15-150 Lecture 20: Dictionaries, using Type Classes and Functors

Lecture by Dan Licata

March 29, 2012

1 Dictionaries, Take 1

Here is the signature for dictionaries that you implemented in lab:

```
signature LABDICT =
sig
  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.

Here's an example implementation as trees:

```
structure TreeDict : LABDICT =
struct
  (* Invariant: BST *)
```

```
  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

  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
```

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,

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

compares integers mod 1024.

If you insert using `Int.compare`:

```
fun isins d p = TreeDict.insert Int.compare d p
val t1 = isins (isins (isins TreeDict.empty (1023,"c")) (111,"a")) (1025,"b")
```

then your tree will be sorted in increasing order according to `Int.compare`; in particular, `1025` will be to the right of `1023`. If you then lookup using `compareMod`, according to which `1025` (i.e. `1`) is less than `1023`, lookup will go left, rather than right, and not find it!

That is, there is an invariant violation: `compareMod (1025,1025) == EQUAL` and `1025 ~ "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

```
(* Invariant: BST *)
```

invariant on the datatype doesn't make sense: which comparison function is the tree sorted according to?

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`. To do this, you need an abstract notion of a dictionary being *sorted* for use in the signature, which would then be instantiated to something specific (e.g. "this tree is a BST" for each implementation).

However, instead of putting this in and doing these proofs, we instead *use the type system to enforce this invariant*, 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 mode of use of signatures, where you describe a type equipped with a (probably non-exhaustive) collection of operations on it. For example:

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

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`—there is no way to construct any values!

Here are some structures that satisfy this signature:

```
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
```

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`, `IntMod.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 prescribes exactly what you can do with the type.

In the latter, where you don't, 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.

3 Substructures

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

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

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

```
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 to be preferred for reasons that will become clear later.

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 whose keys are integers compared mod 1024?

```
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?

```
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 ...
end
```

Questions:

- Is `IntDict.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 its `string`. So it would be *unsound* to consider these types equal—your program would crash!
- Is `IntDict.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've been trying to solve all lecture!

Fortunately, SML gets this right:

> every time you evaluate a datatype declaration, you get a new type

So the type `IntLtDict.tree` is different than the type `StringDict.tree`, because they come from different evaluations 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 a function from modules to modules. For example:

```
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 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 IntModDict : DICT = TreeDict(IntMod)
structure StringDict : DICT = TreeDict(StringLt)
```

Questions:

- Is IntLtDict.Key.t equal to int? Yes! ML 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 abstract types (datatypes that aren't exported), so the definitions propagate through.

- Is IntMod.dict equal to IntLt.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.