: a -> a

Now we will check if the following statements are True or False:

1. h = f . g

f . g is the composition of f and g. However, f :: a -> a and g :: a -> ( a -> a ), so we would be able to compose them only if the codomain of g was equal to the domain of f, and this is False.

1. h x = f . g x

Firstly, h x :: a -> a, from the declaration of function h.

Secondly, f . g x is the same thing as f . ( g x ) , so we compose

f :: a -> a with g x :: a -> a to get

f . ( g x ) :: a -> a, as the domain of f is equal to the domain of ( g x )

So, we get that h x and f . ( g x ) have the same domain and codomain. Now we will test if they return the same value given a random value y.

(h x) (y) = h x y = f ( g x y ) from the hypothesis of the task

(f . ( g x )) (y) = f (( g x ) (y)) = f ( g x y), which is equal to (h x) (y), so they return the same result for any random y.

In conclusion, statement 2. is True.

1. h x y = ( f . g ) x y

As we stated at 1. we cannot compose f and g as they do not have the necessary requirement to be composed (the codomain of g is not equal to the domain of f).

So, statement 3. is False.

3.3

* subst f g x = ( f x ) ( g x )

We begin by allocating a type variable to each name:

subst :: A ; f :: B ; g :: C ; x :: D

Then:

subst f => A = B -> E

(subst f) g => E = C -> F

((subst f) g) x => F = D -> G

f x => B = D -> H

g x => C = D -> I

(f x) (g x) => H = I -> J

From the fact that subst f g x = ( f x ) ( g x ) we get that G = J.

Now, we will obtain A:

A = B -> E

A = ( D -> H ) -> ( C -> F )

A = ( D -> ( I -> J ) ) -> ( C -> ( D -> G ) )

A = ( D -> ( I -> J ) ) -> ( ( D -> I ) -> ( D -> J ) )

A = ( D -> I -> J ) -> ( ( D -> I ) -> D -> J ) )

So, in the end we obtained that

subst :: ( a -> b -> c) -> ( ( a -> b ) -> a -> c ) and this is the most general type.

* fix f = f ( fix f )

We begin by allocating a type variable to each name:

fix :: A ; f :: B

fix f => A = B -> C

f ( fix f ) => B = C -> D

From the fact that fix f = f ( fix f ) we get that C = D.

Now, we will obtain A:

A = B -> C

A = ( C -> D ) -> C

A = ( C -> C ) -> C

So, in the end we obtained that

fix :: ( a -> a ) -> a

* twice f = f . f

Let’s say that f :: A -> B. Because we can compose f and f, that means that A = B.

Therefore, f . f :: A -> A. Since twice f takes as parameter a function f and returns a function f . f, we can conclude that twice :: ( A -> A ) -> ( A -> A ), which is equivalent to

twice :: ( A -> A ) -> A -> A, or twice :: ( a -> a ) -> a -> a.

* selfie f = f f

We begin by allocating a type variable to each name:

selfie :: A ; f :: B.

selfie f => A = B -> C

f (f) => B = B -> D

From the fact that selfie f = f f, we get that C = D, therefore B = B -> C.

Now, we will try to calculate A:

A = B -> C

A = ( B -> C ) -> C

A = ( ( B -> C ) -> C ) -> C

A = ( ( ( B -> C ) -> C ) -> C ) -> C

A = ( ( ( ( B -> C ) -> C ) -> C ) -> C ) -> C

. . . .

and so on, we will never get to a result because we will always be able to replace B with ( B -> C ) and therefore the declaration of selfie will need an infinite number of steps.

By using ghci, we observe that if we declare selfie f = f f we get an error (“cannot construct the infinite type: t ~ t -> t1”), as the declaration will never stop and we will never get to a result.

3.4

1. [] : xs = xs

The syntax for the “:” operator is

element a of a list [a] : list [a]

So, for this to work syntactically we need that xs is a list of lists (that can also be lists of lists), as [] is a list of nothing (so it can be attached to any list of lists). However, this equation will not hold for any xs as xs will become a list of a greater number of lists (if initially xs had n lists, after []:xs, the new list will have (n+1) lists in it)

1. xs : [] = [xs]

As [] can is the empty list, its type is “list of anything” or [a], xs : [] meaning we attach the element xs, whatever it is, to an empty list. So, in the end we get [xs]. So this equality holds for all xs.

1. [[]] ++ xs = xs

For this to be syntactically correct we need xs to be a list of lists, as by concatenating [[]] we add [] to the elements of xs, which are also lists. However, there is no value of xs for which this equality holds, as xs will have more elements in the end than it previously had.

1. [[]] ++ [xs] = [[],xs]

As we concatenate a list of lists with a list of xs, we need xs to be a list. So, after the concatenation we get [[],xs], which is correct.

1. [] : xs = [[],xs]

In order to apply “:” we need xs to be a list of lists, and in the end we get [[],ys] where ys are the elements of xs, which are lists (of anything). So, the equality doesn’t hold for any xs.

1. xs : xs = [xs,xs]

From the LHS we get that xs is an element of itself, but this is impossible, so this is syntactically wrong.

1. [[]] ++ xs = [xs]

We need xs to be a list of lists as we want to concatenate it with [[]], so in the end we get [[],xs] as stated at d). So, although this is syntactically correct for xs being a list of lists, the equality doesn’t hold for any xs.

