$$\begin{aligned}
&\mathrm{Eq} :: * \Rightarrow * \Rightarrow * \\
&\quad = \lambda\alpha :: *.\, \lambda\beta :: *.\, \forall\phi :: * \Rightarrow *.\, \phi\ \alpha \to \phi\ \beta \\
&\\
&\mathrm{refl} : \forall\alpha :: *.\, \mathrm{Eq}\ \alpha\ \alpha \\
&\quad = \Lambda\alpha :: *.\, \Lambda\phi :: * \Rightarrow *.\, \lambda \mathrm{x} : \phi\ \alpha.\, \mathrm{x} \\
&\\
&\mathrm{symm} : \forall\alpha :: *.\, \forall\beta :: *.\, \mathrm{Eq}\ \alpha\ \beta \to \mathrm{Eq}\ \beta\ \alpha \\
&\quad = \Lambda\alpha :: *.\, \Lambda\beta :: *.\, \lambda \mathrm{e} : (\forall\phi :: * \Rightarrow *.\phi\ \alpha \to \phi\ \beta).\, \mathrm{e}\ [\lambda\gamma :: *.\mathrm{Eq}\ \gamma\ \alpha]\ (\mathrm{refl}\ [\alpha]) \\
&\\
&\mathrm{trans} : \forall\alpha :: *.\, \forall\beta :: *.\, \forall\gamma :: *.\, \mathrm{Eq}\ \alpha\beta \to \mathrm{Eq}\ \beta\ \gamma \to \mathrm{Eq}\ \alpha\ \gamma \\
&\quad = \Lambda\alpha :: *.\, \Lambda\beta :: *.\, \Lambda\gamma :: *.\, \lambda \mathrm{ab} : \mathrm{Eq}\ \alpha\ \beta.\, \lambda \mathrm{bc} : \mathrm{Eq}\ \beta\ \gamma.\, \mathrm{bc}\ [\mathrm{Eq}\ \alpha]\ \mathrm{ab} \\
&\\
&\mathrm{Zero} :: * \\
&\quad = \forall\alpha : *.\, \alpha \\
&\\
&\mathrm{Unit} :: * \\
&\quad = \forall\alpha.\, \alpha \to \alpha \\
&\mathrm{unit} : \mathrm{Unit} \\
&\quad = \lambda\alpha.\, \lambda \mathrm{x} : \alpha.\, \mathrm{x} \\
&\\
&\mathrm{Z} :: * \\
&\quad = \mathrm{Zero} \\
&\\
&\mathrm{S} :: * \Rightarrow * \\
&\quad = \lambda \mathrm{X}.\, \mathrm{X} + \mathrm{Unit} \\
&\\
&\mathrm{Neq} :: * \Rightarrow * \Rightarrow * \\
&\quad = \lambda\alpha.\, \lambda\beta.\, \mathrm{Eq}\ \alpha\ \beta \to \mathrm{Zero} \\
&\\
&\mathrm{sz_distinct} : \forall \mathrm{N}.\, \mathrm{Neq}\ (\mathrm{S}\ \mathrm{N})\ \mathrm{Z} \\
&\quad = \lambda \mathrm{N}.\, \lambda \mathrm{eq} : \mathrm{Eq}\ (\mathrm{S}\ \mathrm{N})\ \mathrm{Z}.\ \mathrm{eq}\ [\lambda \mathrm{X}.\, \mathrm{X}]\ (\mathrm{inr}\ [\mathrm{N}]\ \mathrm{unit}) \\
&\\
&\mathrm{eq_pred} : \forall \mathrm{M}.\, \forall \mathrm{N}.\, \mathrm{Eq}\ (\mathrm{S}\ \mathrm{M})\ (\mathrm{S}\ \mathrm{N}) \to \mathrm{Eq}\ \mathrm{M}\ \mathrm{N}
\end{aligned}$$

Figure 1: Definitions in System Fω

1 Types and type inference

(*a*) For each of the following System Fω terms either give a typing derivation or explain why there is no such derivation

(*i*) $\Lambda\alpha{::}*.\lambda \texttt{f}{:}\alpha \to \alpha.\texttt{f f}$

(*ii*) $\lambda \texttt{f}{:}(\forall\alpha{::}*.\alpha).\texttt{f } [\forall\alpha.\alpha \to \alpha]\ [\forall\alpha.\alpha]\ \texttt{f}$

(*iii*) $\lambda \texttt{f}{:}(\forall\phi{::}* \Rightarrow *.\forall\alpha.\phi\ \alpha \to \phi\ \alpha).\Lambda\alpha.\texttt{f } [\alpha]\ [\alpha]$

(6 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 `inl`: `inl M`

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

(4 marks)

(*c*) Both the case for `let` in Algorithm J and the corresponding typing rule only generalize (quantify) type variables that are not already in the context. Explain by means of an example what problems would arise if type variables in the context were also generalized.

