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

WS 2019/20
05.01.2020
Deadline: 17.1.2020, 23:59

Functional Programming and Verification

Sheet 11

Tutorial Exercises

Exercise T11.1 Type Inference

Use the algorithm from the lecture (cf. page 170 of the slides) to determine the most general types of the following definitions:

```
f u v = min (head u) (last (concat v))
ffoldl = foldl . foldl
f x y = y : map (++x) y
```

where `foldl :: (a -> b -> a) -> a -> [b] -> a`.

Exercise T11.2 Repeated IO

Write a function

```
ioLoop :: (a -> Maybe b) -> IO a -> IO b
```

The invocation `ioLoop` *f a* should repeat the IO action *a* until *f* accepts its value (i.e. does not return `Nothing`). It should then return the result of *f*.

Use `ioLoop` and `readMaybe` from `Text.Read`, to write an IO action

```
getInt :: IO Int
```

that reads lines from stdin until it gets a line that represents a valid integer. It should then return this integer. You can read a line from stdin using the function `getLine` from `System.IO`.

Exercise T11.3 Guessing Game

In this exercise you will implement the following game: The program selects a random number in the range `[0..100]` and asks the user to guess the number. If the user's guess is incorrect, the program tells them whether the actual number is larger or smaller and waits for another guess. If the user guesses correctly, the program prints the number of guesses made.

Hints:

- Use `getInt` from the previous exercise to read numbers from the user.
- Use `randomRIO` from `System.Random` to get a random number in a certain range.

Exercise T11.4 Looking for an implementation

In this exercise we will work with the datatype `Either` from the Prelude, which is defined as follows:

```
data Either a b = Left a | Right b
```

As its name suggests, this type is useful when you have a value which can either be of type `a` or type `b`. You might for instance use `Either` for a function that parses integers: The function could have the type `String -> Either String Integer` where a successful parse returns an integer as a `Right` value, whereas an input that cannot be parsed returns an error message as a `Left` value.

Now to the task at hand: It can sometimes be interesting to determine an implementation for a given type signature. For example, the signature `Either a b -> Either b a` has the following implementation:

```
f :: Either a b -> Either b a
f (Left  x) = Right x
f (Right x) = Left  x
```

Your task is to write total functions that implement the following signatures:

```
g :: (a -> a') -> (b -> b') -> Either a b -> Either a' b'
h :: (a -> Either a b) -> a -> b
```

Your functions may not throw exceptions or be `undefined` and they should terminate on all inputs.

Homework

Note: On this exercise sheet, there is a task which requires you to use `IO`. The test system uses a mocked version of `IO` which defines the most important `IO` functions from `Prelude` and `System.IO`. Usually, there is difference in the behaviour of your program between your local Haskell installation and the submission system. The usability of `IO` functions from other modules cannot be guaranteed however. As always, the submission is only graded if it compiles on the test server. Any problems should be brought to our attention immediately.

You need to pass all tests on the submission server to pass this sheet.

Exercise H11.1 Can't see the filesystem for the directories

Originally, this task would have been about Christmas trees but then the Übunsgsleitung noticed that Christmas trees have nothing to do with programming. This is why we consider filesystems which we define as a recursive datatype as follows:

```
type Name = String
type Data = String
data FSItem = File Name Data | Directory Name [FSItem]
  deriving (Eq, Show)
```

As the derived `show` function of `FSItem` doesn't let us see the filesystem for the directories, your task is to write a procedure

```
pretty :: FSItem -> String
```

that prints the filesystem as a tree (similar to the `tree` command which is available on most Linux distributions). In particular, the command should print the directory `Directory "Last" [...]` as `last/`. Files `File "christmas.wav" "Last christmas I..."` are represented by the String `christmas.wav "Last christmas I..."` All subdirectories and files in directory are to be indented by two spaces and are prepended by `|-`. The procedure has to first print all subdirectories of a directory and then the files in the directory, each according to lexicographic order. Below is an example of a beatiful tree that `pretty` generates.

```
>>> fs = Directory "" [
        Directory "Last" [
            Directory "Christmas" [ File "heart.wav" "I gave you" ],
            File "Merry" "Christmas"
        ], File "proof.cprf" "Stop Lemma Time"]
>>> pretty fs
/
|- Last/
  |- Christmas/
    |- heart.wav "I gave you"
  |- Merry "Christmas"
|- proof.cprf "Stop Lemma Time"
```

As a static filesystem is not very useful, we want you to implement some operations to modify the filesystem:

1. `mkdir` takes an absolute path describing a directory and creates said directory.
2. `rm` remove the item corresponding to the given absolute path.
3. `touch` creates an empty file at the given absolute path.
4. `edit` takes an absolute path to a file and a string (without linebreaks) enclosed by double quotes. It overwrites the contents of the file with the string.

The operations `mkdir` and `touch` don't modify the filesystem if the directory respectively file already exists. You may assume that the given path always exists in the filesystem for the other commands. All of the above operations need to navigate and modify the filesystem. To implement this, we recommend that you look at the chapter about Zippers of the excellent book "Learn You a Haskell for Great Good!", which applies the concept of Zippers to filesystems and other data structures.

Your task is now to implement a rudimentary interactive shell that supports the above commands. Initially, the filesystem is empty, i.e. it is equal to `Directory "" []`. The program should listen for commands on standard input and output the state of the filesystem with `pretty` after each operation. If the command `quit` is encountered the program should exit after printing `Bye!`. The console dump of an examplary session is shown below:

```
>>> touch /happy
/
|- happy ""

>>> mkdir /new/
/
|- new/
|- happy ""

>>> mkdir /new/year
/
|- new/
  |- year/
|- happy ""

>>> edit /happy "2020"
/
|- new/
  |- year/
|- happy "2020"

>>> quit
```

```
Bye!
```