L28: Advanced functional programming

Exercise 1

Due on 12th February 2018

Submission instructions

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

Additional instructions

This exercise involves the use of the `fomega` interpreter. Instructions for installing `fomega` are available on the course webpage:

`https://www.cl.cam.ac.uk/teaching/1718/L28/materials.html`

The accompanying file `exercise1.f` contains a number of definitions, any of which may be used in the solutions to questions 3 and 4.

1 Typing derivations

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

(a) $\Lambda\alpha\texttt{::}*.\lambda\texttt{h}\texttt{:}\alpha \to \alpha.\texttt{h h}$

(b) $\lambda\texttt{f}\texttt{:}(\forall\alpha\texttt{::}*.\alpha).(\textbf{fst}\ (\texttt{f}\ [(1 \to 1) \times 1]))\ (\texttt{f}\ [1])$

(c) $\lambda\texttt{g}\texttt{:}(\forall\phi\texttt{::}* \Rightarrow *.\forall\alpha\texttt{::}*.\phi\ \alpha \to \alpha).\Lambda\gamma.\texttt{g}\ [\lambda\beta.1]\ [\gamma]\ \langle\rangle$

(You may assume the existence of the unit type 1 and value $\langle\rangle$ from Lecture 1)

(6 marks)

2 Type inference and polymorphism

(a) Here are two definitions of the identity function

```
let id = fun x -> x
let id2 = id id
```

OCaml gives the first definition a generalized type:

```
val id : 'a -> 'a = <fun>
```

but gives the second definition a non-generalized type:

```
val id2 : '_a -> '_a = <fun>
```

Consequently, `id` can be applied to arguments of many different types:

```
(id 1, id "two", id false)  ⇝ (1, "two", false)
```

while `id2` can only be applied to arguments of a single type:

```
(id2 1, id2 "two", id2 false)  ⇝ error!
```

Explain why the type checker treats the two definitions differently.

(b) Although the following two programs are very similar, the first program is rejected by OCaml while the second program is accepted.

```
(* rejected by OCaml *)
let f x = (x, (fun y -> x :: y))
let g = f []
let i = (1 :: fst g, "two" :: fst g)

(* accepted by OCaml *)
let f x = (x, (fun y -> x :: y))
let g = fst (f [])
let i = (1 :: g, "two" :: g)
```

Explain briefly why the type checker rejects the first program but accepts the second.

(6 marks)

3 Arithmetic identities in System Fω

A number of equations familiar from high school arithmetic can also be applied to types. For example, in arithmetic the value 1 acts as a right identity for multiplication:

$$\text{For any number } a,$$
$$a \times 1 \equiv a$$

and similarly in lambda calculus it is possible to write functions that convert back and forth between the types $A \times 1$ and A:

$$\begin{array}{ll} \mathtt{f} : \forall \mathtt{A}.\mathtt{A}\times 1 \to \mathtt{A} & \qquad \mathtt{g} : \forall \mathtt{A}.\mathtt{A} \to \mathtt{A}\times 1 \\ \quad = \Lambda \mathtt{A}.\lambda \mathtt{p}{:}\mathtt{A}\times 1.\,\mathbf{fst}\ \mathtt{p} & \qquad \quad = \Lambda \mathtt{A}.\lambda \mathtt{x}{:}\mathtt{A}.\langle \mathtt{x}, \langle\rangle\rangle \end{array}$$

(Furthermore, the composition of `f` and `g` corresponds to the identity function. However, to capture this fact in the types we would need to switch from System Fω to a more expressive calculus such as λC.)

Write similar pairs of System Fω functions that correspond to each of the following three arithmetic identities:

(a) $(a + b) + c \equiv a + (b + c)$
(+ is associative)

(b) $a \times (b + c) \equiv a \times b + a \times c$
($\times$ distributes over +)

(c) $0 \times a \equiv 0$
(0 is an annihilator for $\times$)

(6 marks)

4 Circuits and abstraction

The version of System Fω described in the lectures and implemented in the `fomega` tool supports several additional constructs besides the basic forms for abstraction and application: pairs, sums, and existential types. Although they are convenient, these additional constructs are not entirely necessary, since they can be encoded in the core of the language. This question investigates encodings of existential types.

(*a*) Define an operation

```
Exists :: (* ⇒ *) ⇒ *
```

representing an existential type, along with operations

```
pack_ : ?
open_ : ?
```

for constructing and using existentials.

