Functional Programming 1

Practicals

Ralf Hinze

0 Getting started

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: definitions, `Database.lhs`)" indicates that this is a warm-up exercise training primarily the concept of "definitions" and that there is a source file available, called `Database.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/FP1.git
git clone https://gitlab.science.ru.nl/ralf/FP1.git
```

The Git repository will be updated on a regular basis. To obtain the latest updates, simply type `git pull` in the directory `FP1`. 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. '$\wedge$' for conjunction '&&', '$\leqslant$' for ordering '<=', and '$\circ$' for function composition '.'. Of course, the use of Unicode is strictly optional.

Have fun! Ralf Hinze

1 Programming with expressions and values

Exercise 1.1 (Warm-up: definitions, `Database.lhs`). Consider the following definitions, which introduce a type of persons and some sample data.

```
type Person = (Name, Age, FavouriteCourse)
type Name            = String
type Age             = Integer
type FavouriteCourse = String
frits, peter, ralf :: Person
frits = ("Frits", 33, "Algorithms and Data Structures")
peter = ("Peter", 57, "Imperative Programming")
ralf  = ("Ralf",  33, "Functional Programming")
students :: [Person]
students = [frits, peter, ralf]
```

1. Add your own data and/or invent some additional entries. In particular, add yourself to the list of students.

2. The function *age* defined below extracts the age from a person, e.g. *age ralf* ⟹ 33. (In case you wonder why some variables have a leading underscore see Hint 1.)

   ```
   age :: Person → Age
   age (_n, a, _c) = a
   ```

   Define functions

   ```
   name            :: Person → Name
   favouriteCourse :: Person → FavouriteCourse
   ```

   that extract name and favourite course, respectively.

3. Define a function *showPerson* :: *Person* → *String* that returns a string representation of a person. You may find the predefined operator ++ useful, which concatenates two strings e.g. `"hello, "` ++ `"world\n"` ⟹ `"hello, world\n"`. *Hint:* *show* converts a value to a string e.g. *show* 4711 = `"4711"`.

4. Define a function *twins* :: *Person* → *Person* → *Bool* that checks whether two persons are twins. (For lack of data, we agree that two persons are twins if they are of the same age.)

5. Define a function *increaseAge* :: *Person* → *Person* which increases the age of a given person by one e.g.

   ```
   ⟩⟩⟩⟩ increaseAge ralf
   ("Ralf",34,"Functional Programming")
   ```

6. The function *map* takes a function and a list and applies the function to each element of the list e.g.

   ```
   ⟩⟩⟩⟩ map age students
   [33,57,33]
   ⟩⟩⟩⟩ map (\p → (age p, name p)) students
   [(33,"Frits"),(57,"Peter"),(33,"Ralf")]
   ```

   The function *filter* applied to a predicate and a list returns the list of those elements that satisfy the predicate e.g.

   ```
   ⟩⟩⟩⟩ filter (\p → age p > 50) students
   [("Peter",57,"Imperative Programming")]
   ⟩⟩⟩⟩ map (\p → (age p, name p)) (filter (\p → age p > 50) students)
   [(57,"Peter")]
   ```

   Create expressions to solve the following tasks: a) increment the age of all students by two; b) promote all of the students (attach `"dr "` to their name); c) find all students named Frits; d) find all students whose favourite course is Functional Programming; e) find all students who are in their twenties; f) find all students whose favourite course is Functional Programming and who are in their twenties; g) find all students whose favourite course is Imperative Programming or who are in their twenties.

Exercise 1.2 (Pencil and paper: evaluation). 1. Recall the implementation of Insertion Sort from §0.4 (listed below, with some minor modifications).

