1 Dictionaries with polymorphic key and value types

So far we have implemented dictionaries with polymorphic values but with fixed key types (string or int usually). The following is a possible signature for dictionaries that are polymorphic in both their key and value types. Observe that the insert and lookup functions now require a comparison function as an argument.

```
signature POLYDICT =
sig
  type ('k, 'v) dict
  val empty : ('k, 'v) dict
  val insert : ('k * 'k -> order) -> ('k, 'v) dict -> ('k * 'v) -> ('k, 'v) dict
  val lookup : ('k * 'k -> order) -> ('k, 'v) dict -> 'k -> 'v option
end
```

Here is an example implementation using binary search trees:

```
structure TreeDict : POLYDICT =
struct
    (* Representation Invariant: The tree is a binary search tree. *)
    datatype ('k, 'v) tree = Empty |
                          Node of ('k, 'v) tree * ('k * 'v) * ('k, 'v) tree
    type ('k, 'v) dict = ('k, 'v) tree

    val empty = Empty

    fun lookup cmp d k =
      case d of
        Empty => NONE |
        Node (L, (k', v'), R) =>
          (case cmp (k,k') of
            EQUAL => SOME v' |
            LESS => lookup cmp L k |
            GREATER => lookup cmp R k)
```

*Adapted from a document by Dan Licata.
fun insert cmp d (k, v) = 
  case d of 
    Empty => Node (empty, (k,v), empty) 
  | Node (L, (k’, v’), R) => 
    (case cmp (k,k’) of 
      EQUAL => Node (L, (k, v), R) 
    | LESS => Node (insert cmp L (k, v), (k’, v’), R) 
    | GREATER => Node (L, (k’, v’), insert cmp R (k, v))) 
end

Are there any issues with this implementation?
Potentially, yes.
The reason is somewhat subtle: a type can be ordered in more than one way. For example, in
addition to Int.compare, which compares integers using the normal less-than,

(* compare x and y mod 1024 *)
fun compareMod (x:int,y:int) = ... 

compares integers mod 1024.
For example, if you insert using Int.compare, then your tree will be sorted in increasing order
according to Int.compare. In particular, if keys 1023 and 1025 are present, with 1023 at the root
of the tree, 1025 will be to the right of 1023 in the tree. If you then lookup using compareMod,
according to which 1025 is less than 1023 (since 1025 mod 1024 = 1), lookup will go left, rather
than right, and not find the key.

What is the problem here? The root of the issue is that the spec

(* Representation Invariant: The tree is a binary search tree. *)

doesn’t make sense: relative to which comparison function is the tree sorted?
One solution is to change the spec and REQUIRE: A user must use the same cmp function for
all insertion and lookup operations (initialized with some empty dictionary).
Unfortunately, this solution doesn’t the address the issue of someone intentionally breaking
the system or even doing so forgetfully. A second solution is to use the type system to enforce
consistency of the comparison function, by bundling that function together with the key type, and
making dictionaries with different comparison functions be different types. To accomplish this, we
need the idea of a type class.

2 Type Classes
A type class is a mode of use of signatures, in which one describes a type equipped with a (not
necessarily exhaustive) collection of operations. For example:
signature ORDERED = 
sig
  type t
  val compare : t * t -> order
end
The signature ORDERED describes a type \( t \) equipped with a comparison function. Here are some structures that satisfy this signature. Observe that the same type can be ORDERED in different ways, and different types can be ORDERED.

```ocaml
structure IntLt : ORDERED =
  struct
    type t = int
    val compare = Int.compare
  end

structure IntMod : ORDERED =
  struct
    type t = int
    val compare = compareMod
  end

structure StringLt : ORDERED =
  struct
    type t = string
    val compare = String.compare
  end
```

What do clients of these structures know? They know that \( \text{IntLt}.t = \text{int} \), \( \text{IntMod}.t = \text{int} \), \( \text{StringLt}.t = \text{string} \). These types are not abstract.

Recall that an abstract type is a type specified in a signature without specific implementation, and whose actual implementation by a structure is not something a client can manipulate directly (either because the structure ascribes opaquely to the signature or because the structure uses a datatype declaration whose constructors are not specified in the signature or otherwise exported).

When a type is abstract, its signature is prescriptive: the signature prescribes exactly what one can do with the type.

When the type is not abstract, the signature is descriptive: it describes some of the operations that the type supports. This is usually the right choice for a type class, because one wants to use the operations on values constructed elsewhere. E.g., one might want to evaluate \( \text{IntLt}.\text{compare}(3, 5) \)—if the type \( \text{IntLt}.t \) were abstract, this would not be allowed.

### 3 Substructures

We can tie the comparison function to the key type using a substructure in the dictionary signature:

```ocaml
signature DICT =
sig
  structure Key : ORDERED
    type 'v dict

  val empty : 'v dict
  val insert : 'v dict -> (Key.t * 'v) -> 'v dict
  val lookup : 'v dict -> Key.t -> 'v option
end
```
The first component is a structure `KEY` that matches the `ORDERED` signature. The later components can refer to the type components of this substructure using dot notation—`Key.t`.

