Functional Programming 2

Practicals

Ralf Hinze

0 Getting started

Like in "Functional Programming 1", we will be using GHCi for the practicals (https://www.haskell.org/ghc/). To run GHCi, simply open a terminal window and type 'ghci'. One typically uses a text editor to write or edit a Haskell script, saves that to disk, and loads it into GHCi. To load a script, it is helpful if you run GHCi from the directory containing the script. You can simply give the name of the script file as a parameter to the command ghci. Or, within GHCi, you can type ':load' (or just ':l') followed by the name of the script to load, and ':reload' (or just ':r') with no parameter to reload the file previously loaded.

There are practicals for each of the lectures; most of the practicals are programming exercises, but some can also be solved using pencil and paper. For some of the exercises there are skeletons of a solution to save you from having to type in what is provided to start with. The header of each exercise provides some guidance, e.g. "**Exercise 1.1** (Warm-up: constructing a frequency table, Huffman.lhs)" indicates that this is a warm-up exercise and that there is a source file available, called Huffman.lhs. The skeleton files, the slides of the lectures, and these instructions can be obtained via one of the following commands (you need to have Git installed, see https://git-scm.com/):

```
git clone git@gitlab.science.ru.nl:ralf/FP2.git
git clone https://gitlab.science.ru.nl/ralf/FP2.git
```

The Git repository will be updated on a regular basis. To obtain the latest updates, simply type git pull in the directory FP2. If you encounter any problems, please see the teaching assistants.

We will not introduce Haskell's module system in the lectures. However, for solving the exercises some basic knowledge is needed. Appendix A provides an overview of the most essential features.

Haskell and GHC fully support unicode, see www.unicode.org and https://wiki.haskell.org/Unicode-symbols. Both the lectures and the practicals make fairly intensive use of this feature e.g. we usually write '→' instead of '->'. Unicode support in the source code is turned on using a language pragma:

```
{-# LANGUAGE UnicodeSyntax #-}
module Database
where
import Unicode
```

The module *Unicode.lhs*, which can be found in the repository, defines a few obvious unicode bindings e.g. '∧' for conjunction '&&', '⩽' for ordering '<=', and '∘' for function composition '.'. Of course, the use of Unicode is strictly optional.

Have fun! Ralf Hinze

1 Recap: Functional Programming 1

The exercises below evolve around the implementation of a Huffman coding program, which provides a mechanism for compressing ASCII text. The purpose of the exercises is to allow you to apply what you have learned in "Functional Programming 1" to a slightly larger task. (However, even though the exercises share a common theme, they are fairly independent e.g. you can attempt the last exercise without having solved the earlier ones.)

The conventional representation of a textual document on a computer uses the ASCII character encoding scheme. The scheme uses seven or eight bits to represent a single character; a document containing 1024 characters will therefore occupy one kilobyte. The idea behind Huffman coding is to use the fact that some characters appear more frequently in a document. Therefore, Huffman encoding moves away from the fixed-length encoding of ASCII to a variable-length encoding in which the frequently used characters have a smaller bit encoding than rarer ones. As an example, consider the text

```
hello world
```

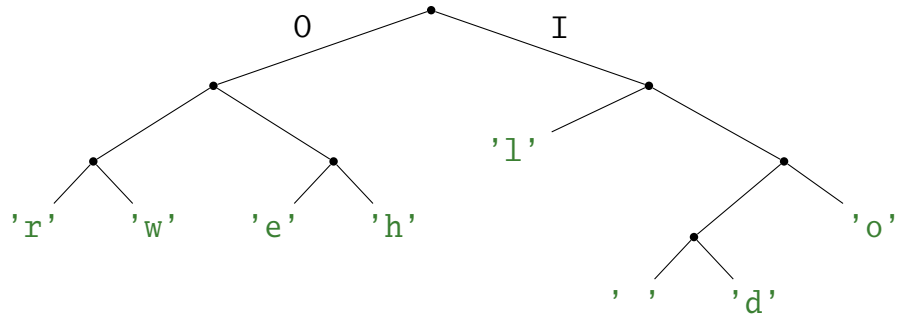
and note that the letter `l` occurs thrice. The table shown below gives a possible Huffman encoding for the eight letters:

`'r'`	`000`	`'e'`	`010`	`'l'`	`10`	`'d'`	`1101`
`'w'`	`001`	`'h'`	`011`	`' '`	`1100`	`'o'`	`111`

The more frequent a character, the shorter the code. Using this encoding the ASCII string above is Huffman encoded to the following data:

```
01101010101111100001111000101101
```

