# Formal specification of ML programs

Donald Sannella

Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh

# 1 Introduction

Specifications play a part in every phase of program development. First, the construction of a program cannot commence without a specification of what it is supposed to do. This *requirements specification* is supplied by the customer for whom the program is being developed. It serves as a *means of communication* between the customer and the program development team.

Specifications also serve as a means of communication between the members of the program development team. Each programmer is responsible for a certain component of the program which may use facilities provided by several "foreign" components. Precise specifications of these components are required before any program which relies on them can be written. These specifications are produced during the design phase when a way of decomposing the task is decided upon and the component subtasks recorded. It is important that the specifications of the components avoid giving away unnecessary details of the implementation — if nobody is able to depend on the idiosyncratic features of a particular solution to a subtask, then another solution may be easily substituted without affecting correctness. In this way, specifications are also a means of *avoiding* undesirable communication, by defining exactly those details of module interfaces on which others are allowed to depend.

Once a program has been written some attempt is normally made to check that it is correct. This check may be an informal test of the program on a few input values, or a formal proof of correctness. In any case, some specification is needed to compare the program against; a program is only correct with respect to some specification of its desired behaviour. Finally, documentation is required, both for the use of the customer and to aid the future maintenance and modification of the program. This documentation is also a specification, serving as a means of communication between the development team and the users and future maintainers of the program.

Up to now the word "specification" has been used in a very broad sense. Every means of describing a program and its behaviour is included, from informal English documentation or program comments to a precise description in a formal specification language. A very simple and straightforward program may be useful as a specification of an equivalent program which must be complex in order to be efficient.

Informal specifications suffer from imprecision. This is a serious problem because of the very heavy penalties which may be incurred if a specification is misunderstood. This is one reason why we advocate the use of *formal* specifications in the program development process. Formal specifications are necessarily precise and unambiguous.

A reason for using formal specifications which is becoming increasingly important is that they enable the use of formal methods in program development. A program can (in principle) be proved to satisfy its formal specification. Perhaps a more reasonable approach is to ensure that a correct program is produced in the first place using formal methods for evolving guaranteed-correct programs from specifications. It cannot be denied that neither of these is currently a possibility for practical development of real programs. But the use of these and other formal methods in the program development process presupposes the use of formal specifications.

The effort of constructing a formal specification often has a large payoff in increased understanding of the task and early detection of difficulties. The careful attention to detail which is required to write a formal specification means that important problems and useful generalisations can be uncovered early in the development process. This is the case even if no formal methods are to be used in the construction of the program itself.

The main problem with formal specifications is that they are hard to construct and hard to use and understand. It was already mentioned that the fact that it is hard to write formal specifications may be a blessing in disguise. Various methods are available for building large specifications in a structured fashion from small, easy to understand and reusable components. But more simply and obviously, formal specifications should be supplemented with informal comments to ease their understanding and use. Ultimately, the advantages of using formal specifications to develop correct programs will outweigh the difficulties involved in their use.

These notes describe methods for specifying Standard ML programs. At the moment, there are no tools available for processing ML specifications, so although they will be written in a font which makes them look like programs, we have no way even to check that they are syntactically well-formed or that they are free from type errors. Such tools would keep us from writing specifications which are meaningless. Other tools are prerequisites to the practical use of specifications; the most important is probably some kind of mechanised theorem prover which allows logical consequences to be inferred from specifications. Finally, the task of constructing specifications is eased if we have available a large library of commonly-used specifications (for example, of standard data types like sets, stacks and queues and standard functions like sorting and searching) so that most of the effort can be devoted to those aspects which are unique to the problem at hand.

# 2 Specifying ML functions

Before we begin to discuss methods for specifying ML functions, note that ML function definitions such as the following are specifications already:

```
fun member(x,nil) = false
  | member(x,y::l) = if x = y then true else member(x,l)
```

This is a specification in the sense that all programs are specifications — namely, it is a precise and unambiguous definition. But more to the point, it is a high-level description of the `member` function, uncluttered by low-level details in comparison with the same program in a more traditional language like Pascal. Using a simple notation, it specifies the result of the `member` function by means of a case analysis. This is arguably the simplest and most straightforward specification of the `member` function.

