Functional Programming - Mandatory 1

Daniel Kuzin
s184225

Mads Kahler
s184206

September 30, 2019

1

1.1

```
    let rec collect f = function
        | [] -> []
        | x::xs -> f x @ collect f xs;;
        val ('a -> 'b list) -> 'a list -> 'b list
```

If we decompose the function into its asserted types one step at a time, you could conclude the following:

Let's say that f, the argument for the function, takes an $'a$. Since the result of fx is to be concatinated with the infix operator '@', we know that the result must be a list. We don't know anything specific about the type, so lets call it a $'b$ list. This means we can write the following:

```
   let rec collect f = function
        | [] -> []
        | x::xs -> 'b list @ collect f xs;;
```

If you call `collect` on f, we get a function with the signature: ($'a$ list `->` $'b$ list). This must be the case because we call the resulting function with the argument xs. We know that xs is of type a' list because xs is the tail of the list $x :: xs$, and since we've called the type of x the name $'a$, and we know that the list xs has the same element type as x, xs must be an $'a$ list. Additionally, the result must be concatinated with the infix operator '@' with a $'b$ list, so the return type must be a $'b$ list. Therefore we can write:

```
   let rec collect f = function
        | [] -> []
        | x::xs -> 'b list @ ('a list -> 'b list) xs;;
```

From here, we call the function we got from `collect f` on xs, where xs is an input of type $'a$ list, and out comes a $'b$ list.

```
   let rec collect f = function
        | [] -> []
        | x::xs -> 'b list @ 'b list;;
```

Finally, we know that concatenating two lists with the infix operator '@' (concatenation), will get us one list of the same type, therefore we can write:

```
   let rec collect f = function
        | [] -> []
        | x::xs -> 'b list;;
```

Since we didn't assert any specific types, and only conformed to those required by the function body, we can be sure that we have found the most general type for the function.

1.2

We start by partially evaluating `collect` with our function from the argument. The resulting function would look like this:

```
let rec collectP xs = function
    | [] -> []
    | (a,b)::xtail -> [a..b] @ collectP xtail;;
```

Now we can much more easily evaluate the function: We start by calling the function with the argument [(1,3);(4,7);(8,8)]:

```
let rec collectP [(1,3);(4,7);(8,8)] = function
    | [] -> []
    | (1,3)::[(4,7);(8,8)] -> [1..3] @ collectP [(4,7);(8,8)];;
```

From here we evaluate the recursive calls:

```
let rec collectP [(4,7);(8,8)] = function
    | [] -> []
    | (4,7)::[(8,8)] -> [4..7] @ collectP [(8,8)];;

let rec collectP [(8,8)] = function
    | [] -> []
    | (8,8)::[] -> [8..8] @ collectP [];;

let rec collectP [] = function
    | [] -> []
    | ...
```

Now we can follow the results of the chain of calls back to the root call. The combined operations will look like this:

$$[1..3] \text{ @ } ([4..7] \text{ @ } ([8..8] \text{ @ } [\,])) = [1;2;3] \text{ @ } ([4;5;6;7] \text{ @ } ([8] \text{ @ } [\,]))$$

The final result is:

$$[1;2;3;4;5;6;7;8]$$

1.3

Our original thought was to assert the types in lambda function in the argument for collect to start with, but this turned out to be pretty complicated, and confusing, as the type of the function should be determined by the acceptable types for the range `(..)` operator[1].

[1]Interestingly, the F# documentation mentions that the required type for the operator is: `'t (requires 't with static member (+) and 't with static member One)`. While both integers and chars fit this description, it seems that the language can't assert the most general type in this case, and just chooses the type of the first usage, or `int` otherwise. We couldn't figure out why this is.

Starting with the argument for the function we partially evaluated as `collectP`, it is easy to see that the range `(..)` operator will be applied on integers, making the type of the lambda function `(int*int) -> int list`.

This means that the type of `collectP` is `(int*int) list -> int list`, meaning that the whole type would be: `((int*int) -> int list) -> (int*int) list -> int list`.

To make sure this is an instance of the asserted type for `collect`, we could figure out what the first occurrences of $'a$ and $'b$ correspond to here, and see if the whole type is fulfilled. Here, $'a$ is `(int * int)`, and $'b$ is `int`. Filling out the rest of the general type, we get: `((int*int) -> int list) -> (int*int) list -> int list`, which is the same as we concluded before.