```
insertionSort :: [Integer] → [Integer]
insertionSort []       = []
insertionSort (x : xs) = insert x (insertionSort xs)

insert :: Integer → [Integer] → [Integer]
insert a [] = a : []
insert a (b : xs)
  | a ⩽ b = a : b : xs
  | a > b = b : insert a xs
```

The function *insert* takes an element and an ordered list and inserts the element at the appropriate position e.g.

$$
\begin{array}{ll}
 & \mathit{insert}\ 7\ (2 : (9 : [\,]))\\
\Longrightarrow & \quad \{\ \text{definition of } \mathit{insert} \text{ and } 7 > 2\ \}\\
 & 2 : (\mathit{insert}\ 7\ (9 : [\,]))\\
\Longrightarrow & \quad \{\ \text{definition of } \mathit{insert} \text{ and } 7 \leqslant 9\ \}\\
 & 2 : (7 : (9 : [\,]))
\end{array}
$$

Recall that Haskell has a very simple computational model: an expression is evaluated by repeatedly replacing equals by equals. Evaluate the expression *insertionSort* $(7 : (9 : (2 : [\,])))$—by hand, using the format above. (We have not yet discussed lists in any depth, but I hope you will be able to solve the exercise anyway. The point is that evaluation is a purely mechanical process—this is why a computer is able to perform the task.)

2. The function twice applies its first argument twice to its second argument.

   $\mathit{twice}\ f\ x = f\ (f\ x)$

   (Like *map* and *filter*, it is an example of a higher-order function as it takes a function as an argument.) Evaluate *twice* $(+1)\ 0$ and *twice twice* $(*2)\ 1$ by hand. Use the computer to evaluate

```
⟩⟩⟩⟩  twice ("|"++) ""
⟩⟩⟩⟩  twice twice ("|"++) ""
⟩⟩⟩⟩  twice twice twice ("|"++) ""
⟩⟩⟩⟩  twice twice twice twice ("|"++) ""
⟩⟩⟩⟩  twice (twice twice) ("|"++) ""
⟩⟩⟩⟩  twice twice (twice twice) ("|"++) ""
⟩⟩⟩⟩  twice (twice (twice twice)) ("|"++) ""
```

   Is there any rhyme or rhythm? Can you identify any pattern?

Exercise 1.3 (Pencil and paper: λ-expressions). 1. An alternative definition of *twice* builds on λ-expressions.

   $\mathit{twice} = \backslash f \to \backslash x \to f\ (f\ x)$

Re-evaluate $\mathit{twice}\ (+1)\ 0$ and $\mathit{twice}\ \mathit{twice}\ (*2)\ 1$ using this definition. You need to repeatedly apply the evaluation rule for λ-expressions (historically known as the β-rule).

$$(\backslash x \to \mathit{body})\ \mathit{arg} \Longrightarrow \mathit{body}\ \{x := \mathit{arg}\}$$

A function applied to an argument reduces to the body of the function where every occurrence of the formal parameter is replaced by the actual parameter e.g. $(\backslash x \to x + x)\ 47 \Longrightarrow x + x\ \{x := 47\} \Longrightarrow 47 + 47 \Longrightarrow 94$.

2. It is perhaps slightly worrying that you can apply a function to itself (as in $\mathit{twice}\ \mathit{twice}\ (*2)\ 1 = ((\mathit{twice}\ \mathit{twice})\ (*2))\ 1)$. Can you guess the type of *twice*?

Exercise 1.4 (Worked example: prefix and infix notation).

1. Haskell features both alphabetic identifiers, written *prefix* e.g. $\mathit{sin}\ \mathit{pi}$, and symbolic identifiers, written *infix* e.g. $2 + 7$. The use of infix notation for addition is traditional. (The symbol "+" is a simplification of "et", Latin for "and".) Most programming languages (with the notable exception of LISP) have adopted infix notation. But is this actually a wise thing to do? What are the advantages and disadvantages of infix over prefix (or postfix) notation. Discuss!

2. Infix notation is inherently ambiguous: $x \otimes y \otimes z$. What does this mean: $(x \otimes y) \otimes z$ or $x \otimes (y \otimes z)$? To disambiguate without parentheses, operators may *associate* to the left or to the right. Subtraction associates to the left: $5 - 4 - 2 = (5 - 4) - 2$. Why? Concatenation of strings associates to the right: $\texttt{"F"} +\!\!+ \texttt{"P"} +\!\!+ \texttt{"1"} = \texttt{"F"} +\!\!+ (\texttt{"P"} +\!\!+ \texttt{"1"})$. Why? Haskell allows the programmer to specify the *association* of an operator using a *fixity declaration*:

