Homework 1: ML

The following are some exercises to gain some proficiency in ML, to investigate some aspects of its design, and to experiment with some concepts from category theory.

Consider this homework to be "due" on September 7.

1. Suppose trees are defined as follows:

   ```
   datatype 'item tree = L of 'item | N of 'item * 'item tree * 'item tree
   ```

   Define a `size` function on trees that counts the number of items in a tree.

2. Write a functor with the following heading to implement points:

   ```
   functor Pair(C: sig type coord end):
           sig
             type  point
             val mkPoint: C.coord * C.coord -> point
             val x_coord: point -> C.coord
             val y_coord: point -> C.coord
           end
   ```

   The above heading is not quite as modular as one might like, since the result signature of `Pair` is not named. Can this be fixed in ML? That is, can one declare signatures `COORD` and `POINT` such that the functor `Pair` has the signature

   ```
   functor Pair(C: COORD) : POINT
   ```

   or something similar to this? Does your answer express the dependence of the result signature on the type `coord` in the argument structure? Is it possible to express this dependence with separate `COORD` and `POINT` signatures in ML?

3. There are two ways to define abstract data types in ML in such a way that representation is hidden from clients: `abstype` and structures with coercive signature mapping. What are the differences between these two facilities?

4. Complete the implementation of the symbol table functor from the "modules" lecture by writing the signatures `IntMapSig`, `ValSig`, and `SymSig` and finishing the body of `SymTblFct`. Make sure your functor compiles.

5. Rydeheard and Burstall, in their book *Computational Category Theory*, represent a category in the core language of ML by a 4-tuple as follows

   ```
   datatype ('o,'a)Cat =
      cat of ('a->'o)*('a->'o)*('o->'a)*(('a*'a)->'a)
   ```

where 'o is the type of the objects, 'a is the type of the arrows (morphisms), and a 4-tuple cat(s, t, i, c) contains: s which maps arrows to their source (domain) object, t which maps arrows to the target (codomain, range) object, i wich gives the identity arrow for any given object, and c which is a composition operator.

(a) The category of finite sets and typed total functions, **FinSet**, is defined (mathematically) as follows: the objects are finite sets, an arrow $f : a \rightarrow b$ is a total function from a to b. For each finite set a, the identity arrow $i_a : a \rightarrow a$ is simply $\lambda x : a \,.\, x$ and composition is defined as usual.

Assume that there is a polymorphic type Set implemented in ML, so that the type 'e Set is the type of finite sets whose elements have type 'e. Let Set_Arrow be the polytype:

```
datatype 'e Set_Arrow =
   set_arrow of ('e Set) * ('e->'e) * ('e Set)
```

There is no way to ensure that the ML function (of type 'e->'e) is a total function from the source set (on the left) to the target set (on the right). One can define the various components of the category using this representation for the arrows. For example, the components of the category can be defined as follows:

```
fun set_s(set_arrow(a,_,_)) = a;
fun set_t(set_arrow(_,_,b)) = b;
fun set_ident(a) = set_arrow(a, fn x => x, a);
fun set_comp(set_arrow(c,g,d), set_arrow(a,f,b)) = ...
```

Complete the above definition of set_comp. (Assume that seteq tests equality of finite sets. What should be done if b is not equal to c?)

The category of finite sets can then be defined by

```
val FinSet: (''a Set,''a Set_Arrow)Cat =
    cat(set_s, set_t, set_ident, set_comp)
```

(b) A functor is mapping between categories that preserve the structure of the categories involved. Let **A** and **B** be categories, a functor $F : \mathbf{A} \rightarrow \mathbf{B}$ consists of two functions, one from the objects of **A** to the objects of **B**, and another from the arrows of **A** to those of **B**. Both of these functions are usually denoted by F. These functions must satisfy the following

$$F(i_a) = i_{F(a)} \tag{1}$$
$$F(g \circ f) = F(g) \circ F(f) \tag{2}$$

whenever the composition $g \circ f$ is defined.

Rydeheard and Burstall represent a functor as follows

```
datatype ('oA, 'aA, 'oB, 'aB)Functor =
        ffunctor of ('oA,'aA)Cat * ('oA->'oB) * ('aA->'aB) * ('oB,'aB)Cat
```

where ffunctor is used as the type constructor to avoid the ML reserved word functor. For example, the identity functor on a category A is defined by

```
fun I(A) = ffunctor(A, fn x => x, fn x => x, A)
```

Consider the functor $X : \mathbf{FinSet} \to \mathbf{FinSet}$ that maps a set a to the cartesian product $a \times a$ and that maps an arrow f to $f \times f$, where

$$(f \times f)(x, y) = (f(x), f(y)). \tag{3}$$

Complete the following representation of the functor:

```
fun cartesian_prod(A: ''a Set, B: ''a Set): (''a*''a)Set = ...
fun prod_arrow(set_arrow(A,f,B), set_arrow(C,g,D)) = ...
val X = ffunctor(FinSet,
                 fun A => cartesian_prod(A,A),
                 fun f => prod_arrow(f,f),
                 FinSet)
```

An implementation of `Set` will be provided if you wish to test your answer.

(c) For a category $\mathbf{A}$, the dual category, $\mathbf{A}^{\mathrm{op}}$ (Hoare uses $\breve{}\mathbf{A}$) has as its objects the objects of $\mathbf{A}$. An arrow $f : b \to a$ in $\mathbf{A}^{\mathrm{op}}$ is the same as the arrow $f : a \to b$ in $\mathbf{A}$. (Note: the arrows in $\mathbf{A}^{\mathrm{op}}$ are *not* the inverses of the arrows in $\mathbf{A}$, only the source and target mappings are reversed.) Thus if $f : a \to b$ and $g : b \to c$ are arrows of $\mathbf{A}^{\mathrm{op}}$, then their composition is the arrow $f \circ g : c \to a$ in $\mathbf{A}$. The dual of a functor $F : \mathbf{A} \to \mathbf{B}$ is the same pair of functors as F, but between the dual categories: $F^{\mathrm{op}} : \mathbf{A}^{\mathrm{op}} \to \mathbf{B}^{\mathrm{op}}$.

Define ML functions `dual_Cat` and `dual_Fun` that take a category or functor and return its dual.

(d) Reimplement the above using structures and functors (the ML kind), instead of datatypes. What are the advantages of the Rydeheard and Burstall representation?