This example suggests that ML could itself be useful as a specification language for programs in a language like Pascal. ML function definitions could be used to specify Pascal functions in the obvious way, and also to specify procedures by describing the value of variables on exit as a function of the values of variables on entrance. An advantage of using ML as a specification language is that specifications are runnable (or rather, walkable[1]) so a specification can also be used as a prototype implementation of the system being developed.

Each ML function definition consists of a collection of equations having a certain special form — namely, the left-hand side of each equation is the name of the function being defined, applied to a *pattern*. This is what makes it possible to execute functions in ML; function evaluation works by matching the patterns supplied against the given argument, returning the value of the right-hand side of the matching equation. Now suppose this restriction were to be relaxed and functions could be defined in ML by means of arbitrary equations? (Or slightly more generally, not just functions but arbitrary ML values?)

As an example, consider the problem of specifying the square root of a (real) number. To write an ML program, we would have to code some algorithm for finding the square root, such as Newton's method. But if we are allowed to use unrestricted equations, the specification is short and sweet:

```
sqrt(a)*sqrt(a) = a
```

[1]Joke due to Rod Burstall.

Since this equation is no longer in the special form required by ML, `sqrt` is not runnable (or even walkable), but only *thinkable*. But as a high-level specification, this is preferable to a program which uses Newton's method to compute the square root — it says *what* we want without being cluttered by the least suggestion of *how* to compute it.

As another example, suppose we have already defined a data type of matrices as well as matrix multiplication (`x`) and the identity matrix (`I`). We can now specify a function `inv:matrix -> matrix` to invert a matrix as follows:

```
inv(A) x A = I
A x inv(A) = I
```

Once we have dropped the restriction that equations must be in the form required by ML, there is already no particular reason to require that specifications be expressed using equations only. It is sometimes convenient to use other logical notations to specify programs, possibly mixed with equations. For example, here is a specification of a function `maxelem:int list -> int` which finds the largest element in a list of integers. This specification refers to the function `member` defined earlier.

```
l <> nil ==> member(maxelem l,l)
member(a,l) ==> (maxelem l) >= a
```

This says that the maximum element of a non-empty list is an element of the list (line 1) and moreover it is greater than or equal to all the elements in the list (line 2). Both assertions (or *axioms*) use `==>` to denote *implication*. For example, the second axiom should be read

"`member(a,l)` implies `(maxelem l) >= a`"

or

"if `member(a,l)` then `(maxelem) l) >= a`"

The things on the left- and right-hand side of the `==>` should be equations or else (as above) `bool`-valued expressions which can really be regarded as abbreviating equations of the form *expr*`=true`. We will use these two forms interchangeably. (In fact, the second axiom above is equivalent to the equation:

```
(if member(a,l) then (maxelem l) >= a else true) = true
```

but this is a bit more cryptic.)

Other logical notation which will come in handy when writing specifications are the connectives `and`, `or` and `not` and the quantifiers `forall` and `exists`. All axioms are already surrounded by an implicit `forall` quantifier over the unbound variables in the axiom; for example the axiom

```
member(a,l) ==> (maxelem l) >= a
```

is equivalent to the axiom

```
forall a,l => member(a,l) ==> (maxelem l) >= a
```

(the `=>` here is the same one used with `fn` to write unnamed functions, which is not the same as `==>`). In fact, the ML function definition

```
fun member(x,nil) = false
  | member(x,y::l) = if x = y then true else member(x,l)
```

is equivalent to the two axioms

```
axiom forall x => member(x,nil) = false
axiom forall x,y,l => member(x,y::l) = if x = y then true
                                       else member(x,l)
```

Two more of these notations are used in the following specification of the (built-in) function `>=` in terms of `+` and a function `nonneg`[2] which returns `true` unless its argument is a negative number:

```
n >= m = exists r => (n = m + r and (nonneg r))
```

The right-hand side of this equation should be read "there exists some `r` such that `n = m + r` and `nonneg r` is true". Again, the two arguments of `and` and the argument of `exists` after the `=>` should be either equations or `bool`-valued expressions. Note that although the above specification of `>=` is superficially an equation in the form required by ML, it is not runnable because `exists` is not runnable. But note that runnable functions like `+` may also be used in the specification.