```
infixl -
infixr ++
```

Function application can be seen as an operator ("the space operator") and associates to the left: $f\ a\ b$ means $(f\ a)\ b$. (On the other hand, the "function type" operator associates to the right: $\mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}$ means $\mathit{Integer} \to (\mathit{Integer} \to \mathit{Integer})$.) Haskell also features an explicit operator for function application, which associates to the right: $f \mathbin{\$} g \mathbin{\$} a$ means $f \mathbin{\$} (g \mathbin{\$} a) = f\ (g\ a)$. Can you foresee possible use-cases?

3. The operator

   $$\textbf{infixl} \otimes$$
   $$a \otimes b = 2 * a + b$$

   can be used to capture binary numbers e.g. $1 \otimes 0 \otimes 1 \otimes 1 \Longrightarrow 11$ and $(1 \otimes 1 \otimes 0) + 4711 \Longrightarrow 4717$. The fixity declaration determines that $\otimes$ associates to the left. Why this choice? What happens if we declare $\textbf{infixr}\otimes$?

4. Association does not help when operators are mixed: $x \oplus y \otimes z$. What does this mean: $(x \oplus y) \otimes z$ or $x \oplus (y \otimes z)$? To disambiguate without parentheses, there is a notion of *precedence* (or binding power), eg $*$ has higher precedence (binds more tightly) than $+$.

   $$\textbf{infixl}\ 7\ *$$
   $$\textbf{infixl}\ 6\ +$$

   The precedence level ranges between 0 and 9. Function application ("the space operator") has the highest precedence (ie 10), so $square\ 3 + 4 = (square\ 3) + 4$. Find out about the precedence levels of the various operators and *fully* parenthesize the expression below.

   $$f\ x \geqslant 0\ \&\&\ a\ ||\ g\ x\ y * 7 + 10 == b - 5$$

Exercise 1.5 (Programming). Define the string

$$thisOldMan :: String$$

that produces the following poem (if you type $putStr\ thisOldMan$).

This old man, he played one,
He played knick-knack on my thumb;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played two,
He played knick-knack on my shoe;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played three,
He played knick-knack on my knee;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played four,
He played knick-knack on my door;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played five,
He played knick-knack on my hive;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played six,
He played knick-knack on my sticks;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played seven,
He played knick-knack up in heaven;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played eight,
He played knick-knack on my gate;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played nine,
He played knick-knack on my spine;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played ten,

He played knick-knack once again;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

Try to make the program as short as possible by capturing recurring patterns. Define a suitable function for each of those patterns.

Exercise 1.6 (Programming, `Shapes.lhs`). The datatype *Shape* defined below captures simple geometric shapes: circles, squares, and rectangles.

```
data Shape
   = Circle Double                -- radius
   | Square Double                -- length
   | Rectangle Double Double      -- length and width
  deriving (Show)
```

Examples of concrete shapes include *Circle* (1 / 3), *Circle* 2.1, *Square pi*, and *Rectangle* 2.0 4.0.

The function *showShape* illustrates how to define a function that consumes a shape. A shape is one of three things. Correspondingly, *showShape* consists of three equation, one for each kind of shape.

```
showShape :: Shape → String
showShape (Circle r)      = "circle of radius " ++ show r
showShape (Square l)      = "square of length " ++ show l
showShape (Rectangle l w) = "rectangle of length " ++ show l
                              ++ " and width " ++ show w
```

Use the same definitional scheme to implement the functions

