In the last lecture, we saw dictionaries that used strings as keys and had polymorphic values. Today, we’ll try several times to make dictionaries with polymorphic keys.

1 Dictionaries, Take 1

Let’s just go ahead and make a signature for dictionaries with keys ‘k and values ‘v.

signature DICT =
sig
 (* dictionaries map keys ‘k to values ‘v *)
 type (‘k, ‘v) dict

 (* the empty mapping *)
 val empty : (‘k, ‘v) dict

 (* insert cmp (k1 ~ v1, ..., kn ~ vn) (k,v)
   == (k1 ~ v1, ..., ki ~v,...) if cmp(k,ki) == EQUAL for some ki
   == (k1 ~ v1, ..., kn ~ vn, k ~ v) otherwise *)
 val insert : (‘k * ‘k -> order) -> (‘k, ‘v) dict -> (‘k * ‘v) -> (‘k, ‘v) dict

 (* lookup cmp (k1 ~ v1,...,kn ~ vn) k
   == SOME vi if cmp(k,ki) == EQUAL for some ki
   == NONE otherwise *)
 val lookup : (‘k * ‘k -> order) -> (‘k, ‘v) dict -> ‘k -> ‘v option
end

Here, we’ve annotated the signature with a specification of the behavior of each operation, in terms of a mathematical dictionary notation (k1 ~ v1, ..., kn ~ vn). E.g. (1 ~ true, 2 ~ false) represents the dictionary that maps 1 to true and 2 to false. These mathematical dictionaries model the actual dictionaries as a set of key-value pairs. This way, you can reason about the behavior of your code in terms of this abstraction, without knowing the particular implementation.

*Based on notes by Brandon Bohrer and Michael Erdmann
Here's an example implementation as trees:

```ml
structure BSTDict : DICT =
struct
  (* Invariant: every function takes and returns 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 Empty k = NONE
    | lookup cmp (Node (l, (k', v'), r)) k =
        (case cmp (k, k') of
          EQUAL => SOME v'
        | LESS => lookup cmp l k
        | GREATER => lookup cmp r k)

  fun insert cmp Empty (k, v) = Node (empty, (k,v), empty)
    | insert cmp (Node (l, (k', v'), r)) (k, v) =
        (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
```

Does this implementation of dictionaries as trees meet the above spec?

In fact, it doesn't. 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,

```ml
(* compare x and y using >=, not <= *)
fun compareGt (x:int,y:int) = Int.compare (y,x)
```

compares integers in the opposite ordering of `Int.compare`.

If you insert using `Int.compare`:

```ml
fun ins d p = TreeDict.insert Int.compare d p
val t1 = ins (ins (ins TreeDict.empty (1,"c")) (2,"a")) (3,"b")
```

then your tree will be sorted in increasing order according to `Int.compare`; in particular, 3 will be to the right of 2. If you then lookup using `compareGt`, according to which 3 is less than 2, lookup will go left, rather than right, and not find it!

That is, there is an invariant violation: `compareGt (3,3) == EQUAL` and 3 ~ "b" is in the model of the dictionary, but `lookup` returns `NONE`.

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

```ml
(* Invariant: every function takes and returns a binary search tree *)
```
invariant on the datatype doesn't make sense: which comparison function is the tree sorted according to? We're assuming very implicitly that the comparison function is always the same, but there's nothing to guarantee that.

One solution is to change the spec: What you want to say is that, if you `lookup cmp d` where `d` is sorted according to `cmp`, then you will get the appropriate result. That is, which dictionaries are appropriate to pass to `lookup cmp` depends on `cmp`. This gets a bit complicated: in the signature we could state that `lookup` works for sorted dictionaries, but for every implementation of structures we would need to redefine what it means to be sorted (since we can use different types in each implementation).

However, this is still not optimal: specs only get checked by humans (and only if you're lucky!). It would be much better if we could use the type system to make sure that we use the right comparison function.

We do this by bundling the comparison 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 certain kind of signature which describes one type and an operation (or sometimes multiple operations) on that type. What makes a type class different from other signatures is that we are not trying to give all the operations for a given type, we just want to know that it supports a certain operation.

For example:

```plaintext
signature ORDERED =
sig
  type t
  val compare : t * t -> order
end
```

We use the name `t` to suggest that the type `t` could be almost anything (lots of different types have `compare` functions). This signature describes a type `t` equipped with a comparison function. It would not be useful for this to be the only thing you know about `t`—ORDERED doesn't tell you how to make a `t`, and if you can't make a `t` then you can't do anything!  

Here are some structures that satisfy this signature:

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

structure IntGt : ORDERED =
struct

1In this case, `t` is just like tea. I could make green tea, and I could make Pu’er tea, but if you just tell me to make tea then I don’t know what to do unless you tell me what kind of tea I’m making.
type t = int
  fun compare (x, y) = Int.compare (y, x)
end

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

This illustrates that the same type can be ORDERED in different ways, and that different types can be ORDERED.

