Recommended exercises with hints to Chapters 12-13

– D7012E Declarative languages –

Håkan Jonsson
Luleå University of Technology, Sweden

April 7, 2019

1 How to work

- Sit by a computer with your book, pen, and paper.
- Work on the exercises below one at a time and in order.

2 Chapter 12

Exercise 12.2 on page 214

You could write `numEqual` in a number of different ways. I did it in one way at an earlier lecture... The type is affected by the fact `x` and the elements of the list can be compared with `==`.

By the way, the last question is pretty simpel since being a member essentially means "occurs at least once".

Exercise 12.3 on page 214

First, implement the two functions separately.

Second, they are very similar, so (if we are not too concerned with efficiency) we could also have one of them just swap the pairs and then call the other. Try to re-write the other in this way.

Exercise 12.4 on page 219

Skip `Bool` because an instance for that type already exists (see page 216). Instead write an instance declaration for `Int` where the size is twice the value of the integer and `toString` returns the string `"Not yet available"`.

To write instances for `(a,b)` and `(a,b,c)` go back and compare with how an instance for pairs are written in `Eq`.

Exercise 12.5 on page 219

Write a plain recursive function that "shaves of" one digit at a time using the operators `'div'` and `%`. Then change the instance declaration concerning `Int` in you solution to Excercise 12.4 so `toString` returns what you new function returns.

Exercise 12.6 on page 219

`size` is the function in `Visible`.

Exercise 12.8 on page 219

Find out how this was done a couple of pages earlier (in `Eq` and `Visible`). Look up what the complete declaration of `Ord` contains.

You need a recursive function that simultaneously traverses two lists and compares their elements, to compare lists lexicographically.

Exercise 12.10 on page 225

Skip `showBoolFun` and go directly for `showBoolFunGen` with type `(a -> String) -> (Bool -> a) -> String`. Note that `showBoolFunGen` "takes two arguments"[1]: 1) A function that, simular to the method `toString` in Java, makes a `String` out of a value of type `a` and 2) a Boolean function. As for 1), you could use `show` from class `Show`. This, however, effects the type (how?).

(The last part of the exercise is unclear. If "multiple argument Boolean functions" means functions with types `Bool -> Bool -> ...  -> Bool -> a`, I see no way to write a general solution. It is not possible to do recursion over the number of arguments and there is no pattern matching that works that way. If you find a way to solve this last part, let me know!)

3 Chapter 13

Exercise 13.1 on page 230

Investigate all four functions like we did in class. Note that a numerical constant *without* decimals is of type `Num t => t` while a constant *with* decimals is of type `Fractional t`

[1]No, not really. Like all functions it takes just one argument.

`=> t`. `Fractional` extends `Num`, and therefore all values of type `Fractional` are also of type `Num` (and can be used as such). See page 224. Example:

```
*Main> :type 3.1
3 :: Num p => p
*Main> :type 3.1
3.1 :: Fractional p => p
*Main>
```

However, note that the type system in Haskell is very strict. The function

```
f :: (Fractional t, Num u) => t -> u
f x = x
```

could look alright (alright?). Since every `Fractional` is also a `Num`, it should be safe to return `x`, a `Fractional`. But no. This is rejected by Haskell, and on good grounds, because see what happens if we instead start to investigate the type of the result `x`. It has type `u`, which is a `Num`. A `Num` is not a `Fractional`, so the type `t` of the parameter (`x`) does not fit the type `u` of the result (`x`). Types must work both (all) ways, and not just one. Indeed very strict.

Exercise 13.2 on page 237

Note that when we unify, we hunt for the most general type that fits both types being unified while not being more general than the context requires.

As an example, assume we have the type `[(a,b,Int)]` and try to unify with the following.

- `c`. This is too general to say anything vital to us. The unified type becomes just `[(a,b,Int)]`. Really, type `c` stands for any kind of value inclusing, for instance, pairs. And pairs are not lists. So what restricts the unified type is `[(a,b,Int)]`. Not that this type is the same as any type `[(x,y,Int)]`, `[(t,u,Int)]`, `[(r,s,Int)]`, ...as long as the type variables (x,y,t,u,`r`, and `s`) are unbound.
- `Int`. This unification does not work because an `Int` is not a list. `Int` is far too restricted.
- `[c]`. This is slightly too general compared with `[(a,b,Int)]` but works fine. The unified type is `[(a,b,Int)]`.
- `[(g,Float,h)]`. This gets interesting! Both types are lists with triplets. However, one restricts the second component of the pairs to `Float` while the other restricts the third component to be an `Int`. So, the unified typ is `[(w,Float,Int)]`.
- `(Num t, Num u) => [(Char,t,u)`. Even more interesting! Clearly, the first component must be a `Char` and the second a `Num`, because `Num` is more restricted than `b` ("anything"). For the third component we unify `Int` and `Num u => u`, and find that the most general (not too general) type is `Int` since `Num` includes `Int` (`Int` is a special case of `Num`, so to speak). So, the final unified type is `Num t => [(Char,t,Int)`.

Exercise 13.3 on page 237

Note that the use of the word "unify" here is a bit different than in the chapter: Find values of `a`, `b`, and `c` so that both types become `(Bool,[Bool])` while `a == b` and `[a] == c`.

Exercise 13.4 on page 237

Start with the unification of each of the three arguments and `(a,[a])`.

The type of the result is not related to `a` so it does not depend on the argument. Not a very good question, in my opinion. Answer instead 13.5 below.

Exercise 13.5 on page 237

Just derive the types of the results in the cases where the arguments are those given in Exercise 13.4. Here, the relation between argument and result is clear.

Exercise 13.6 on page 237-238

Note that the type of an empty list is "whatever fits".

Try this in your Haskell system! I wrote

```
f :: [a]->[b]->a->b
f (x:_) (y:_) z = y
```

and then defined `h`. But before I checked types in the Haskell system I found them doing the ordinary detective work needed to unify/find out types.

Exercise (13.7*) on page 238

This is a very hard exercise. You might like to check out how the pre-defined functions `asTypeOf` and `const` are programmed, but do not spend too much time on it.

Exercise 13.8 on page 238

Good exercise! Use the types

```
id :: i -> i
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (d -> e -> f) -> ((d, e) -> f)
```

and note that the two last ones can also be written

```
curry :: ((a, b) -> c) -> (a -> (b -> c))
uncurry :: (d -> e -> f) -> (d, e) -> f
```

This makes it easier to work out the unification. And do like I have done when I wrote these types: Use fresh, unbound, type variables(!)

I give you the first one free as a warm-up: `curry id` means we need to unify `i -> i` with `(a, b) -> c`, which means that `c` must be `(a,b)`. Substituting `c` for `(a,b)` in `a -> b -> c`, the type of the result when `curry` gets its argument, gives us the type `a -> b -> (a,b)`. So `curry id :: a -> b -> (a,b)`.

Now do the other ones (and in some you use previous results).

Exercise 13.10 on page 240

Investigate how variables and operators are used in `merge`. Note that `<` is a better clue than `==`.

Exercise 13.12 on page 240

Look up the types of the operators and you get all the answers.