```
area        :: Shape → Double
perimeter   :: Shape → Double
center      :: Shape → (Double, Double)   -- x- and y-coordinates
boundingBox :: Shape → (Double, Double)   -- width and height
```

(The names are hopefully self-explanatory.)

Hints to practitioners 1. Functional programming folklore has it that a functional program is correct once it has passed the type-checker. Sadly, this is not quite true. Anyway, the general message is to exploit the compiler for *static* debugging: compile often, compile soon. (To trigger a re-compilation after an edit, simply type `:reload` or `:r` in GHCi.)

We can also instruct the compiler to perform additional sanity checks by passing the option `-Wall` to GHCi e.g. call `ghci -Wall` (turn all warnings on). The compiler then checks, for example, whether the variables introduced on the left-hand side of an equation are actually used on the right-hand side. Thus, the definition $k\ x\ y = x$ will provoke the warning “Defined but not used: y”. Variables with a leading underscore are not reported, so changing the definition to $k\ x\ \_y = x$ suppresses the warning.

2 Types and polymorphism

Exercise 2.1 (Warm-up: programming).

1. How many total functions are there that take one Boolean as an input and return one Boolean? Or put differently, how many functions are there of type $Bool \to Bool$? Define all of them. Think of sensible names.

2. How many total functions are there that take two Booleans as an input and return one Boolean? Or put differently, how many functions are there of type $(Bool, Bool) \to Bool$? Define at least four. Try to vary the definitional style by using different features of Haskell, e.g. predefined operators such as $||$ and $\&\&$, conditional expressions (**if** . . **then** . . **else** . .), guards, and pattern matching.

3. What about functions of type $Bool \to Bool \to Bool$?

Exercise 2.2 (Programming, `Char.lhs`). Haskell's *String*s are really lists of characters i.e. **type** $String = [Char]$. Thus, quite conveniently, all of the list operations are applicable to strings, as well: for example, *map toLower* `"Ralf"` $\Longrightarrow$ `"ralf"`. (Recall that *map* takes a function and a list and applies the function to each element of the list.)

1. Define an equality test for strings that, unlike $==$, disregards case, e.g. `"Ralf"` $==$ `"raLF"` $\Longrightarrow$ *False* but *equal* `"Ralf"` `"raLF"` $\Longrightarrow$ *True*.

2. Define predicates

   $$\begin{aligned} &isNumeral :: String \to Bool \\ &isBlank \quad\;\; :: String \to Bool \end{aligned}$$

   that test whether a string consists solely of digits or white space. You may find the predefined function $and :: [Bool] \to Bool$ useful which conjoins a list of Booleans e.g. $and\ [1 > 2, 2 < 3] \Longrightarrow False$ and $and\ [1 < 2, 2 < 3] \Longrightarrow True$. You also may want to import *Data.Char*, see Appendix A for details.

3. Define functions

   $$\begin{aligned} &fromDigit :: Char \to Int \\ &toDigit \quad\; :: Int \to Char \end{aligned}$$

   that convert a digit into an integer and vice versa, e.g. *fromDigit* `'7'` $\Longrightarrow$ 7 and *toDigit* 8 $\Longrightarrow$ `'8'`.

4. Implement the Caesar cipher $\mathit{shift} :: \mathit{Int} \to \mathit{Char} \to \mathit{Char}$ e.g. $\mathit{shift}\ 3$ maps `'A'` to `'D'`, `'B'` to `'E'`, `...`, `'Y'` to `'B'`, and `'Z'` to `'C'`. Try to decode the following message (*map* is your friend).

```
msg = "MHILY LZA ZBHL XBPZXBL MVYABUHL HWWPBZ JSHBKPBZ "
    ++ "JHLJBZ KPJABT HYJUBT LZA ULBAYVU"
```

Exercise 2.3 (Programming). Explore the difference between machine-integers of type *Int* and mathematical integers of type *Integer*. Fire up GHCi and type:

