Proving that a structure meets its specification

The problem of verifying that a structure without substructures satisfies its specification is just the same as the problem of proving that all the functions it contains meet their specifications. So, for example, proving that `Array` satisfies its specification would be a matter of showing that the functions satisfy the five axioms in `ARRAY`.

If we have a structure with substructures, there are two stages in the proof:

1. We have to show that all the substructures meet their specifications. (Note that the substructures may themselves have substructures.)

2. The functions in the structure must be shown to satisfy their specifications.

The order in which these are done is irrelevant. But concerning (2): since these functions may make use of functions in the substructures, these proofs will often need to use information about the substructures.

An important point is that it is normally most convenient to use the *specifications* of the functions in the substructures in these proofs rather than the *code* of these functions. First, the specification is normally at a more abstract level than the code, defining *what* the function does — which is interesting in such proofs — rather than *how* it works — which is not. Second, this allows a different substructure (satisfying the same specification but possibly using a different data representation or different algorithms) to be substituted without affecting the correctness of the proof. Third, this is essential if opaque signature ascription is used: clients can only make use of what is written in the signature, not the extra information that is in the code, and that fact needs to be reflected in proofs.

Recall the `Histogram` example:

```
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 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
```

Suppose that we have already shown that `Histogram.A` (i.e. `Array`) satisfies `ARRAY`. To prove that `Histogram` satisfies `HISTOGRAM`, we then have to show that the functions and constants in `Histogram` (`create`, `incrementcount` and `display`, defined by the code in the body of `Histogram`), satisfy the axiom in `HISTOGRAM`:

```
Forall (n,h) => A.retrieve(n,display h) == count(n,h)
```

where `count` is defined by the "hidden" axioms in `HISTOGRAM`, namely:

```
Forall n => count(n,create) == 0
Forall (n,h) => count(n,incrementcount(n,h)) == 1 + count(n,h)
Forall (n,m,h) => n<>m implies count(n,incrementcount(m,h)) == count(n,h)
```

and `A` satisfies `ARRAY`.

Specifying functors Axioms in signatures can also be used to specify functors. The only difference is that a functor has both a parameter signature and a result signature, and so each functor is associated with two specifications.

The specification in Practical 3 of a sorting functor (with the gap there filled in) is an example:

```
signature PO =
  sig
    type t
    val le : t * t -> bool
    axiom Forall x => le(x,x)
    axiom Forall (x,y,z) => le(x,y) andalso le(y,z) implies le(x,z)
    axiom Forall (x,y) => le(x,y) andalso le(y,x) implies x==y
    axiom Forall (x,y) => le(x,y) orelse le(y,x)
  end

signature SORT =
  sig
    structure OBJ : PO
    local
      val count : 'a * 'a list -> int
      axiom Forall a => count(a,nil) = 0
      axiom Forall (a,l) => count(a,a::l) = 1+count(a,l)
      axiom Forall (a,b,l) => a=/=b implies count(a,b::l) = count(a,l)
      val permutation : 'a list * 'a list -> bool
      axiom Forall (l,l') =>
        permutation(l,l') = (Forall x => count(x,l) = count(x,l'))
      val ordered : OBJ.t list -> bool
      axiom Forall l =>
        ordered l iff
        (Forall (a,b,c,x,y) => l==a@[x]@b@[y]@c implies OBJ.le(x,y))
    in
```

```
    val sort : OBJ.t list -> OBJ.t list
    axiom Forall l => permutation(l,sort l)
    axiom Forall l => ordered(sort l)
  end
end

functor Sort(X : PO) :> SORT where type OBJ.t=X.t = ?
```

The axioms in `PO` require that the function `le` satisfies the properties of a total order relation. (So `PO` isn't a very good name – oops!) A sorting program wouldn't be expected to work for arbitrary choices of `le`.

The axioms in `SORT` specify the properties we want `sort` to satisfy, under the assumption that `le` satisfies the properties in `PO`. The most abstract way to specify `sort` is to require that the output is a *permutation* of the input, and that the output is *ordered* according to `OBJ.le`. Thus we specify `permutation` and `ordered` as auxiliary functions. An easy way to specify `permutation` is using a function to count the number of occurrences of an element in a list, which is the purpose of the function `count`.

One way of implementing `Sort`, using heapsort, is given in Practical 3. Another way is using the *quicksort* algorithm. Here we choose an element of the list, partition the remaining elements in the list into those that are less than or equal to that element and those that are greater than it, sort both lists, and then append the results. This requires an auxiliary function to do the partitioning step.

As before, we can provide an executable body, or first give a body that is partly executable. For instance, we might supply code for `sort` but only *specify* `partition`:

```
functor Sort(X : PO) :> SORT where type OBJ.t=X.t =
  struct
    structure OBJ = X
    val partition : OBJ.t * OBJ.t list -> (OBJ.t list * OBJ.t list) = ?
    axiom Forall (a,l) =>
        let val (l1,l2) = partition(a,l)
        in  Forall b => member(b,l1) implies OBJ.le(b,a)
            andalso
            Forall c => member(c,l2) implies not(OBJ.le(c,a))
            andalso
            Forall d => member(d,l) iff member(d,l1) orelse member(d,l2)
            end
    fun sort nil = nil
      | sort(a::l) = let val (l1,l2) = partition(a,l)
                     in  (sort l1)@(a::(sort l2))
                     end
  end
```

We take the first element in the list as the one to partition by. The axiom for `partition` assumes that `member` has been defined earlier. A complete executable version of the functor is:

```
functor Sort(X : PO) :> SORT where type OBJ.t=X.t =
  struct
    structure OBJ = X
    fun partition(a,nil) = (nil,nil)
      | partition(a,b::l) =
          let val (l1,l2) = partition(a,l)
```

```
        in  if OBJ.le(b,a) then (b::l1,l2) else (l1,b::l2)
        end
  fun sort nil = nil
    | sort(a::l) = let val (l1,l2) = partition(a,l)
                   in  (sort l1)@(a::(sort l2))
                   end
end
```

Proving that a functor meets its specification Proving that a functor meets its specification is like showing that a structure with substructures satisfies its specification: the functor parameter may be regarded in the same way as substructures during the proof. In this case the argument for using the *specifications* of substructures in the proof, rather than the code, gains added force: we do not know ahead of time which structures a functor will be applied to, and so we have no access to the code!

So, to prove correctness of `Sort`, we have to prove the axioms specifying `sort` in `SORT`, namely:

```
Forall l => permutation(l,sort l)
Forall l => ordered(sort l)
```

from the code in the body of `Sort`, under the assumption that `X` satisfies `PO` and where `permutation`, `ordered` and `count` are defined by the "hidden" axioms in `SORT`, namely:

```
Forall a => count(a,nil) = 0
Forall (a,l) => count(a,a::l) = 1+count(a,l)
Forall (a,b,l) => a=/=b implies count(a,b::l) = count(a,l)
Forall (l,l') => permutation(l,l') = (Forall x => count(x,l) = count(x,l'))
Forall l => ordered l iff
               (Forall (a,b,c,x,y) => l==a@[x]@b@[y]@c implies OBJ.le(x,y))
```