Class 2: Types and Classes,[*] v1.4

Jacek Malec (but see Introduction)

November 23, 2015

1 Introduction

This document is a derivative of many sources, most notably (in no particular order):

1. "Haskell" on WikiBooks: `http://en.wikibooks.org/wiki/Haskell`;
2. Bryan O'Sullivan, John Goerzen and Don Stewart, "Real World Haskell", O'Reilly, 2009, also: `http://book.realworldhaskell.org`;
3. Simon Thompson, "The Craft of Functional Programming", 2nd ed., (formally our course book), Addison-Wesley, 1999;
4. Manuel M. T. Chakravarty, course materials COMP1011 at University of New South Wales;
5. Materials of TDA555 from Chalmers,
`http://www.cse.chalmers.se/edu/course/TDA555/`;
6. Course material from Lennart Ohlsson.

NOTE: Recent changes in the text below include addition of the type derivation exercise, taken from previous exams.

2 Exercises

2.1 Propositional Logic (TDA555)

A proposition is a boolean formula of one of the following forms:

- a variable name (a string)
- $p \wedge q$ (and)
- $p \vee q$ (or)
- $\neg p$ (not)

where p and q are propositions. For example, $p \vee \neg p$ is a proposition.

[*]Intended for EDAN40 course, after the lecture on types.

1. Design a data type `Proposition` to represent propositions.

2. Define a function

   ```
   vars :: Proposition -> [String]
   ```

   which returns a list of the variables in a proposition. Make sure each variable appears only once in the list you return.

   Suppose you are given a list of variable names and their values, of type `Bool`, for example, `[("p",True),("q",False)]`. Define a function

   ```
   truthValue :: Proposition -> [(String,Bool)] -> Bool
   ```

   which determines whether the proposition is true when the variables have the values given.

3. Define a function

   ```
   tautology :: Proposition -> Bool
   ```

   which returns true if the proposition holds for all values of the variables appearing in it.

2.2 File Systems (TDA555)

A file either contains data or is a directory. A directory contains other files (which may themselves be directories) along with a name for each one.

1. Design a data type to represent the contents of a directory. Ignore the contents of files: you are just trying to represent file names and the way they are organised into directories here.

2. Define a function to search for a given file name in a directory. You should return a path leading to a file with the given name. Thus if your directory contains a, b, and c, and b is a directory containing x and y, then searching for x should produce b/x.

2.3 Sets (TDA555)

1. Design a datastructure for sets . I.e. there should be a type `Set a`, and a number of functions for creating, combining, and investigating sets. There should at least be a function to create an empty set, add an element to a set, take the union of two sets, remove an element from the set, and check if an element is in the set.

2. Now, implement the Set datastructure. You may use lists internally.

3. Redo the above exercise, but now use sorted lists of unique elements as your internal representation. Set union becomes more efficient that way.

2.4 Ordering (Thompson)

Complete the following instance declarations:

```
instance (Ord a, Ord b) => Ord (a,b) where ...
instance Ord b => Ord [b] where ...
```

where pairs and lists should be ordered lexicographically, like the words in dictionary.

2.5 ListNatural (lecture)

Natural numbers may correspond to lists of nothing!!

```
type ListNatural = [()]
```

For example:

```
twoL = [(),()]
threeL = [(),(),()]
```

What is: `(:)`
What is: `(++)`
What is: `map (const ())`

1. What do these functions do?

   ```
   f1 x y = foldr (:) x y
   f2 x y = foldr (const (f1 x)) [] y
   f3 x y = foldr (const (f2 x)) [()] y
   ```

2. Continue this definition:

   ```
   instance Num ListNatural where ...
   ```

   Note: This requires `ListNatural` to be declared as a `newtype`[1]. One can ask: Why?

2.6 Type derivation

Give the types of the following expressions:

1. `(.)(:)`
2. `(:(.))`
3. `((.):)`
4. `((:):)`
5. Haskel wheels: `(.)(.)`
6. The Haskell smiley: `(8-)`
7. Haskell goggles: `(+0).(0+)`
8. A Haskell treasure: `(($)$($))`
9. Haskell swearing: `([]>>=)(\_->[(>=)])`

[1]The `newtype` construct is explained e.g. on the Haskell wiki: `http://haskell.org/haskellwiki/Newtype`.