As a slightly more difficult example, we now specify two functions,

```
before: string * string -> string
```

and

```
after: string * string -> string
```

Given two strings `s` and `r`, `before(s,r)` is the part of `r` before the first occurrence of `s` in `r` and `after(s,r)` is the part of `r` after the first occurrence of `s` in `r`. So for example,

[2] `nonneg` is not a built-in function.

`before("my","Elementary, my dear Watson")` is `"Elementary, "`

and

`after("my","Elementary, my dear Watson")` is `" dear Watson"`.

The specification of `before` and `after` will make use of an auxiliary function `initial_substring:string*string->bool`. Given two strings `s` and `r`, `initial_substring(s,r)` is `true` if the first part of `r` matches `s`, and `false` otherwise. So for example,

`initial_substring("my","my dear Watson")` is `true`

while

`initial_substring("dear","my dear Watson")` is `false`.

The specification of `initial_substring` is as follows:

```
initial_substring(s,r) = exists t => (s^t = r)
```

That is, `s` is an initial substring of `r` if there is a string (possibly empty) which can be added to the end of `s` so that the result is `r`.

Using `initial_substring` we can write a simple and elegant specification of `before` and `after`:

```
before(s,r)^s^after(s,r) = r
t1^s^t2 = r ==> initial_substring(before(s,r),t1)
```

The first axiom of this specification states that any string `r` (containing at least one occurrence of `s`) consists of the part of `r` before `s`, followed by `s` itself, followed by the part of `r` after `s`. The second axiom says that `before` and `after` are with respect to the *first* occurrence of the first argument in the second argument, since if there is another way of decomposing `r` into three parts `t1`, `s` and `t2` then `before(s,r)` must be an initial substring of `t1`. Note that this specification requires `before(s,r)` and/or `after(s,r)` to produce no result in the case where `s` does not occur in `r`; although it would be possible to specify that they produce some particular result in this case as well, the specification may also be adequate for some purposes as it stands.

**Exercise** Modify the above specification so that `before(s,r)` and `after(s,r)` return `nil` if `s` does not occur in `r`.

Note that the specifications of `before` and `after` above are completely intertwined; in contrast to the earlier specifications there is no single axiom or collection of axioms which are entirely devoted to specifying either `before` or `after`. Even though the second axiom contains no explicit use of `after`, it constrains the implementation of `after` because of the way that `after` and `before` are related by the first axiom.

One of the advantages of using arbitrary equations (mixed with logical notation or not) is that it is possible to write definitions which are purposefully vague; that is, we are not required to specify the value of the function being specified exactly under all circumstances but we can instead leave decisions open to be made later. For example, the alert reader will have noticed that the result of applying `maxelem` to `nil` is undefined and that our specification of `sqrt` does not say whether we want the positive or negative square root. Specifications such as these which leave some things unspecified are called *loose* specifications.

A loose specifications is neither imprecise nor ambiguous; it specifies precisely those aspects of the program which we are interested in while leaving some choices open to be made at later stages of the design process or by the programmer. For example, we might want to specify a square root function which is required to produce a result which is correct to within a certain precision (say 1%). We can specify this as follows:

```
sqrt(a)*sqrt(a) >= 0.99*a
1.01*a >= sqrt(a)*sqrt(a)
```

Any algorithm for producing the square root of a number will be acceptable according to this specification provided that it works with at least the specified precision. When a system involving this function is implemented, the programmer or designer may decide to use a simple algorithm which produces answers correct to within 1% rather than a more complex algorithm which produces more accurate results on the basis of mundane considerations like the amount of storage and time required by the two algorithms.

# 3 Proving that a function meets its specification

Suppose that we have written an ML function and we wish to ensure that it satisfies its specification. This is the problem of *program verification.* We must prove that the ML function we have defined satisfies each of the axioms in the specification.

To take a concrete example, let us recall the specification of the function `maxelem : int list -> int` which finds the largest integer in a list:

```
l<>nil ==> member(maxelem l,l)
member(a,l) ==> (maxelem l) >= a
```

An ML function which satisfies this specification (at least, we would like to show it does) is the following:

```
fun maxelem(a::nil) = a
  | maxelem(a::b::l) = if a>maxelem(b::l) then a
                       else maxelem(b::l)
```

Since the definition of `maxelem` is recursive, we will use *induction* to show that `maxelem` satisfies each of the axioms in the above specification. The specification makes use of other functions, namely `<>`, `member` and `>=`. A rigorous proof would make reference to the definitions of these functions but it will simplify matters slightly if we allow ourselves to use various facts about these functions without proving that they follow from the definitions; for example, we will need to use the fact that if `a>b` and `b>=c` then `a>=c`.