The signature says that an implementation comes with a particular key type, rather than supplying a type `dict` that is parameterized by the key type.

For example, here is a dictionary where the keys are integers:

```haskell
structure IntLtDict : DICT =
  struct
    structure Key : ORDERED = IntLt
      datatype 'v tree =
        Empty
      | Node of 'v tree * (Key.t * 'v) * 'v tree
  
    type 'v dict = 'v tree

    val empty = Empty

    fun lookup d k =
      case d of
        Empty => NONE
      | Node (L, (k', v'), R) =>
        (case Key.compare (k,k') of
          EQUAL => SOME v'
        | LESS => lookup L k
        | GREATER => lookup R k)

    fun insert d (k, v) =
      case d of
        Empty => Node (empty, (k,v), empty)
      | Node (L, (k', v'), R) =>
        (case Key.compare (k,k') of
          EQUAL => Node (L, (k, v), R)
        | LESS => Node (insert L (k, v), (k', v'), R)
        | GREATER => Node (L, (k', v'), insert R (k, v)))
  end
```

In later components, we refer to the components of substructures using dot notation (e.g., `IntLtDict.Key.t` and `IntLtDict.Key.compare`).

In client code, one can refer to components of substructures using dot notation (e.g., `IntLtDict.Key.t` and `IntLtDict.Key.compare`).
How could one make a dictionary whose keys are integers compared mod 1024? Answer:

```
structure IntModDict : DICT =
struct
  structure Key : ORDERED = IntMod
      datatype 'v tree =
          Empty
          | Node of 'v tree * (Key.t * 'v) * 'v tree
  type 'v dict = 'v tree

... copy and paste same code as before ...

How about a dictionary whose keys are strings? Answer:

structure StringDict : DICT =
struct
  structure Key : ORDERED = StringLt
      datatype 'v tree =
          Empty
          | Node of 'v tree * (Key.t * 'v) * 'v tree
  type 'v dict = 'v tree

... copy and paste same code as before ...
```

Questions:

- Is IntLtDict.dict equal to StringDict.dict? On the surface, it looks like they are defined by the same datatype declaration. But in one case, Key.t is int and in the other it is string. So it would be unsound to consider these types equal—your program would crash!

- Is IntLtDict.dict equal to IntModDict.dict? This would be sound, but it is undesirable—we would still be able to insert using Int.compare, and lookup using compareMod, which is exactly the problem we have been trying to solve!

Fortunately, SML gets this right:

```
Every time you evaluate a datatype declaration, you get a new type.
```

This is know as *datatype generativity.*

The type IntLtDict.tree is different than the type IntModDict.tree, because the two types come from different evaluations of a datatype declaration (even though it is the same textual piece of code). Using this mechanism, we can make different types for dictionaries sorted by different comparison functions, which avoids the issue discussed on page 2.
4 Functors

Unfortunately, we’ve also introduced a lot of code duplication, because we had to copy and paste the dictionary implementation for each key type.

We can fix this with a functor, which is a function from structures to structures. For example:

```haskell
functor TreeDict(K : ORDERED) : DICT =
struct
  structure Key : ORDERED = K

datatype 'v tree =
  Empty
  | Node of 'v tree * (Key.t * 'v) * 'v tree

type 'v dict = 'v tree

val empty = Empty

fun lookup d k =
  case d of
    Empty => NONE
  | Node (L, (k', v'), R) =>
      (case Key.compare (k,k') of
        EQUAL => SOME v'
      | LESS => lookup L k
      | GREATER => lookup R k)

fun insert d (k, v) =
  case d of
    Empty => Node (empty, (k,v), empty)
  | Node (L, (k', v'), R) =>
      (case Key.compare (k,k') of
        EQUAL => Node (L, (k, v), R)
      | LESS => Node (insert L (k, v), (k', v'), R)
      | GREATER => Node (L, (k', v'), insert R (k, v)))
end
```

TreeDict is the name of the functor; it takes an argument structure K which has signature ORDERED; and it produces a DICT. The implementation is the same code that we had been cutting and pasting before, after defining the Key component of the result to be the structure K. This is why we wrote Key.t and Key.compare above, even though we didn’t have to: in fact the code works generically in any key type and comparison function.

We can create our earlier structures by applying the functor to an argument, which must satisfy the declared argument signature:

```haskell
structure IntLtDict : DICT = TreeDict(IntLt)
structure IntModDict : DICT = TreeDict(IntMod)
structure StringDict : DICT = TreeDict(StringLt)
```
Questions:

- **Is IntLtDict.Key.t equal to int?** Yes! SML propagates the definitions: In the functor body, Key is defined to be the argument K, and K is instantiated by IntLt, and IntLt.t is int. None of these are abstract types, so the definitions propagate through.

- **Is IntModDict.dict equal to IntLtDict.dict?** No! Each time you apply a functor, you evaluate its body, which generates a new copy of each datatype in it. So the abstract types provided by different applications of a functor are different.