Specifying structures in Extended ML

Axioms can be used to specify structures too. A structure will typically contain several functions, and we just have to specify each of them.

For example, here is a structure that implements an unbounded array of integers indexed starting from 0, using a list of integers to represent the array:

```
structure Array =
  struct
    type array = int list
    val empty = nil
    fun put(n,v,nil) = if n=0 then v::nil else 0::put(n-1,v,nil)
      | put(n,v,w::l) = if n=0 then v::l else w::put(n-1,v,l)
    fun retrieve(n,nil) = 0
      | retrieve(n,v::l) = if n=0 then v else retrieve(n-1,l)
  end
```

We can specify this just as if it were two independent functions `Array.retrieve` and `Array.put` and a value `Array.empty`:

```
Forall (n,v,w,l) =>
    Array.put(n,v,Array.put(n,w,l)) = Array.put(n,v,l)
Forall (n,m,v,w,l) =>
    n<>m implies
      Array.put(n,v,Array.put(m,w,l)) = Array.put(m,w,Array.put(n,v,l))
Forall n => Array.retrieve(n,Array.empty) = 0
Forall (n,v,l) => Array.retrieve(n,Array.put(n,v,l)) = v
Forall (n,m,v,l) =>
    n<>m implies Array.retrieve(n,Array.put(m,v,l)) = Array.retrieve(n,l)
```

Recall that the signature associated with a structure plays the role of that structure's interface to the outside world. The signature of `Array` is:

```
signature ARRAY =
  sig
    type array
    val empty : array
    val put : int * int * array -> array
    val retrieve : int * array -> int
  end
```

This signature is sufficient for the purpose of compiling functions which refer to `Array`, but otherwise it is not very informative. For example, it isn't sufficient for proving program correctness.

It's natural to combine the information in the signature with the axioms specifying `Array` to form a more complete interface:

```
signature ARRAY =
  sig
    type array
    val empty : array
    val put : int * int * array -> array
```

```
    val retrieve : int * array -> int
    axiom Forall (n,v,w,l) => put(n,v,put(n,w,l)) = put(n,v,l)
    axiom Forall (n,m,v,w,l) =>
            n<>m implies put(n,v,put(m,w,l)) = put(m,w,put(n,v,l))
    axiom Forall n => retrieve(n,empty) = 0
    axiom Forall (n,v,l) => retrieve(n,put(n,v,l)) = v
    axiom Forall (n,m,v,l) =>
            n<>m implies retrieve(n,put(m,v,l)) = retrieve(n,l)
  end
```

We can ascribe this signature to `Array`, using the usual syntax:

```
structure Array :> ARRAY =
  struct
    type array = int list
    . . .
  end
```

Note the use of `:>` rather than `:` , so `ARRAY` is ascribed opaquely rather than transparently. In EML, it is natural to use opaque signature ascription. This fits with the desire for reusability and independent development of separate parts of a system.

Logical equality. But wait a minute: with opaque signature ascription, `array` doesn't admit equality and so the use of `=` in the axioms won't typecheck!

We don't know how arrays will be represented. In `Array`, we happen to have used `int list`. But it could just as well have been implemented using a function type, e.g. `int->int`.

On the other hand, recall that the main reason why ML doesn't provide equality on all types is that it is impossible to *compute* equality on functional types. This isn't relevant in a specification language like EML: we have already introduced non-computational constructs like quantifiers, so there is no harm in allowing use of equality as well.

So, EML provides a *logical* equality `==` in addition to the *computational* equality `=` provided by ML. When t=t' has a defined boolean value (i.e. when t, t' are of a type that admits equality and when evaluation of t and t' terminate without raising an exception) then t==t' and t=t' coincide. But t==t' is defined for all types, with equality on functional types being *extensional* equality. And if t and/or t' fail to terminate or raise an exception, t==t' holds iff *both* fail to terminate or *both* raise the same exception. So it is extensional with respect to non-termination and exceptions as well. But I've made the assumption that all functions are total, so this case won't ever arise here.