The following test case may be helpful in developing your definitions. Given a signature for booleans

```
Bools = λβ.β × β×(β → (∀α.α → α → α))
```

you should be able to construct the type `Exists Bools` and the terms involving `pack_` and `open_` on the left hand side below. (The corresponding terms involving the built-in **pack** and **open** are shown for reference on the right.)

```
pack_ [Bools]
      [Unit + Unit]
       ⟨inr [Unit] unit,
        inl [Unit] unit,
        λb:Unit+Unit.
         Λα.
          λr:α.
           λs:α.
            case b of x.s | y.r ⟩
```

```
pack [Unit + Unit],
      ⟨inr [Unit] unit,
       inl [Unit] unit,
       λb:Unit+Unit.
        Λα.
         λr:α.
          λs:α.
           case b of x.s | y.r ⟩
      as ∃β.Bools β
```

```
λp:Exists Bools.
  open_ [Bools] p [Unit]
   (Λβ.λbools:Bools β.
    (@2 (@2 bools))
      (@1 bools)
     [Unit]
     unit unit)
```

```
λp:∃β.Bools β.
  open p as
    β, bools in
      (@2 (@2 bools))
        (@1 bools)
       [Unit]
       unit unit
```

(*b*) It is often useful to build signatures for modules involving parameterized abstract types. For example, the `Queue` and `Stack` mmodules in the standard library both expose an abstract type for the container, paramterized by the type of elements:

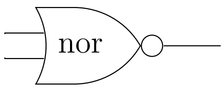
```
module Queue : sig
  type 'a t
  val create : unit -> 'a t
  (* ... *)
```

```
module Stack : sig
  type 'a t
  val create : unit -> 'a t
  (* ... *)
```

Here is a System Fω signature involving parametrized abstract types, and exposing operations for building logical circuits with *nor* gates:

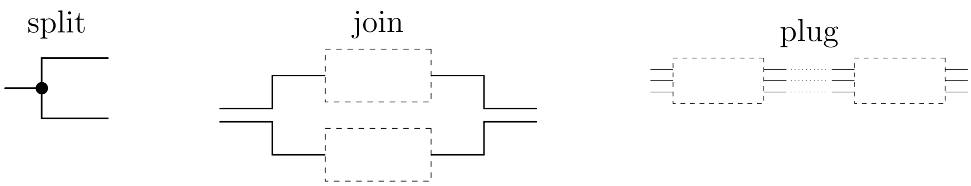
```
Gates =
 λT::* ⇒ * ⇒ *.
  (T (Bool × Bool) Bool) ×                           # nor
  (∀α.T α (α×α)) ×                                   # split
  (∀α.∀β.∀γ.∀δ.T α γ → T β δ → T (α × β) (γ × δ)) ×  # join
  (∀α.∀β.∀γ.T α β → T β γ → T α γ)                   # plug
```

The signature is parameterized by a type constructor `T`; in turn, `T` is parameterised by two types representing the input and output to a circuit. There are four value components. The first, of type `T (Bool × Bool) Bool` represents a *nor* gate with the following standard truth table:



P	**Q**	**nor P Q**
F	F	T
F	T	F
T	F	F
T	T	F

The remaining three components correspond to plumbing operations that can be used to pass the same input to multiple components, and to wire circuits together in parallel or series.



Finally, for reference, here is an OCaml version of `Gates`:

```
module type Gates = sig
  type ('i, 'o) t
  val nor : (bool * bool, bool) t
  val split : ('a, 'a * 'a) t
  val join : ('a,'c) t -> ('b,'d) t -> ('a * 'b, 'c * 'd) t
  val plug : ('a,'b) t -> ('b,'c) t -> ('a,'c) t
end
```

(i) Using the `Gates` signature, define a function `not` with the following type and truth table:

```
not : ∀G::* ⇒ * ⇒ *.
        Gates G → G Bool Bool
```

P	**not P**
F	T
T	F

(*ii*) Using the `Gates` signature (and, if you like, your definition of `not` from the previous question, define a function `and` with the following type and truth table:

```
and : ∀G::* ⇒ * ⇒ *.
     Gates G → G (Bool × Bool) Bool
```

P	**Q**	**and P Q**
F	F	F
F	T	F
T	F	F
T	T	T

(*iii*) Give an implementation of `Gates` as a value of the following type:

```
function_gates : Gates (λX::*.λY::*.X → Y)
```

that represents a circuit as a function from input to output.

(*iv*) Give a second implementation of `Gates` as a value of the following type:

```
count_gates : Gates (λX.λY.Nat)
```

that represents a circuit as a natural number that corresponds to the number of `nor` gates in the circuit.

(*v*) Since the type component `T` of the `Gates` signature does not have kind `*`, `Gates` cannot be used with `Exists`.

Define `Exists2`, a variant of `Exists` for binary type operators, along with constructor and deconstructor `pack2_` and `open2_` that are suitable for use with `Gates`. You should ensure that the following expressions pass type checking with your definitions:

```
Exists2 Gates

pack2_ [Gates] [λα.λβ.α → β] function_gates

pack2_ [Gates] [λα.λβ.Nat] count_gates

λg:Exists2 Gates.
 open2_ [Gates] g [Unit]
 (ΛG::*⇒ *⇒ *.λg:Gates G.
   (λg:G Bool Bool.unit)
   (plug [G] g [Bool] [Bool] [Bool] (not [G] g) (not [G] g)))
```

(12 marks)