Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC Sr Eberl

WS 2020/21
05.02.2021
No homework

Functional Programming and Verification

Sheet 13

You made it – you (almost) survived FPV! To celebrate, we meet Friday evening for FPV & Chill (Volume 2) here. You can vote for your preferred time here on Zulip. Fire up your camera and mic, enjoy a semester review by the MCs, the unveiling of the artwork submitted as part of the *Schönheitswettbewerb*, the grand *Wettbewerb* awards and final ranking, and a good afterparty with your peers, playing games and chit-chat. See you there :)

And don't forget to join us at the Programming Contest on Wednesday, 17.00 – 19.00! It's a good exam preperation as well ;)

Tutorial Exercises

Exercise T13.1 Tail recursive functions I

Decide for each of the following functions whether they are tail recursive:

1.
```
prod :: Num a => a -> [a] -> a
prod n [] = n
prod n (m:ms) = prod (n*m) ms
```

2.
```
prod :: Num a => [a] -> a
prod [] = 1
prod (m:ms) = if m = 0 then 0 else m * prod ms
```

3.
```
prod :: Num a => a -> [a] -> a
prod n [] = n
prod n (m:ms) = if m == 0 then 0 else prod (n*m) ms
```

Exercise T13.2 Tail recursive functions II

Consider the function `concat :: [[a]] -> [a]` that concatenates a list of lists:

```
concat [[1,2],[],[5,6],[7]] = [1,2,5,6,7]
```

Give a tail recursive implementation of `concat`.

Exercise T13.3 Tail recursive functions III

Discuss: What are the benefits and disadvantages of tail recursive functions in a call-by-name language? Should we always aim to write our functions in a tail recursive manner?

Exam-style Exercises

Exercise T13.4 Inductive proof over lists

Prove that

```
map f (concat xss) = concat (map (map f) xss)
```

where

```
map f []      =  []
map f (x:xs)  =  f x : map f xs

concat []          =  []
concat (xs:xss)    =  xs ++ concat xss
```

You may use the Lemma `map_append`:
`map f (xs ++ ys) = map f xs ++ map f ys`

Exercise T13.5 QuickCheck test suite

Write one or more QuickCheck test for the function `sortP` as defined below. The tests should be complete, i.e. every correct implementation of `sortP` passes every test and for every incorrect implementation, there is at least one test that fails for suitable test parameters.

The function `sortP :: (Ord a, Eq b) => [(a,b)] -> [(a,b)]` sorts a list of tuples with respect to the first element of the tuple in ascending order. Tuples with the same first element may occur in any order.

Examples for correct behaviour:

```
sortP [(3,'a'), (1,'b'), (2,'c')] = [(1,'b'), (2,'c'), (3,'a')]
sortP [(3,'a'), (1,'b'), (3,'c')] = [(1,'b'), (3,'c'), (3,'a')]
sortP [(3,'a'), (1,'b'), (3,'c')] = [(1,'b'), (3,'a'), (3,'c')]
```

Examples for incorrect behaviour:

```
sortP [(3,'a'), (1,'b'), (2,'c')] = [(1,'a'), (2,'b'), (3,'c')]
sortP [(3,'a'), (1,'b'), (3,'c')] = [(1,'b'), (3,'a'), (3,'a')]
sortP [(3,'a'), (1,'b'), (2,'c')] = [(3,'a'), (2,'c'), (1,'b')]
```

Important: It is not required to implement the function `sortP`.

Exercise T13.6 Infer Me An Instance

Consider the classes `Semigroup` and `Monoid`:

```
class Semigroup a where
    (<>) :: a -> a -> a
```

```
class Semigroup m => Monoid m where
    mempty :: m
```

We define the type of pairs as follows:

```
data Pair a = Pair a a
```

Your task is to write instances of `Semigroup` and `Monoid` for `Pair` assuming that there are `Semigroup`/`Monoid` instances on the pair's carrier types. For `Semigroup`, you have to implement the operation `<>` which should combine two pairs by applying `<>` componentwise. Make sure the operation is associative:

```
Pair a b <> (Pair c d <> Pair e f) =
(Pair a b <> Pair c d) <> Pair e f
```

`Monoid` requires you to give a neutral element `mempty` with respect to `<>`, i.e:

```
Pair a b <> mempty = mempty <> Pair a b = Pair a b
```

Exercise T13.7 IO

We consider a game of matches for two players. In the beginning, there are 10 matches on the table. The players take turns in taking matches off the table (at least 1 and at most 5). The winner is the player who takes the last match.

Define an IO action `match :: IO ()` that implements the game of matches. Before any player's turn the program should print the number of the player and of the remaining matches. When a player wins the program should print the winner and exit afterwards. The program should ensure that the player only takes a valid number of matches. If not enough matches remain, the player takes all of them.

You can use the function `putStrLn :: String -> IO ()` to print a string to standard output and `readLn :: Read a => IO a` to read from standard input.

```
Matches: 10. Player 1?
4
Matches: 6. Player 2?
6
The input must be between 1 and 5.
5
Matches: 1. Player 1?
1
Player 1 wins!
```

Thanks for joining us this semster. The next 1100 Haskell programmers are ready to go. We hope you enjoyed the course :) Once again, special thanks to our tutors for helping us creating fun *Wettbewerb* exercises and running our workshops, programming contest, FPV & Chill, etc. We wish you all the best for the exam and hopefully see you at one of our chair's other lectures soon!

> Do you train for passing tests or do you train for creative inquiry?
> — Noam Chomksy in The Purpose of Education