Recitation 9
Set and map interfaces

Collections

We have now seen several types of collections - lists, queues, stacks, maps, and various sorts of trees. You will find most data structures in SML share something like the following interface:

```
(* THERE'S NO IMPLEMENTATION OF THIS *)
signature COLLECTION = sig
  type 'a collection

  (* These functions are nearly always present*)
  val empty: 'a collection (* what parameters might this have? *)
  val map: ('a -> 'b) -> 'a collection -> 'b collection
  val app: ('a -> unit) -> 'a collection -> unit
  (* If the collection is ordered, there may be foldl and foldr *)
  val fold: (('a * 'b) -> 'b) -> 'b -> 'a collection -> 'b

  (* These functions are sometimes present *)
  val size: 'a collection -> int
  val exists: ('a -> bool) -> 'a collection -> bool
  val find: ('a -> bool) -> 'a collection -> 'a option
  val filter: ('a -> bool) -> 'a collection -> 'a collection
end
```

When writing your own collection types, you ought to provide functions like this for the user. When looking to use a collection, you ought to try to use these functions before writing your own equivalent code.

Sets and maps are two common abstract data types; let's look at signatures for each of these, and how the two types relate.

```
signature SET = sig
  type 'a set
  val add: ('a set * 'a) -> 'a set
  val isMemberOf: ('a set * 'a) -> bool
end
```

A set might also include other operations, such as union, intersection, difference, isSubset, singleton and the collection operations above, but all can be derived from **`add`** and **`isMemberOf`** (and **`empty`**).

```
signature MAP = sig
  type ('key, 'value) map
  val insert: (('key, 'value) map * 'key * 'value) -> ('key, 'value) map
  val find: (('key, 'value) map * 'key) -> 'value option
end
```

A map might also have an operation to remove a binding. Maps will probably also include general collection functions - **`map`**, **`fold`**, etc, but one wrinkle with these is that when iterating over the map, one might want to look at all keys, all values, or all key, value pairs. Different map interfaces will provide one or more of these options.

It is actually possible to implement sets in terms of maps, and maps in terms of sets. Consider the following:

```
(* sets in terms of maps *)
val add (set, item) = insert (map, item, 1) (* or true or item or whatever *)
val isMemberOf (set, item) =
  case find (map, item) of
    NONE -> false
  | SOME _ -> true

(* maps in terms of sets - very mathematical *)
val insert (map, key, value) = add set (key,value)
val find (map, key) = member set (key, _)
```

Note that to make the latter implementation work, we must implement the set so that when it compares entries, it only compares the key part of each key, value pair.

Comparison functions

Any implementation of a set or map depends on a way of comparing elements (or in the case of a map, comparing keys). What sort of comparison is required depends on the implementation. Let's consider implementations of a map. For an association list (see **Recitation 8**), only a notion of equality is needed. For a hashtable, we need both a notion of equality as well as a hash function from our key type to integers. Finally, for a tree-based implementation, we need a comparison function returning a total ordering.

Mutability

Programming in a purely functional or applicative manner requires the use of immutable data structures. The signatures we have provided so far assume this paradigm. It is also possible to have mutable collections, such as hashtables. In that case, a signature will look more like this:

```
signature MUTABLE_MAP = sig
  type ('key, 'value) map
  val insert: (('key, 'value) map * 'key * 'value) -> unit
  val find: (('key, 'value) map * 'key) -> 'value option
end
```

That is, the insert function, rather than returning a new, immutable map with the new key, instead modifies the existing map. It is often possible to gain an efficiency improvement by using mutable structures - hash tables offer O(1) access and update, assuming the hash function is good. On the other hand, mutable structures are more complex to reason about, both theoretically and practically.

Sets and maps in SML/NJ

The preceding discussion has all been abstract, but now we will talk about the actual set and map structures available in SML. SML proper does not have any such structures, but SML of New Jersey does. Most of the functions are available through the use of *functors*. A functor is a module that takes a module as an argument and produces a module as output. A functor is to a module as a function is to a value. For now, all you need to know is the syntax for this.

```
structure ListSetOfInts = ListSetFn (struct
  type ord_key = int
  val compare = Int.compare
end)
```

What's going on here is that we are defining an (anonymous) structure containing one type (ord_key) and one value (compare), and passing that structure as an argument to the **`ListSetFn`** functor. The return value is a module having the signature **`ORD_MAP`**, which we give the name ListSetOfInts. The argument of **`ListSetFn`** must conform to the signature **`ORD_KEY`**.

SML/NJ offers a number of functors returning sets and maps: RedBlackSetFn, BinarySetFn, ListSetFn, SplaySetFn, RedBlackMapFn, BinaryMapFn, ListMapFn, SplayMapFn. Red-Black trees offer an efficient functional implementation based on balanced binary trees; we will discuss them later in the course.

Docmentation for most of these functors is available at **http://www.smlnj.org/doc/smlnj-lib/Manual/toc.html**, but this documentation is not quite up to date (it doesn't include the red-black trees, for example). You can easily look up the actual code and interfaces for these modules in your sml distribution. On Windows it's probably at C:\sml\src\smlnj-lib\Util*.sml, and on Mac it's at SMLNJ/smlnj-lib/Util/*.sml, whereever you put the SML/NJ installation.

One thing to note about these structures is that unlike the examples given so far, they are *not* polymorphic in the type of their key (or item, for sets). Rather, the functor's argument specifies the type. A functorized analogue of the collection signature we began with might look like this:

```
funsig COLLECTION_FUNCTOR (structure Key: sig
                             type key
                             val equals: key*key -> bool
                           end) = sig
  type collection

  val empty: collection
```

```
  val map: (Key.key -> Key.key) -> collection -> collection
  val app: (Key.key -> unit) -> collection -> unit

  (* If the collection is ordered, there may be foldl and foldr.
   * Key would need to have an operation "compare: key*key->order" in that
   * case.
   *)
  val fold: ((Key.key * 'b) -> 'b) -> 'b -> collection -> 'b

  (* Operations like these are often useful. *)
  val size: collection -> int
  val exists: (Key.key -> bool) -> collection -> bool
  val find: (Key.key -> bool) -> collection -> Key.key option
  val filter: (Key.key -> bool) -> collection -> collection
end
```

This has the advantage that collections don't have to keep track of their own key operations. Also, if you have two collections of the same type, you know that they have the same underlying key comparison function, which helps avoid mistakes. For map abstractions, it is also possible to have a map that is polymorphic with respect to the value type but for which the key type is passed as a functor parameter. Unfortunately, SML makes you decide which style you want. But you're still better off with any of these styles than in many languages.

Hash tables

SML/NJ also contains hash tables. These are *not* functorized, but instead the equivalent of the **`empty`** function takes arguments which specify the comparison functions

```
(* raised when I do a lookup and can't find something *)
exception Can't_Find_It

(* used to convert a string to a hash code -- i.e., word *)
val hash_fn : string->word = HashString.hashString

(* used to compare keys in the hash table *)
fun cmp_fn : string*string->bool = (op =)

(* initial table size -- need something relatively prime and
* approximately the size of the number of elements you expect
* to be in the hash table.  Note that the implementation resizes
* as needed to avoid major collisions. *)
val initial_size : int = 101

val t : (string,int) table = mkTable (hash_fn, cmp_fn)
(initial_size, Can't_Find_it)

insert t ("Eric",30);
insert t ("Jason",24);
lookup t "Jason" (* returns 30 *)
lookup t "Daniel" (* raises Can't_Find_It exception *)
```