1. [xs] ++ [] = [xs]

So, xs can be anything, as []’s type is list of anything. By concatenating [xs] with [] we don’t add any new elements to [xs], so in the end we have [xs]. So, this is correct.

1. xs : [] = xs

By looking at b) we understand that although it is syntactically correct, the equality doesn’t hold for any xs.

1. xs : [xs] = [xs,xs]

As xs is an element of the type of the elements from [xs], we can apply “:” and we get a new list which has 2 xs: [xs,xs]. So, the equality is correct.

1. [[]] ++ xs = [[],xs]

As stated at d), in order for this to work it needs to be [[]] ++ [xs] = [[],xs], as xs are the elements of the final list we produce. So, this doesn’t hold for any xs.

1. [xs] ++ [xs] = [xs,xs]

This is correct because by concatenating two lists we obtain a list which contains the elements from the two lists. In this case, it contains two xs, so [xs,xs], so this is correct.

4.1

If f and g are strict, that would mean that f bottom = bottom and g bottom = bottom. So whenever one of the arguments of f or g is bottom, the function returns bottom. Now, f . g is strict if f . g bottom = bottom.

f . g bottom = f ( g bottom) = f bottom (as g is strict) = bottom (as f is strict).

Therefore, if f and g are strict, then f . g is also strict.

The converse implies that if f . g is strict then f and g are also strict.

As a counterexample, let’s consider const1 and const2, which composed make a strict function, however const2 is not strict:

inf :: Integer

inf = 1+inf

const1 :: a -> (a,a)

const1 x = (x,x)

const2 :: (a,b) -> a

const2 (x,y) = x

As we can see, const2 . const1 inf will not terminate, so const2 . const1 bottom will return bottom, therefore const2 . const1 is strict, however we can see that const2 is not a strict function because const2(x,inf) = x for any x, therefore const2 is only partially strict ( in the case where we want const2(inf,x) ). So, the converse is not True for all functions f and g.

4.2

Because now we count bottom as a Bool, the Bool set is formed of 3 elements. A function f from Bool to Bool will have a domain consisting of 3 elements and a codomain consisting of 3 elements. So, the number of functions of type Bool -> Bool is 33=27.

For a function f :: Bool -> Bool to be computable, f needs to be monotonic with respect to the fact that if x is less or equally useful than y, than f x is less of equally useful than f y. In the case of Bool values, only bottom is less useful than True and less useful than False, so if f bottom is True, we need that f True is True and also f False is also True.

So, we can compute any function that has the image in {True, False}. So, as the codomain has now 2 values ( we don’t want the bottom value here) and the domain has 3, the number of computable functions is 23=8.

First of all, we can define

inf :: Bool

inf = not inf -- which is in fact the bottom value for Bool here

f1, f2 :: Bool -> Bool

f1 x = True

f2 x = False

This way we have created 2 functions that actually work and give result for each argument.

The other 6 require different values for at least two arguments, so this will involve conditioning the arguments and using them, so we cannot define them in Haskell, as we would need to use our bottom value (inf), which will cause our program to not run properly. For example, if we try to create a f3 function

f3 :: Bool -> Bool

f3 x

| x == True = False

| otherwise = True

we would think that f3 inf would return True, but when we run f3 inf in the ghci, it goes silent as the program needs to compare inf with True, which causes him to enter an infinite loop, trying to determine if inf is True, or not.

So, as f1 and f2 ignore the argument and behave like constant functions, only those 2 can be defined in Haskell and actually work.

4.3

By evaluating (&&) with all possible pairs we obtain

1. False && False = False
2. False && True = False
3. False && undefined = False
4. True && False = False
5. True && True = True
6. True && undefined = error
7. undefined && False = error
8. undefined && True = error
9. undefined && undefined = error

The definition for (&&) function that would make it behave like that is

(&&) :: Bool -> Bool -> Bool

False && x = False

True && x = x

This way, the 1-3 equalities are correct given the fact that False && x = False for any x, 4 and 5 are correct because we set True && x = True for any x, 6 involves printing inf, which would create and infinite loop, and 7-9 imply calculating inf in order to see what is value is, so as to apply one of the two cases of the functions, which will also cause the program to go silent unless interrupted.

We create the function (&&&):

(&&&) :: Bool -> Bool -> Bool

False &&& y = False

x &&& False = False

True &&& True = True

And now we should test the 9 cases to see what happens:

1. False &&& False = False
2. False &&& True = False
3. False &&& undefined = False
4. True &&& False = False
5. True &&& True = True
6. True &&& undefined = error
7. undefined &&& False = error
8. undefined &&& True = error
9. undefined &&& undefined = error

We notice that (&&&) behaves exactly like (&&), but we would expect that undefined &&& False to return False, since that’s how our function is described. But, if we want undefined &&& False to work, we need to swap the first two cases, as when we call undefined &&& False, at first the program checks if undefined is False to be able to apply the first case or not, and that would mean the answer is undefined. By swapping the first two cases, when we try to calculate undefined &&& False, we first test x &&& False, whatever x is, so the program will instantly return False.

(&&&) :: Bool -> Bool -> Bool

x &&& False = False

False &&& y = False

True &&& True = True

However, this way if we try to calculate False &&& undefined, we will get undefined as the program will test the first case, x is False, but he doesn’t know if undefined is False or not, so the answer will be undefined.

?????????????????????????????

4.4

Song.hs