L28: Advanced functional programming

Exercise 1

Due on 8th February 2016

Submission instructions

Your solutions for this exericse should be handed in to the Graduate Education Office by 4pm on the due date. Additionally, for questions 2 and 3, please email the completed text file `exercise1.f` to `jeremy.yallop@cl.cam.ac.uk`.

Preliminaries

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

$$
\begin{aligned}
\mathrm{Nat} &:: * \\
&= \forall\alpha :: * .\, \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \\[1em]
\mathrm{zero} &: \mathrm{Nat} \\
&= \Lambda\alpha :: * .\, \lambda z : \alpha .\, \lambda s : \alpha \rightarrow \alpha .\, z \\[1em]
\mathrm{succ} &: \mathrm{Nat} \rightarrow \mathrm{Nat} \\
&= \lambda n : \mathrm{Nat} .\, \Lambda\alpha :: * .\, \lambda z : \alpha .\, \lambda s : \alpha \rightarrow \alpha .\, s\ (n\ [\alpha]\ z\ s) \\[1em]
\mathrm{add} &: \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat} \\
&= \lambda m : \mathrm{Nat} .\, \lambda n : \mathrm{Nat} .\, m\ [\mathrm{Nat}]\ n\ \mathrm{succ} \\[1em]
\mathrm{Eq} &:: * \Rightarrow * \Rightarrow * \\
&= \lambda\alpha :: * .\, \lambda\beta :: * .\, \forall\phi :: * \Rightarrow * .\, \phi\, \alpha \rightarrow \phi\, \beta \\[1em]
\mathrm{refl} &: \forall\alpha :: * .\, \mathrm{Eq}\, \alpha\, \alpha \\
&= \Lambda\alpha :: * .\, \Lambda\phi :: * \Rightarrow * .\, \lambda x : \phi\, \alpha .\, x \\[1em]
\mathrm{symm} &: \forall\alpha :: * .\, \forall\beta :: * .\, \mathrm{Eq}\, \alpha\, \beta \rightarrow \mathrm{Eq}\, \beta\, \alpha \\
&= \Lambda\alpha :: * .\, \Lambda\beta :: * .\, \lambda e : (\forall\phi :: * \Rightarrow * .\phi\, \alpha \rightarrow \phi\, \beta) .\, e\ [\lambda\gamma :: * .\mathrm{Eq}\, \gamma\, \alpha]\ (\mathrm{refl}\ [\alpha]) \\[1em]
\mathrm{trans} &: \forall\alpha :: * .\, \forall\beta :: * .\, \forall\gamma :: * .\, \mathrm{Eq}\, \alpha\, \beta \rightarrow \mathrm{Eq}\, \beta\, \gamma \rightarrow \mathrm{Eq}\, \alpha\, \gamma \\
&= \Lambda\alpha :: * .\, \Lambda\beta :: * .\, \Lambda\gamma :: * .\, \lambda ab : \mathrm{Eq}\, \alpha\, \beta .\, \lambda bc : \mathrm{Eq}\, \beta\, \gamma .\, bc\ [\mathrm{Eq}\, \alpha]\ ab
\end{aligned}
$$

Figure 1: Definitions in System Fω

1 Types and type inference

(a) For each of the following Fω terms either give a typing derivation or explain why the term has no typing derivation:

(i) λf:$\forall\alpha$::*.α $\rightarrow$ α.f f

(ii) $\Lambda\beta$::*.($\Lambda\phi$::* $\Rightarrow$ *.λf:($\forall\alpha$.ϕ α).f [β]) [β]

(iii) $\Lambda\alpha$::*.λf:($\forall\phi$::* $\Rightarrow$ *.ϕ α).λx:α.f [$\lambda\beta$::*.α $\rightarrow$ α] x

(8 marks)

(b) 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 value of sum type using `inr`: `inr M`

(ii) Scrutinising a value of sum type: `case L of x.M | y.N`

(4 marks)

(c) Why does OCaml's type checker reject the following program?

```
let f = fun x -> x in
let g = f f in
   g g
```

(4 marks)

2 Encoding data types in Fω

The following OCaml type represents non-empty trees.

```
type 'a tree =
  Leaf : 'a -> 'a tree
| Branch : 'a tree * 'a tree -> 'a tree
```

(a) Write an Fω encoding of the `tree` datatype. Your encoding should include a type operator of the following kind:

$\texttt{Tree} :: * \Rightarrow *$

and functions of the following types

$\texttt{leaf} :: \forall\alpha :: *.\alpha \rightarrow \texttt{Tree}\,\alpha$
$\texttt{branch} :: \forall\alpha :: *.\texttt{Tree}\,\alpha \rightarrow \texttt{Tree}\,\alpha \rightarrow \texttt{Tree}\,\alpha$

(b) Write an Fω function that computes the sum of the elements in a `tree` of Nats in Fω. Your function should have the following type:

$\texttt{totalNatTree} : \texttt{Tree Nat} \rightarrow \texttt{Nat}$

(6 marks)

3 Type equality in Fω

The lecture notes introduce the following definition of type equality, based on Leibniz's principle:

$$\texttt{Eq} = \lambda\alpha{::}{*}.\lambda\beta{::}{*}.\forall\phi{::}{*}\Rightarrow{*}.\phi\,\alpha \to \phi\,\beta$$

Here is a second definition of equality, based on the encoding of a data type:

$$\texttt{Equal} = \lambda\alpha{::}{*}.\lambda\beta{::}{*}.\forall\phi{::}{*}\Rightarrow{*}\Rightarrow{*}.(\forall\gamma{::}{*}.\phi\;\gamma\;\gamma) \to \phi\,\alpha\,\beta$$

(Here is the OCaml data type corresponding to `Equal`, which we will consider in more detail in lectures 8 and 9:)

```
type ('a, 'b) eql = Refl : ('x, 'x) eql
```

(*a*) Show that `Equal` represents an equivalence relation by defining values that encode reflexivity, symmetry and transitivity properties:

$\texttt{reflEqual} : \forall\alpha.\texttt{Equal}\;\alpha\;\alpha$

$\texttt{symmEqual} : \forall\alpha.\forall\beta.\texttt{Equal}\;\alpha\;\beta \to \texttt{Equal}\;\beta\;\alpha$

$\texttt{transEqual} : \forall\alpha.\forall\beta.\forall\gamma.\texttt{Equal}\;\alpha\;\beta \to \texttt{Equal}\;\beta\;\gamma \to \texttt{Equal}\;\alpha\;\gamma$

(Hint: don't worry too much about how your implementations of these functions *behave*. Focus on defining values of the appropriate types.)

(*b*) Define functions of the following types that convert between the two definitions of equality:

$\texttt{toLeibniz} : \forall\alpha.\forall\beta.\;\texttt{Equal}\;\alpha\;\beta \to \texttt{Eq}\;\alpha\;\beta$

$\texttt{fromLeibniz} : \forall\alpha.\forall\beta.\;\texttt{Eq}\;\alpha\;\beta \to \texttt{Equal}\;\alpha\;\beta$

(8 marks)