L28: Advanced functional programming

Exercise 1

Due on Febuary 9th

For these questions, you may assume that all the definitions given in Figure 1 are available in System Fω.

1. Give a typing derivation for $\Lambda\alpha :: *.\lambda x : \alpha.\langle x,\, x\rangle$ *(3 marks)*

2. Algorithm J is defined recursively over the structure of terms. The case for function application (M N) is as follows:

   ```
   J (Γ, M N) = β
           where A = J (Γ, M)
           and B = J (Γ, N)
           and unify' ({A = B → β}) succeeds
           and β is fresh
   ```

   Give similar cases to handle the following constructs:

   (i) Constructing a pair (⟨M, N⟩)

   (ii) Projecting the first element of a pair (**fst** M)

   (iii) Constructing a sum using `inl` (**inl** M)

   (iv) Destructing a sum (**case** L **of** x.M | y.N)

   (5 marks)

3. The following OCaml type represents ternary trees:

   ```
   type 'a ttree =
     | Empty : 'a ttree
     | Tree : 'a * 'a ttree * 'a ttree * 'a ttree -> 'a ttree
   ```

   (i) Give an encoding of this type in System Fω consisting of a type operator (`TTree`) of kind $*\Rightarrow*$, a function for each constructor (`empty` and `tree`), and a function (`foldTTree`) for folding over trees.

   (3 marks)

   (ii) Write a function

   ```
   totalNatTree : TTree Nat → Nat
   ```

   that computes the total of a ternary tree of Nats in System Fω.

   (1 marks)

4. Using existentials, products, the List type, the Nat type and the Option type, implement a queue data structure in System Fω with a type corresponding to the following OCaml signature:

```
type 'a t
val empty : 'a t
val enqueue : 'a -> 'a t -> 'a t
val dequeue : 'a t -> 'a t * 'a option
val size : 'a t -> int
```

(6 marks)

5. (i) Why does the type checker reject the following program? Specifically, what potentially unsafe behaviour is the type checker's rejection of the program intended to prevent?

```
let rec make size elem =
  if size <= 0 then []
  else elem :: (make (size - 1) elem)

let foo =
  let make_three = make 3 in
    make_three "hello",
    make_three 5
```

(3 marks)

(ii) Replace `make` with a function of the same type which would make the program unsafe if the type checker did not reject it.

(1 mark)

(iii) Adjust the definition of `make_three` so that the program is accepted by the type-checker, without changing the result of the program.

(1 mark)

(iv) The following program is also rejected by the type-checker:

```
let make size =
  Printf.printf "Making lists of length %d\n" size;
  let rec loop size elem =
    if size <= 0 then []
    else elem :: (loop (size - 1) elem)
  in
    loop size

let foo =
  let make_three = make 3 in
    make_three "hello",
    make_three 5
```

It is not possible to make this program pass the type-checker by only changing `make_three` without affecting the program's observable behaviour (i.e. the number of times the message is printed).

However, using a record field with a universal type, it is possible to wrap the result of `make` so that it can be used polymorphically.

Using this technique, adjust the program so that it passes the type-checker without affecting its observable behaviour.

(2 marks)

```
Nat = ∀α::*.α → (α → α) → α

zero = Λα::*.λz:α.λs:α → α.z

succ = λn:Nat.Λα::*.λz:α.λs:α → α.s (n [α] z s)

add = λm:Nat.λn:Nat.m [Nat] n succ

List = λα::*.∀φ::*⇒*.φ α → (α → φ α → φ α) → φ α

nil = Λα::*.Λφ::*⇒*.λnil:φ α.λcons:α → φ α → φ α.nil

cons = Λα::*.λx:α.λxs:List α.
         Λφ::*⇒*.λnil:φ α.λcons:α → φ α → φ α.
             cons x (xs [φ] nil cons)

foldList = Λα::*.Λβ::*.
        λc:α → β → β.λn:β.λl:List α.
           l [λγ::*.β] n c

append = Λα::*.
          λl:List α.λr:List α.
            foldList [α] [List α] (cons [α]) l r

reverse = Λα::*.
          λl:List α.
            foldList [α] [List α →List α]
              (λx:α. λk:List α →List α. λxs: List α. k (cons [α] x xs))
              (λxs: List α. xs) l (nil [α])

Option = λα::*.∀φ::*⇒*.φ α → (α → φ α) → φ α

none = Λα::*.Λφ::*⇒*.λnone:φ α.λsome:α → φ α.none

some = Λα::*.λx:α.
         Λφ::*⇒*.λnone:φ α.λsome:α →φ α.
             some x

foldOption = Λα::*.Λβ::*.
        λs:α → β.λn:β.λo:Option α.
           o [λγ::*.β] n s
```

Figure 1: Definitions in System Fω