It is important that the codes are chosen in such a way that an encoded document gives a unique decoding that is the same as the original document. The reason why the codes shown above are decipherable is that *no code is a prefix of any other code*. This property is easy to verify if the code table is converted to a code tree:



```
As software becomes more and more complex, it
is more and more important to structure it
well. Well-structured software is easy to
write, easy to debug, and provides a
collection of modules that can be re-used to
reduce future programming costs. Conventional
languages place a conceptual limit on the way
problems can be modularised. Functional
languages push those limits back. In this
paper we show that two features of functional
languages in particular, higher-order
functions and lazy evaluation, can contribute
greatly to modularity. Since modularity is
the key to successful programming, functional
languages are vitally important to the real
world.
```

Table 1: Excerpt of “Why Functional Programming Matters” by J. Hughes

Each code corresponds to a path in the tree e.g. starting at the root the code `OII` guides us to `'h'` i.e. we walk left, right, and right again. Given a code tree and an encoded document, the original text can be decoded.

For a slightly larger example, consider the text shown in Table 1. The document is 696 characters long, but contains only 35 different characters (including the newline character `'\n'`). The shortest Huffman code for this document is 3 bits long (for `' '`); the longest codes comprise 10 bits (for `'A'`, `'C'`, `'F'`, `'I'`, `'S'`, `'W'`, `'x'`, and `'z'`). The Huffman encoded document has a total length of 3019 bits. As the original text is $8 \cdot 696 = 5568$ bits long, we obtain a compression factor of almost 2. (Of course, this is not quite true as in reality we also need to store the code tree along the compressed data.)

Strange results are obtained if the encoded document contains just one character, repeated many times—for which the Huffman tree consists only of a single leaf, and the path describing it is empty. Therefore, we assume that the to-be-encoded text contains at least *two* different characters.

Exercise 1.1 (Warm-up: constructing a frequency table, `Huffman.lhs`)**.** Define a function

```
frequencies :: (Ord char) ⇒ [char] → [With Int char]
```

that constructs a frequency table, a list of frequency-character pairs, that records the number of occurrences of each character within a piece of text. For example,

```
⟩⟩⟩⟩ frequencies "hello world"
[1 :− ' ',1 :− 'd',1 :− 'e',1 :− 'h',3 :− 'l',2 :− 'o',1 :− 'r',1 :− 'w']
```

The datatype *With a b* is similar to the pair type (a, b), but with a twist: when comparing two pairs only the first component is taken into account:

```
infix 1 :−
data With a b = a :− b
instance (Eq a) ⇒ Eq (With a b) where
  (a :− _) == (b :− _) = a == b
instance (Ord a) ⇒ Ord (With a b) where
  (a :− _) ⩽ (b :− _) = a ⩽ b
```

We use *With a b* instead of (a, b) as this slightly simplifies the next step.

Exercise 1.2 (Constructing a Huffman tree, `Huffman.lhs`)**.** A Huffman tree or simply a code tree is an instance of a *leaf tree*, a tree in which all data is held in the leaves. Such a tree can be defined by the datatype declaration

```
data Tree elem = Leaf elem | Tree elem :^: Tree elem
```

The algorithm for constructing a Huffman tree works as follows:

- sort the list of frequencies on the *frequency* part of the pair—i.e. less frequent characters will be at the front of the sorted list;
- convert the list of frequency-character pairs into a list of frequency-tree pairs, by mapping each character to a leaf;
- take the first two pairs off the list, add the frequencies and combine the trees to form a branch; insert this pair into the remaining list of pairs in such a way that the resulting list is still sorted on the frequency part;

- repeat the previous step until a singleton list remains, which contains the Huffman tree for the character-frequency pairs.

(Do you see why the special pair type *With Int char* is useful?)

