— *Solution notes* —

COMPUTER SCIENCE TRIPOS Part IA – 2014 – Paper 1

1 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

types, polymorphism

(*a*) Write brief notes on polymorphism in OCaml, using lists and standard list functions such as `@` (append) and `List.map`. [4 marks]

Answer: Key points are that polymorphism assigns a type to every expression — at compile time — while at the same time allowing natural genericity. For instance, the elements of a list must have the same type, but it can be any type. The type of append, `'a list -> 'a list -> 'a list`, indicates that it combines two lists of the same type, returning another list of that type. The type of `map`, `('a -> 'b) -> 'a list -> 'b list`, indicates that it transforms a list of one type to another, as indicated by the type (`'a -> 'b`) of the function.

---

variants, functions

(*b*) Explain the meaning of the following declaration and describe the corresponding data structure, including the role of polymorphism.

```
type 'a se = Void | Unit of 'a | Join of 'a se * 'a se
```

[4 marks]

Answer: This declares a variant type containing three constructors: `Void`, `Unit` and `Join`. The latter two constructors require arguments, while `Void` is a constant. This is a tree-like data structure with unlabelled binary branching (`Join`), labelled leaves (`Unit`) and unlabelled leaves (`Void`). Type `'a se` is polymorphic, as indicated by the type variable `'a`, which shows that `'a` is the type of the labels. Functions involving the new type can be declared using pattern matching.

---

variants, functions, recursion

(*c*) Show that OCaml lists can be represented using this variant type by writing the functions `encode_list` of type `'a list -> 'a se` and `decode_list` of type `'a se -> 'a list`, such that `decode_list (encode_list xs) = xs` for every list `xs`. [3 marks]

Answer:

```
let rec encode_list = function
| [] -> Void
| x::xs -> Join (Unit x, encode_list xs)

exception Not_a_list
let rec decode_list = function
| Void -> []
| Join (Unit x, v) ->
    x :: decode_list v
| _ -> raise Not_a_list
```

---

functions as values

(*d*) Consider the following function declaration:

```
let rec cute p = function
```

```
| Void -> false
| Unit x -> p x
| Join (u, v) ->
    cute p u || cute p v
```

What does this function do, and what is its type? [4 marks]

Answer: The function `cute` has type `('a -> bool) -> 'a se -> bool`, and `cute p s` returns true if and only if `s` contains an element of the form `Unit x`, where `p x` is true. It is analogous to the function `exists`, for lists.

---

functions as values

(*e*) Consider the following expression:

```
fun p -> cute (cute p)
```

What does it mean, and what is its type? Justify your answer carefully. [5 marks]

Answer: This is a function of type `('a -> bool) -> 'a se se -> bool`. Through the `fun` binder, it takes an argument `p`, which has type `'a -> bool`. Now `cute p` has type `'a se -> bool`, and because `cute` is polymorphic, `cute (cute p)` is well-defined and has type `'a se se -> bool`.

Now if `fun p -> cute (cute p)` is applied to some specific `p` and then to a term `s`, it returns true if and only if `s` contains an element of the form `Unit x`, where `cute p x` is true. Thus the expression is like `cute` but for type `'a se se -> bool`, that is, for the data structure nested in itself.
