F28PL - Programming Languages

Question E (Higher-Order Functions)

Tommy Lamb

H00217505

The plain text file which accompanies this PDF contains a listing of all functions, with some formatting to aid readability. It can be opened and executed in Poly/ML from the Terminal by navigating to the save location of the file, opening Poly/ML, and using the command : ' use "F28PL - Tommy Lamb - Question E - Code.txt"; '. This method will cause Poly/ML to ignore the document formatting and only display system output (not input). To maintain formatting, all text can be copied and pasted into a running instance of Poly/ML. For best results it should be copied/pasted in small blocks, around 1 question in size.

1. Write a function of type `bool -> bool -> bool`.

   ```
   fun Q1 x y = x andalso y;
   ```

   Given the use of logical operator `andalso`, x and y must both have type `bool` in order to carry out the operation.

2. How many different (in the sense of calculating different truth-tables) functions are there in `bool -> bool -> bool`?

   Eight. Interpreting this as a function which takes in two Boolean values and outputs a third Boolean value, there are four possible inputs:

   - True, True
   - True, False
   - False, True; and
   - False, False

   For each of these inputs the function can output either True or False, so $4 \times 2 = 8$. This can be represented more abstractly as 2^n, where $n = Number\ of\ input\ and\ output\ variables$ - in this case 2^3. This is because for any Boolean variable x, there are two possible values; this number of possible values can be represented as 2_x. Given two Boolean variables x_1 and x_2, for any value of x_1 there exists two possible values for x_2:

   $$2_{x1} \times 2_{x2} = 4_{x1\,\&\,x2}$$

   Thus for any amount of Boolean variables, n, the total number of possible value combinations can be expressed as:

   $$2_{x1} \times 2_{x2} \times 2_{x3} \times \ldots\ 2_{xn} = 2^n$$

3. Explain the similarities and differences between `fn (x:int) => x` and `fn (x:real) => x`.

   The main similarity between the two functions is that both are of the form `x => x`, that is they return the same value they are given. However both functions have different types: the first `int -> int`, and the second `real -> real`. This difference in type is the result of the explicit type casting on the functions' arguments. In each the variable x is typed as being either `int` or `real`, and as `x` is the return value, the declared type extends to the return type.

4. Write two functions of type ('a -> bool) -> 'a list -> bool.
   - The first should return true when the first argument is true of all elements in the second argument;
   - the second should return true when the first argument is true of some element in the second argument.

```
fun Q4A f [] = true
  | Q4A f (head::tail) = (f head) andalso (Q4A f tail);
```

   Applies predicate f to the first element of the list, and recursively checks the remainder of the list. The result of applying f to each element in turn is combined using logical AND, ensuring f returns true for each and every element in the list. The uncomfortable assumption that the predicate is true for the empty list is required to act as a base case in the recursion.

```
fun Q4B f [] = false
  | Q4B f (head::tail) = (f head) orelse (Q4B f tail);
```

   This function works as that before it, however it makes use of logical OR, such that the function returns true if predicate f is true of any element within the list. Again an uncomfortable assumption is used to provide a base case.

5. Write a function of type `'a -> unit`. Is it the only one?

```
fun Q5 x = ();
```

   Yes. Given the special nature of `()`, the only way to return it is directly. No operations can be done (as far as I'm aware) with `()`, and it cannot be the result of any primitive operation carried out on another type. It is akin to "null" in Java in that it can only be returned by a function or method explicitly.

6. Write a function of type `'a -> ('a*'a)`. Is it the only one?

```
fun Q6 x = (x,x);
```

   Yes. Having the lone input of type `'a` means that the only way to return something of type `'a` is to return the input variable itself. Performing any operations on the input variable would result in a return type of `'b`. The tuple format is just dressing up the returned value and obfuscating the answer.

7. Write a function of type `'a -> 'b -> ('a*'b).`

   ```
   fun Q7 x y = (x,y);
   ```

   Simply takes in two variables, and output them together in a tuple.

8. How many different functions are there in `'a -> 'b -> ('a*'b)`?

   One. The only way to return a tuple of type `'a*'b` given two variables of type `'a` and `'b` is to return those variables. Otherwise the return type would likely be `('c,'d)` if any operations or functions were carried out.

9. Write a function of type `'a -> 'a -> ('a*'a).`

   ```
   fun Q9 (x:'a) (y:'a) = (x,y);
   ```

   The only way to have two input variables of the same abstract type `'a` is to force the issue with the compiler. Without the type declarations the compiler cannot infer that both variables are of the same type, thus it would provide a function of type `'a -> 'b -> 'a * 'b`. The explicit type declarations force to compiler to treat both variables as being of the same type.

10. How many different functions are there in `'a -> 'a -> ('a*'a)`?

    Four. Given the variables x and y of type `'a`, then the function may return a tuple of:
    - `(x, x)`
    - `(x, y)`
    - `(y, x)`
    - `(y, y)`

    All of which have the type `('a * 'a)`.