(3 marks)

2 De Morgan's laws in Fω

In lecture 2 we saw System Fω encodings of a number of types, including the type with one inhabitant (equivalent to OCaml's `unit` type), and the type with two inhabitants (equivalent to `bool`). We can also encode the type with no inhabitants, following the same pattern:

```
Zero :: *
Zero = ∀α::*.α
```

Under the Curry-Howard correspondence, `Zero` represents the false proposition, and the fact that it has no inhabitants corresponds to the fact that the false proposition has no proof. The fold for `Zero` corresponds to the logical principle *ex falso quodlibet* ("from falsehood anything follows"):

```
fold_Zero : Zero → ∀α::*.α
fold_Zero = λe:Zero.e
```

Using Zero, we can represent logical negation as a type operator :

```
Not :: * ⇒ *
Not = λA.A → Zero
```

and using `Not` we can build System Fω types corresponding to De Morgan's laws:

```
DM_1 = ∀α.∀β.Not (α × β) → (Not α + Not β)

DM_2 = ∀α.∀β.(Not α + Not β) → Not (α × β)

DM_3 = ∀α.∀β.Not (α + β) → (Not α × Not β)

DM_4 = ∀α.∀β.(Not α × Not β) → Not (α + β)
```

(a) Three of De Morgan's laws have proofs in System Fω — that is, it is possible to write System Fω terms that have the corresponding types.

(i) Add a term to `exercise1.f` for each of those three types.

(ii) Identify the type for which it is not possible to define a term, and explain (in a sentence or two) why it is not possible.

(b) De Morgan's laws can be extended from binary connectives to quantified propositions, treating universal quantification as a generalized product and existential quantification as a generalized sum. For example, the following definition may be viewed as a generalized version of `DM_1`:

```
DM_5 = ∀ϕ::* ⇒ *.Not (∀α.ϕ α) → (∃α.Not (ϕ α))
```

(i) Add types `DM_6`, `DM_7` and `DM_8` representing similarly extended versions of `DM_2`, `DM_3` and `DM_4` to `exercise1.f`.

(ii) For each of the types $\mathtt{DM}_5$, $\mathtt{DM}_6$, $\mathtt{DM}_7$ and $\mathtt{DM}_8$, either give a term of that type, or explain why it is not possible to define such a term.

(11 marks)

3 Vectors in Fω

System Fω is a *total* language—i.e. the evaluation of every System Fω expression completes without error, producing a value. For this reason a number of OCaml functions cannot be defined using the System Fω encoding of datatypes seen in the lectures. For example, here is the definition of a function `hd` that returns the first element of a list or fails if the list is empty:

```
val hd : 'a list -> 'a
let hd = function
    [] -> failwith "hd"
  | a::_ -> a
```

Since evaluating `hd v` may fail, there is no equivalent of this function in System Fω.

However, it is possible to define a richer sequence type that distinguishes between empty and non-empty lists, and for which `hd` can be defined. Here is a definition of a `List`-like type function `Vec`, whose two arguments respectively represent the type of the elements in the sequence and the length of the sequence:


```
Vec :: * ⇒ * ⇒ *
Vec = λα.λM.∀ϕ::* ⇒ *.ϕ Z → (∀N.α → ϕ N → ϕ (S N)) → ϕ M
```


For example, a value of type `Vec Nat (S (S Z))` represents a sequence of two `Nat` values.

We assume that we have available type operators `Z` (zero) and `S` (successor) for constructing a representation of numbers:


```
Z :: *
S :: * ⇒ *
```

Furthermore, it will be useful to have representations of some basic facts about numbers. Making use of `Eq` from the lectures and `Not` from Question 2, we can define a type rerepsenting the proposition that one number is the predecessor of another:

```
Pred :: * ⇒ * ⇒ * = λM.λN.Eq (S M) N
```

and a term representing the fact that zero is not the successor of any number:

```
sz_distinct : ∀N.Not (Eql (S N) Z)
```

and a term representing the fact that successor is injective (i.e. that if `S M` is equal to `S N` then `M` is equal to `N`):

```
eq_pred : ∀M.∀N.Eq (S M) (S N) → Eq M N
```

(a) Define `Vec` constructors `nil` and `cons` analogous to the constructors for lists, and with the following types:

```
nil : ∀α.Vec α Z
cons : ∀α.∀μ.α → Vec α μ → Vec α (S μ)
```

(Hint: this is relatively straightforward, and you should not need to make use of any of the facts about numbers.)

(b) Define a function `head` that extracts the first element of a non-empty vector. Your function should have the following type:

```
head : ∀α.∀μ.Vec α (S μ) → α
```

(Hint: this is harder. The key is to find an appropriate instantiation for ϕ in the definition of `Vec`. Remember that the result type of ϕ may be a function type.)

(c) Define tail a function `tail` that removes the first element of a non-empty vector. Your function should have the following type:

```
tail : ∀α.∀μ.Vec α (S μ) → Vec α μ
```

(Hint: this is really quite tricky! You may find it helpful to start by defining an auxiliary function that returns a pair corresponding to the list itself together with its tail.)

(d) The `map` function for `Vec` behaves similarly to `map` for lists: it changes the type of each element, but preserves the length. Give a definition of `map` with the following type:


```
map : ∀α.∀β.∀μ.(α → β) → Vec α μ → Vec β μ
```


(6 marks)