CS109B ML Notes for the Week of 5/22/95

Abstypes

Goal: complete concealment of the values of a datatype.

- It is impossible to access values of this datatype except through the functions provided.
- Syntax:
  1. A datatype, with the keyword `abstype` in place of `datatype`.
  2. A list of definitions surrounded by `with...end`.

Example: Our previous example of a stack structure allowed stacks to be modified by ways other than the functions provided in the structure.

- It is only a few structures that have that flaw. You need to both:
  a) Have a function like `create` that allows assignment of a value from that structure to an external variable.

  b) Have a type for those values that is modifiable, i.e., a ref or array.

  □ Note, e.g., that an array like `register` in the `Random` structure cannot be attacked because it remains internal to the structure, i.e., condition (a) is not met, and it is declared local.
- Ironclad protection is obtained through the abstype. By wrapping values in a data constructor, these values cannot be seen, let alone modified, other than through the functions defined after the `with`.

  □ That is the abstype's "superpower"; its constructors are local, while those of a datatype are global.

Example: Here is the stack example with values wrapped in data constructor `Stk`.

```
abstype '1a stack =
    Stk of '1a list ref
    with
    exception EmptyStack;

    fun create(x:'1a) = Stk(ref [x]);

    fun push(x,Stk(s)) = s := x:: !s;

    fun top(Stk(ref nil)) = raise EmptyStack
    |   top(Stk(ref(x::xs))) = x;

    fun pop(Stk(ref nil)) = raise EmptyStack
    |   pop(Stk(s)) = s := tl(!s);
    end;
```

- Note that funny type variable `'1a`. It is needed because we define a stack to be a reference to a list of elements of this type, and references cannot be to arbitrary types.
    - The type must involve only concrete subtypes, e.g., `int*int` or `int->int`, but not `'a` or `'a->'a`.
- We can do the usual push, pop, etc., as if the `Stk` weren't there.
- But an attempt to get at the value of a stack directly is doomed to failure:

```
fun grab(Stk(x)) = tl(!x);
```

Error: non-constructor applied to argument in pattern: Stk

Functors (Simple Form)

Consider the structure `Random` from the previous notes. It had built into it a particular size of the `register` array and a particular "feedback function," the positions of the array that got complemented.

- We might like to generate a number of similar structures with different sizes and feedback functions.
- The *functor* is the ML construct that lets us do so. It consists of, in its simplest form:

1. The keyword `functor` followed by the name of the functor.

2. A parenthesized *argument structure* and its signature.

3. An equal sign and the definition of the structure created by the functor from the argument structure.

Example: Here is a signature suitable for the argument of a functor `MakeRandom`.

- This signature describes an integer n (the size of the register) and a list `feed` of the positions in the register that get complemented.

```
signature RANDOM_DATA = sig
    val n : int;
    val feed : int list;
end;
```

The functor `MakeRandom` is in Fig. 1.

- Notice how the output structure of the functor must open the input structure (the line `open Data`) in order to get the needed components n and `feed`.

Applying a Functor

Now, we can define a structure with the correct signature to provide the needed parameters, `n` and `feed`. Here is an example:

```
structure MyData: RANDOM_DATA = struct
    val n = 20;
    val feed = [0,2,4,6,7,14,17,19];
end;
```

Finally, we apply functor `MakeRandom` to the structure `MyData`. The result is another structure, `Random`, that behaves like the old `Random`, but with the new size n and new feedback function.

```
structure Random = MakeRandom(MyData);
```

This structure `Random` is used exactly like the one from the previous notes.

```
functor MakeRandom(Data: RANDOM_DATA):
    sig
        open Data;
        val init: unit -> unit;
        val getBit: unit -> int;
    end
= struct
    open Data;
    val register = array(n,0);
    fun feedback1(nil) = ()
    |   feedback1(x::xs) = (
        update(register,x,1-sub(register,x));
        feedback1(xs));
    fun feedback() = feedback1(feed);
    fun shift1(0) = update(register,0,0)
    |   shift1(i) = (
        update(register,i,sub(register,i-1));
        shift1(i-1));
    fun shift() = shift1(n-1);
    fun init1(0) = (
        update(register,n-1,1);
        update(register,0,0))
    |   init1(i) = (
        update(register,i,0);
        init1(i-1))

    fun init() = init1(n-1);
    fun getBit() =
        let val bit = sub(register,n-1);
        in (
            shift();
            if bit=1 then feedback() else ();
            bit)
        end;
end;
```

Fig. 1. Functor MakeRandom.