So, to get a correct EML specification we need to replace `=` by `==` in the axioms. Where `=` is used on integers it doesn't matter which one we use, but only because of the totality assumption. A rule of thumb is to use `==` unless you want to emphasize something about computational equality. We get:

```
signature ARRAY =
  sig
    type array
    val empty : array
    val put : int * int * array -> array
    val retrieve : int * array -> int
    axiom Forall (n,v,w,l) => put(n,v,put(n,w,l)) == put(n,v,l)
    axiom Forall (n,m,v,w,l) =>
            n<>m implies put(n,v,put(m,w,l)) == put(m,w,put(n,v,l))
```

```
  axiom Forall n => retrieve(n,empty) == 0
  axiom Forall (n,v,l) => retrieve(n,put(n,v,l)) == v
  axiom Forall (n,m,v,l) =>
          n<>m implies retrieve(n,put(m,v,l)) == retrieve(n,l)
end
```

Partial implementations. In EML, we can specify `Array` without providing an implementation. We just use a question mark in place of the structure body:

```
structure Array :> ARRAY = ?
```

Or we can provide a *partial* implementation by including some definitions but omitting others, again using a question mark:

```
structure Array :> ARRAY =
  struct
    type array = int list
    val empty = nil
    fun put(n,v,nil) = if n=0 then v::nil else 0::put(n-1,v,nil)
      | put(n,v,w::l) = if n=0 then v::l else w::put(n-1,v,l)
    fun retrieve(n,nil) = 0
      | retrieve(n,v::l) = ?
  end
```

This might represent an intermediate stage of development, where we have decided how to represent arrays and how to implement the empty array and the `put` function, but we haven't yet decided how to implement `retrieve` except that it should return `0` for the empty array.

Although we haven't decided how to implement `retrieve`, we do know what properties it is supposed to satisfy, namely the ones from `ARRAY` that mention it. We can add these to the structure in place of the missing code:

```
structure Array :> ARRAY =
  struct
    type array = int list
    val empty = nil
    fun put(n,v,nil) = if n=0 then v::nil
                       else 0::put(n-1,v,nil)
      | put(n,v,w::l) = if n=0 then v::l
                        else w::put(n-1,v,l)
    fun retrieve(n,nil) = 0
      | retrieve(n,v::l) = ?
    axiom Forall (n,v,l) => retrieve(n,put(n,v,l)) == v
    axiom Forall (n,m,v,l) =>
            n<>m implies retrieve(n,put(m,v,l)) == retrieve(n,l)
  end
```

In fact, the previous version (without the axioms) wouldn't be correct in EML! The requirement is that all the possible ways of filling in the question marks in the body must satisfy the axioms in the signature. Without axioms to constraint the implementation of `retrieve`, there are some implementations that don't behave properly.

Specifying substructures Signatures with axioms can have a hierarchical structure just as ordinary signatures can.

For example, here is a specification of a structure containing functions for creating, updating and displaying a histogram.

```
signature HISTOGRAM =
  sig
    structure A : ARRAY
    type histogram
    val create : histogram
    val incrementcount : int * histogram -> histogram
    val display : histogram -> A.array
    local
      val count : int * histogram -> int
      axiom Forall n => count(n,create) == 0
      axiom Forall (n,h) =>
              count(n,incrementcount(n,h)) == 1 + count(n,h)
      axiom Forall (n,m,h) =>
              n<>m implies count(n,incrementcount(m,h)) == count(n,h)
    in
      axiom Forall (n,h) => A.retrieve(n,display h) == count(n,h)
    end
  end

structure Histogram :> HISTOGRAM = ?
```

This specification uses an auxiliary function (`count`) in order to specify `display`. (It's possible to specify it directly: exercise.) We don't want this function to become part of the signature because there may be implementations that don't involve `count`.

This specification can be implemented by representing histograms as arrays and using the identity function for `display`:

```
structure Histogram :> HISTOGRAM =
  struct
    structure A : ARRAY = Array
    type histogram = A.array
    val create = A.empty
    fun incrementcount(n,h) = A.put(n, 1 + A.retrieve(n,h), h)
    fun display h = h
  end
```

The `display` function produces an array as result. If we want this to be the same kind of arrays as used elsewhere in the system rather than allowing some other implementation of arrays, we can specify this using a sharing constraint:

```
structure Histogram :> HISTOGRAM where type A.array = Array.array = ?
```

Equivalently, we can add the sharing constraint to `HISTOGRAM` by writing

```
structure A : ARRAY sharing type A.array = Array.array
```