For example, for the sorted list of frequencies

```
[1:-' ',1:-'d',1:-'e',1:-'h',1:-'r',1:-'w',2:-'o',3:-'l']
```

the algorithm takes the following steps (*L* is shorthand for *Leaf*):

```
[1:-L' ',1:-L'd',1:-L'e',1:-L'h',1:-L'r',1:-L'w',2:-L'o',3:-L'l']
[1:-L'e',1:-L'h',1:-L'r',1:-L'w',2:-(L' ':^:L'd'),2:-L'o',3:-L'l']
[1:-L'r',1:-L'w',2:-(L'e':^:L'h'),2:-(L' ':^:L'd'),2:-L'o',3:-L'l']
[2:-(L'r':^:L'w'),2:-(L'e':^:L'h'),2:-(L' ':^:L'd'),2:-L'o',3:-L'l']
[2:-(L' ':^:L'd'),2:-L'o',3:-L'l',4:-((L'r':^:L'w'):^:(L'e':^:L'h'))]
[3:-L'l',4:-((L' ':^:L'd'):^:L'o'),4:-((L'r':^:L'w'):^:(L'e':^:L'h'))]
[4:-((L'r':^:L'w'):^:(L'e':^:L'h')),7:-(L'l':^:((L' ':^:L'd'):^:L'o'))]
[11:-(((L'r':^:L'w'):^:(L'e':^:L'h')):^:(L'l':^:((L' ':^:L'd'):^:L'o')))]
```

First, the characters that occur only once are combined. The most frequent character, `'l'`, is considered only in the second, but last step, which is why it ends up high in the tree. The final tree is the Haskell rendering of the code tree shown in the introduction above.

1. Write a function

   ```
   huffman :: [With Int char] → Tree char
   ```

   that constructs a code tree from a frequency table. For example,

   ```
   ⟩⟩⟩⟩ huffman (frequencies "hello world")
   ((Leaf 'r' :^: Leaf 'w') :^: (Leaf 'e' :^: Leaf 'h'))
     :^: (Leaf 'l' :^: ((Leaf ' ' :^: Leaf 'd') :^: Leaf 'o'))
   ```

   yields the Huffman tree shown above.

2. Apply the algorithm to the relative frequencies of letters in the English language, see for example https://en.wikipedia.org/wiki/Letter_frequency.

3. *Optional:* lists are the functional programmers favourite data structure but they are not always appropriate. The algorithm above uses an ordered list to maintain frequency-tree pairs. A moment's reflection reveals that the ordered list really serves as a *priority queue*,

where priorities are given by frequencies. The central step of the algorithm involves extracting two pairs with minimum frequency and inserting a freshly created pair. Both of these operations are well supported by priority queues. If you feel energetic, implement a priority queue and use the implementation to replace the type of ordered lists.

Exercise 1.3 (Encoding ASCII text, `Huffman.lhs`)**.** It is now possible to Huffman encode a document represented as a list of characters.

1. Write a function

   ```
   data Bit = O | I
   encode :: (Eq char) ⇒ Tree char → [char] → [Bit]
   ```

   that, given a code tree, converts the list of characters into a sequence of bits representing the Huffman coding of the document. For example,

   ```
   ⟩⟩⟩⟩ ct = huffman (frequencies "hello world")
   ⟩⟩⟩⟩ encode ct "hello world"
   [O,I,I,O,I,O,I,O,I,O,I,I,I,I,I,O,O,O,O,I,I,I,I,O,O,O,I,O,I,I,O,I]
   ```

   Test your function on appropriate test data, cutting and pasting the example evaluations into your file. *Hint:* it is useful to first implement a function

   ```
   codes :: Tree char → [(char,[Bit])]
   ```

   that creates a code table, a character-code list, from the given code tree. This greatly simplifies the task of mapping each character to its corresponding Huffman code. For example,

   ```
   ⟩⟩⟩⟩ codes ct
   [('r',[O,O,O]),('w',[O,O,I]),('e',[O,I,O]),('h',[O,I,I]),
    ('l',[I,O]),(' ',[I,I,O,O]),('d',[I,I,O,I]),('o',[I,I,I])]
   ```

2. *Optional:* lists are the functional programmers favourite data structure but they are not always appropriate. The function *codes* should really yield a *finite map* (aka dictionary or look-up table), which maps characters to codes. An association list, a list of key-value pairs, is a simple implementation of a finite map. Better implementations include balanced search trees or tries. If you feel energetic, implement a finite map and use the implementation to replace the type of association lists.

Exercise 1.4 (Decoding a Huffman binary, `Huffman.lhs`)**.** Finally, write a function

```
decode :: Tree char → [Bit] → [char]
```

that, given a code tree, converts a Huffman-encoded document back to the original list of characters. For example,

```
⟩⟩⟩⟩ ct = huffman (frequencies "hello world")
⟩⟩⟩⟩ encode ct "hello world"
[O, I, I, O, I, O, I, O, I, O, I, I, I, I, I, O, O, O, O, I, I, I, I, O, O, O, I, O, I, I, O, I]
⟩⟩⟩⟩ decode ct it
"hello world"
```

Again, test your function on appropriate test data, cutting and pasting the example evaluations into your file. In general, decoding is the inverse of encoding: $decode\ ct \circ encode\ ct = id$. Use this relationship to test your program more thoroughly.