What do clients of these modules know? They know that IntLt.t = int, IntGt.t = int, StringLt.t = string. These types are not abstract! Why not?

**Methodology:** If you define a type to be a datatype that is not exported in the signature, the type is abstract. If you don’t, it’s not.

In the former case, where you make a type abstract, the signature is prescriptive: it says you can do certain operations, and that’s it!

In the latter case, where you don’t make a type abstract, the signature is descriptive: it describes some of the operations that a type supports. This is usually the right choice for a type class, because you want to use the operations on values that you have around. E.g. you can write IntLt.compare (3,5)—if you made the type abstract, you could never actually call compare. This is why SML automatically propagates type definitions: when you write IntLt : ORDERED, SML writes down that IntLt.t = int, because that’s what’s in the structure. Thus, if you want a type to be abstract, you have to define it to be a type that no one else an do anything with—e.g. a datatype that is not exported.

Actually, we’ve already seen one type class that’s (somewhat) built in to SML.

signature EQUAL =
sig
  type t
  val equal : t * t -> bool
end

If you see a message:

**Warning:** calling polyEqual

when writing code that uses the = operator, what does this mean? All this means is that if you use a polymorphic equality operator, SML has to explicitly create EQUAL modules and pass around equality functions, which makes things a bit slower. So this warning is a just a warning about performance, not correctness.
3 Substructures

We can tie the comparison function to the key type using a substructure. Substructures express hierarchical abstraction: you can build structures out of other structures. Here is the revised dictionary signature:

```plaintext
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 that matches the ORDERED signature. The later components can refer to the type components of a substructure using dot notation—Key.t.

This signature says that an implementation comes with a particular key type, rather than supplying a type dict that is parametrized by the key type. It also does not mean that there is one structure called Key that is somehow inside the signature. It just means that any implementation of DICT must contain a module named Key.

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

```plaintext
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 (Key.compare). In these components, we know that Key.t = int, so we could equivalently have written

| Node of 'v tree * (int * 'v) * 'v tree

and

case Int.compare (k,k’) of

However, the above form is better for reasons that will be clear soon.
In client code, you can refer to components of substructures using dot notation (e.g. IntLtDict.Key.t and IntLtDict.Key.compare).

How do you make a dictionary where the keys are sorted with ≥?

structure IntGtDict : DICT =
struct
  structure Key : ORDERED = IntGt

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

type 'v dict = 'v tree

(* ... *)
end

How about a dictionary whose keys are strings?

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
Every time you write the same datatype declaration, you get a new type.

So the type IntLtDict.tree is different than the type IntGtDict.tree, because they come from different copies of the “same” datatype declaration (the two declarations have the same text). Using this mechanism, we can make different types for dictionaries sorted by different comparison functions, which avoids the above confusion.

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 analogous to a function from modules to modules. For example:

```ml
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), (k', v'), R)
```
TreeDict is the name of the functor; it takes an argument module 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 recover the above modules by applying the functor to an argument, which must satisfy the declared argument signature:

structure IntLtDict : DICT = TreeDict(IntLt)
structure IntGtDict : DICT = TreeDict(IntGt)
structure StringDict : DICT = TreeDict(StringLt)

5 Functors, Polymorphism, Equality Types

There’s actually an interesting connection between functors and polymorphism. Because we have functors in the language, we actually don’t need polymorphism at all. For example, we could implement map on lists as a functor:

signature TWO_TYPES = sig
  type a
  type b
end

signature MAP = sig
  type a
  type b
  val map : (a -> b) -> a list -> b list
end

functor Map(T:TWO_TYPES):MAP =
  struct
    type a = T.a
    type b = T.b
    fun map (f : a -> b) (L : a list): b list =
      case L of
        [] => []
        | x::xs => (f x)::map f xs
  end

And we could then call map by calling the functor:

(* This is analogous to choosing 'a = int, 'b = int *)
structure IntInt : TWO_TYPES =
  struct
type a = int
  type b = int
end

(* Map from ints to ints *)
structure IIM : MAP = Map (IntInt)

(* We also could write the argument anonymously, without giving it the name IntInt *)
structure IIM = Map(struct
  type a = int
  type b = int
  end)

val [5,6,7] = IIM.map (fn x => x + 5) [0, 1, 2]

Why is this useful? Well, this particular example isn’t, but it helps us think about other language features. Remember that in our first try at implementing dictionaries as functions, we expressed the key type as ’’a to allow keys to range over all types that supported equality. In the dictionary functor, we allow keys to range over all types that support comparison. The combination of type classes and functors turns out to allow us to express restricted forms of polymorphism that SML doesn’t support by default.