11. Write a function of type `(('a -> 'b)*'a) -> 'b.`

    ```
    fun Q11 (f,x) = f x;
    ```

    The function takes in a tuple comprising a function of type `('a -> 'b)` and a variable of type `'a`. The compiler infers the type of the lone variable x to be `'a` as it must have the same type as the argument passed to function f as that is how it is used. It cannot be inferred however what type the function returns, and so the return type is set to `'b`.

12. How many different functions are there in `(('a -> 'b)*'a) -> 'b`?

    One. Since the only non-function variable available is of type `'a`, the only way to return something of type `'b` is to use apply function f to variable x.

13. <u>Write a function of type `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c).`</u>

```
fun Q13 f1 f2 = f2 o f1;
```

The use of function composition (piping) is required to force the compiler to treat both `f1` and `f2` as functions; simply saying `f2(f1)` would cause the compiler to treat f1 as a non-function variable of type `'a` to which `f2` would be applied. Since a function can return a something of a different type than that given as an argument, a function automatically has type `'a -> 'b` by default; and as the result of function f1 is fed as argument to function f2, the f1 result and `f2` argument must have the same type: `'b`. Because `f1` has been assumed to be `'a -> 'b` and the argument of `f2` thus type `'b`, the return type of `f2` cannot be assumed to be either `'a` or `'b`, and so it is designated `'c`. The result of function composition is simply another function which takes an argument equal to that of the 'first' function in the composition, and returns that of the 'last'. The order of evaluation for such piping can be thought of as right to left, with `f1` being the first function to be evaluated and `f2` the last. Thus the resulting function of the function composition has the argument type of `f1`: 'a and return type of `f2`: `'c`. The function also makes use of partial application for brevity and simplicity.

This could also be written less succinctly as:

```
fun Q13 f1 f2 x = f2(f1 x);
```

or

```
fun Q13 f1 f2 x = (f2 o f1) x;
```

14. <u>How many different functions are there in `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)`?</u>

Taking either form of the function with or without x there is only one way of producing the output `'a -> 'c`. The only variation is in explicitly declaring the existence of variable x rather than implying it, and the different semantic methods of taking the result of one function as the argument to another.

15. <u>Write a function of type `('a -> ('b -> 'c)) -> ('a -> 'b) -> ('a -> 'c).`</u>

```
fun Q15 f1 f2 x = f1 x (f2 x);
```

Removing the extra brackets here that are not displayed by the Poly/ML compiler makes this somewhat easier to understand. Using implicit bracketing where possible, this becomes:

```
('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Which can be intuitively interpreted as a function which take three arguments with one return value:

- A function `f1` which takes two arguments, and returns one value:
    - ◊ Take variable of type `'a`
    - ◊ Take variable of type `'b`
    - ◊ Return variable of type `'c`
    - ◊ Type: `('a -> 'b -> 'c)`
- A second function `f2` which takes one argument and returns one value:
    - ◊ Take variable of type `'a`
    - ◊ Return variable of type `'b`
    - ◊ Type: `('a -> 'b)`
- A non-function variable x of type `'a`
- Returning a variable of type `'c`

Working from this, it can be concluded that in order to return a value of type `'c`, the function must return the final result of executing function `f1`. In order to execute `f1` it must take arguments of type `'a` and `'b`, though only a variable of type `'a` is available. In order to take an argument of type `'b`, function `f2` must first be run on the variable of type `'a`, as it returns a value of type `'b`.

So function `f2` is first run on the `'a` type variable:

```
f2 x
```

Which returns a value of type `'b`. This is then used as an argument for `f1` alongside the `'a` type variable:

```
f1 x (f2 x)
```

Which gives the final return type of the overall function:

```
'a -> 'c
```

The rest of the function type is declaring exactly what variables (including functions) it takes as arguments, as interpreted above. Combing all of these gives the result:

```
('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Which with the implicit brackets added becomes:

```
('a -> ('b -> 'c)) -> ('a -> 'b) -> ('a -> 'c)
```

16. <u>(Hard) Using exceptions or otherwise, write a function of type `(unit -> 'a).`</u>

```
fun Q16 () = OS.Process.terminate OS.Process.success;
```

This answer is neither obvious nor in the spirit of the questions, though technically correct. It was after much traipsing through the SML Basis Library that this solution was found. `OS.Process.success` is an inbuilt value of type `status`, and `OS.Process.terminate` is a function of type `status -> 'a` which has the effect of terminating the running instance of Poly/ML with some status value. This function executes the `terminate` function in order to achieve a return type of `'a`, whilst taking `()` as argument. Running this function within the terminal will cause the running instance of Poly/ML to close and force a return to the Bash command line interface, however the type of Q16 should be printed out first before closing.

Later it was discovered that the act of raising an exception was considered by Poly to be of type `'a`, such that the following functions also have type `unit -> 'a`:

```
exception Hell;
fun Q16 () = raise Hell;
```

or more succinctly:

```
fun Q16 () = raise Match;
```