A Modules

Haskell has a relatively simple module system which allows programmers to create and import modules, where a *module* is simply a collection of related types and functions.

A.1 Declaring modules

Most projects begin with something like the following as the first line of code:

```
module Main
where
```

This declares that the current file defines functions to be held in the *Main* module. Apart from the *Main* module, it is recommended that you name your file to match the module name. So, for example, suppose you were defining a number of protocols to handle various mailing protocols, such as POP3 or IMAP. It would be sensible to hold these in separate modules, perhaps named *Network.Mail.POP3* and *Network.Mail.IMAP*, which would be held in separate files. Thus, the POP3 module would have the following line near the top of its source file.

```
module Network.Mail.POP3
where
```

This module would normally be held in a file named

```
src/Network/Mail/POP3.hs .
```

Note that while modules may form a hierarchy, this is a relatively loose notion, and imposes nothing on the structure of your code.

By default, all of the types and functions defined in a module are exported. However, you might want certain types or functions to remain private to the module itself, and remain inaccessible to the outside world. To achieve this, the module system allows you to explicitly declare which functions are to be exported: everything else remains private. So, for example, if you had defined the type *POP3* and functions *send*::*POP3* → *IO* () and *receive*::*IO POP3* within your module, then these could be exported explicitly by listing them in the module declaration:

```
module Network.Mail.POP3 (POP3 (..), send, receive)
```

Note that for the type *POP3* we have written *POP3* (..). This declares that not only do we want to export the *type* called *POP3*, but we also want to export all of its constructors too.

A.2 Importing modules

The *Prelude* is a module which is always implicitly imported, since it contains the definitions of all kinds of useful functions such as $map :: (a \to b) \to (f\ a \to f\ b)$. Thus, all of its functions are in scope by default. To use the types and functions found in other modules, they must be imported explicitly. One useful module is the *Data.Maybe* module, which contains useful utility functions:

$$\begin{aligned} &maybe &&:: b \to (a \to b) \to Maybe\ a \to b \\ &catMaybes &&:: [Maybe\ a] \to [a] \end{aligned}$$

Importing all of the functions from *Data.Maybe* into a particular module is done by adding the following line below the module declaration, which imports every entity exported by *Data.Maybe*

```
import Data.Maybe
```

It is generally accepted as good style to restrict the imports to only those you intend to use: this makes it easier for others to understand where some of the unusual definitions you might be importing come from. To do this, simply list the imports explicitly, and only those types and functions will be imported:

```
import Data.Maybe (maybe, catMaybes)
```

This imports *maybe* and *catMaybes* in addition to any other imports expressed in other lines.

A.3 Qualifying and hiding imports

Sometimes, importing modules might result in conflicts with functions that have already been defined. For example, one useful module is *Data.Map*. The base datatype that is provided is *Map* which efficiently stores values indexed by some key. There are a number of other useful functions defined in this module:

$$\begin{aligned} &empty &&:: Map\ k\ v \\ &insert &&:: (Ord\ k) \Rightarrow k \to v \to Map\ k\ v \to Map\ k\ v \\ &update &&:: (Ord\ k) \Rightarrow k \to Map\ k\ v \to Maybe\ v \end{aligned}$$

It might be tempting to import *Map* and these auxiliary functions as follows:

```
import Data.Map (Map (..), empty, insert, lookup)
```

However, there is a catch here! The *lookup* function is initially always implicitly in scope, since the *Prelude* defines its own version. There are a number of ways to resolve this. Perhaps the most common solution is to qualify the import, which means that the use of imports from *Data.Map* must be prefixed by the module name. Thus, we would write the following instead as the import statement:

```
import qualified Data.Map
```

To actually use the functions and types from *Data.Map*, this prefix would have to be written explicitly. For example, to use *lookup*, we would actually have to write *Data.Map.lookup* instead.

These long names can become somewhat tedious to use, and so the qualified import is usually given as something different:

```
import qualified Data.Map as M
```

This brings all of the functionality of *Data.Map* to be used by prefixing with *M* rather than *Data.Map*, thus allowing you to use *M.lookup* instead.

Another solution to module clashes is to hide the functions that are already in scope within the module by using the *hiding* keyword:

```
import Prelude hiding (lookup)
```

This will override the *Prelude* import so that the definition of *lookup* is excluded.