Department of Computer Science and Engineering
University of California at San Diego CSE 230
Winter 1998

REVISED Problem Set 3 Solutions

1. Consider an infix type operator $+$ that returns a disjoint sum type.

   (i) What feature of standard ML gives about the same effect as this operator?

   (ii) Nevertheless, suppose the $+$ type operator were added to ML, and suppose there were builtin polymorphic constructors `inleft`: $\alpha \rightarrow \alpha + \beta$ and `inright`: $\beta \rightarrow \alpha + \beta$, and destructors `outleft`: $\alpha + \beta \rightarrow \alpha$ and `outright`: $\alpha + \beta \rightarrow \beta$.

   Explain why some expressions using these operators cannot be type-checked at compile-time. Is this compatible with the spirit of ML? Is it possible for ML to deal sensibly with these polymorphic constructors and destructors?

   (iii) Suggest a *single* builtin polymorphic function that would provide the functionality of `outleft` and `outright` and would allow full compile-time type-checking. Be sure to specify the signature of your suggested function clearly.

   [This sample solution could be expanded.] (i) Types defined by the programmer with the `datatype` keyword.

   (ii) It is only correct to apply `outleft` to a value of of type $\alpha + \beta$ that was created using `inleft`. At compile-time, it is in general impossible to tell whether `inleft` or `inright` was used to create an $\alpha + \beta$ value. Therefore, type-checking of expressions involving `outleft` and `outright` would have to be done at run-time. This violates the ML design principle of performing all type-checking at compile-time. Even with run-time checking, it is hard to deal sensibly with `outleft` and `outright`: what sort of exception should be raised when `outright` is applied to a value created using `inleft`, or vice versa?

   (iii) One possibility is a builtin higher-order function, called `split` say, that requires two alternative functions, one for the α case and one for the β case. These functions must yield results of the same type, say γ. Assuming that `split` is curried, its signature would be

$$(\alpha + \beta) \rightarrow (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow \gamma$$

   This function resembles the standard `if-then-else` operator in that only one of its two alternatives is ever invoked, but both alternatives must still have the same result type.

2. (i) Define your own list append function in ML, and use it in the obvious function to flatten binary trees. What is the time complexity of this function?

   (ii) Define a less obvious function that flattens binary trees in linear time.

   [Solution due to Giovanni Di Crescenzo.] The usual type for binary trees is:

```
datatype btree = empty | leaf of int | node of btree * btree;
```

   My own list append function is:

```
fun myappend (nil,list) = list
  | myappend (h :: tail,list) = h :: myappend(tail,list);
```

   The obvious function to flatten binary trees returns `nil` if the tree is empty, the integer `x` if the tree is just a leaf containing `x`, and recursively flattens the left and right subtrees otherwise.

```
fun flatten1(empty) = nil
  | flatten1(leaf x) = [x]
  | flatten1(node(l,r)) = myappend(flatten1(l),flatten1(r));
```

   The function has worst-case time complexity $O(n^2)$ where in the worst case the `myappend` function takes time $O(n)$, as the its first argument may be $O(n)$ long. The worst-case example for this is a binary tree which is completely unbalanced to the left.

   A less obvious function that flattens binary trees in linear time recursively visits all the nodes of the tree from the right subtree to the left and adds each time one new leaf to a list.

```
fun fl2(empty,fringe) = fringe
  | fl2(leaf x,fringe) = x :: fringe
  | fl2(node(l,r),fringe) = fl2(l,fl2(r,fringe));

fun flatten2(T) = fl2(T,nil);
```

   As the operator `::` takes constant time, and each time only one new leaf is inserted in the list fringe, the above function has worst-case time complexity $O(n)$. Here is an example for both functions.

```
- flatten1(node(node(leaf 0,leaf 1),node(leaf 2,leaf 3)));
val it = [0,1,2,3] : int list

- flatten2(node(node(leaf 0,leaf 1),node(leaf 2,leaf 3)));
val it = [0,1,2,3] : int list
```