```
⟩⟩⟩⟩  product [1 .. 10] :: Int
⟩⟩⟩⟩  product [1 .. 20] :: Int
⟩⟩⟩⟩  product [1 .. 21] :: Int
⟩⟩⟩⟩  product [1 .. 65] :: Int
⟩⟩⟩⟩  product [1 .. 66] :: Int
```

The expression $\mathit{product}\ [1 \mathinner{..} n]$ calculates the product of the numbers from 1 up to n, aka the factorial of n. The type annotation $::\mathit{Int}$ instructs the compiler to perform the multiplications using machine-integers. Repeat the exercise using the type annotation $::\mathit{Integer}$. What do you observe? Can you explain the differences? On my machine the expression $\mathit{product}\ [1 \mathinner{..} 66] :: \mathit{Int}$ yields 0. Why? (Something to keep in mind. Especially, if you plan to work in finance!)

Exercise 2.4 (Programming).

1. Define a function

   $$\mathit{swap} :: (\mathit{Int}, \mathit{Int}) \to (\mathit{Int}, \mathit{Int})$$

   that swaps the two components of a pair. Define two other functions of this type (be inventive).

2. What happens if we change the type to

   $$\mathit{swap} :: (a, b) \to (b, a)$$

   Is your original definition of *swap* still valid? What about the other two functions that you have implemented?

3. What's the difference between the type $(\mathit{Int}, (\mathit{Char}, \mathit{Bool}))$ and the type $(\mathit{Int}, \mathit{Char}, \mathit{Bool})$? Can you define a function that converts one "data format" into the other?

Exercise 2.5 (Warm-up: static typing).

1. Which of the following expressions are well-formed and well-typed? Assume that the identifier b has type $Bool$.

   $(+4)$
   div
   $div\ 7$
   $(div\ 7)\ 4$
   $div\ (7\ 4)$
   $7\ `div`\ 4$
   $+\ 3\ 7$
   $(+)\ 3\ 7$
   $(b, \texttt{'b'}, \texttt{"b"})$
   $(abs, \texttt{'abs'}, \texttt{"abs"})$
   $abs \circ negate$
   $(*3) \circ (+3)$

   If you get stuck, try to evaluate the expressions and/or see Hint 2. (As an aside, if you prefer ASCII over Unicode: in ASCII function composition is simply a full stop i.e. "**.**").

2. What about these?

   $(abs\circ) \circ (\circ negate)$
   $(div\circ) \circ (\circ mod)$

   (They are more tricky—don't spend too much time on this.)

3. Try to infer the types of the following definitions.

   $i\ x = x$
   $k\ (x, y) = x$
   $b\ (x, y, z) = (x\ z)\ y$
   $c\ (x, y, z) = x\ (y\ z)$
   $s\ (x, y, z) = (x\ z)\ (y\ z)$

   If you get stuck see Hint 2. Are any of these functions predefined (perhaps under a different name)? Again, see Hint 2.

Exercise 2.6 (Worked example: polymorphism). The purpose of this exercise is to explore the concept of *parametric polymorphism*. (The findings are not specific to Haskell or functional programming. Many statically typed object-oriented languages feature parametric polymorphism under the name of *generics*.)

1. Define total functions of the following types:

   (a) $Int \to Int$

   (b) $a \to a$

   (c) $(Int, Int) \to Int$

   (d) $(a, a) \to a$

   (e) $(a, b) \to a$

   How many total functions are there of type $Int \to Int$? By contrast, how many total functions are there of type $a \to a$?

2. Define total functions of the following types:

   (a) $(a, a) \to (a, a)$

   (b) $(a, b) \to (b, a)$

   (c) $(a \to b) \to a \to b$

   (d) $(a, x) \to a$

   (e) $(x \to a \to b, a, x) \to b$

   (f) $(a \to b, x \to a, x) \to b$

   (g) $(x \to a \to b, x \to a, x) \to b$

   Have you worked on a similar exercise before? Perhaps in a different context? *Hint:* read "$\to$" as logical implication and "," as logical conjunction.

3. Define total functions of the following types:

   (a) $Int \to (Int \to Int)$

   (b) $(Int \to Int) \to Int$

   (c) $a \to (a \to a)$

   (d) $(a \to a) \to a$

   How many total functions are there of type $(Int \to Int) \to Int$? By contrast, how many total functions are there of type $(a \to a) \to a$?

Hints to practitioners 2. GHCi features a number of commands that are useful during program development: e.g. `:type` ⟨expr⟩ or just `:t` ⟨expr⟩

shows the type of an expression; `:info` ⟨name⟩ or just `:i` ⟨name⟩ displays information about the given name e.g.

```
⟩⟩⟩⟩ :info map
map :: (a → b) → [a] → [b]   — Defined in ‘GHC.Base’
⟩⟩⟩⟩ :type map ∘ map
map ∘ map :: (a → b) → [[a]] → [[b]]
```

This is particularly useful if your program does not typecheck. (Or, if you are too lazy to type in signatures.)

More detailed information about the standard libraries is available online: `https://www.haskell.org/hoogle/`. Hoogle is quite nifty: it not only allows you to search the standard libraries by function name, but also by type! For example, if you enter `[a] -> [a]` into the search field, Hoogle will display all list transformers.

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:

```
maybe     :: b → (a → b) → Maybe a → b
catMaybes :: [Maybe a] → [a]
```

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:

```
empty  :: Map k v
insert :: (Ord k) ⇒ k → v → Map k v → Map k v
update :: (Ord k) ⇒ k → Map k v → Maybe v
```

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.