**Proof** (`maxelem` satisfies `l<>nil ==> member(maxelem l,l)`) We assume `l<>nil` and prove by induction that `member(maxelem l,l) = true`.

**Base case** Suppose `l = a::nil` for some integer `a`. Then `maxelem l = a`, and so `member(maxelem l,l) = member(a,a::nil) = true`.

**Step case** We assume `member(maxelem l,l) = true` and show that then `member(maxelem(a::l),a::l) = true` for any integer `a`. According to the definition of `maxelem`, `maxelem(a::l)` is either `a` or `maxelem l`. If it is `a`, then `member(maxelem(a::l),a::l) = member(a,a::l) = true`. If it is `maxelem l`, then `member(maxelem(a::l),a::l) = member(maxelem l,a::l) = true` because of our assumption that `member(maxelem l,l) = true`. □

**Proof** (`maxelem` satisfies `member(a,l) ==> (maxelem l) >= a`) We assume that `member(a,l)` and prove by induction that `(maxelem l) >= a`.

**Base case** Suppose that `l = a::nil`; then `maxelem l = a` and so we have `maxelem l = a >= a`.

**Step case** We assume that `(maxelem l) >= a` for every integer `a` such that `member(a,l)` and show that then `maxelem(b::l) >= a` for every `a` such that `member(a,b::l)`, for every integer `b`. According to the definition of `maxelem`, there are two cases to consider:

**Case 1** (`b > maxelem l`) In this case, `maxelem(b::l) = b`. For every `a` such that `member(a,b::l)`, either `a = b` (and so `maxelem(b::l) = b = a >= a`) or `member(a,l)` (and so `maxelem(b::l) = b > maxelem l >= a`).

**Case 2** (`maxelem l >= b`) In this case, `maxelem(b::l) = maxelem l`. For every `a` such that `member(a,b::l)`, either `a = b` (and so we have `maxelem(b::l) = maxelem l >= b = a`) or `member(a,l)` (in which case we get `maxelem(b::l) = maxelem l >= a`). □

We have thus proved the correctness of our definition of `maxelem`. The proof was rather tedious and would have been much longer and even more tedious had we attempted to give a rigorous proof directly from the definitions of `<>`, `member`, `>=` and `maxelem`. It is easy to make mistakes in such proofs when they are done by hand, especially when they involve even slightly more complicated programs (and also since the person doing the proof does not expect to find bugs!).

Considerations such as these have prompted research into computer-assisted program verification systems, or more generally into computer-assisted theorem-proving systems. It is not within the scope of these notes to discuss this topic here, except to suggest that such a system would provide a great deal of help in performing proofs like those above; indeed, the Boyer-Moore theorem prover[3] would probably be able to carry out the above proof entirely automatically.

**Exercise** Write ML programs to compute `>=`, `initial_substring`, `before` and `after` and prove that they satisfy their specifications.

**More difficult exercise** Write ML programs to compute `sqrt` and `inv` and prove that they satisfy their specifications (for `sqrt`, use the specification at the end of the last section).

# 4 Specifying structures and functors

Just as we used axioms to specify functions, we can use axioms to specify structures. The only difference is that since a structure may contain several functions, the specification of a structure will be larger than the specification of a single function.

For example, consider the following structure which implements an array of integers indexed starting from 0 using a list of integers:

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

[3] R.S. Boyer and J.S. Moore, *A Computational Logic*, Academic Press, 1979.

```
                        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)
    end;
```

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

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

These axioms state properties of `Array.retrieve`, `Array.put` and `Array.empty` such as the fact that inserting a value using `Array.put` at the same place as an earlier insertion supercedes the value inserted earlier (axiom 1) and that when using `Array.retrieve` to obtain a value from a given place in the array, insertions at other places in the array have no effect (axiom 5).

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:

```
sig
  type array
  val empty: array
  val retrieve: int * array -> int
  val put: int * int * array -> array
end;
```

As a description of what `Array` makes available, this signature is sufficient for the purpose of compiling functions which refer to `Array`, but otherwise it is not very informative. It is not sufficient, for example, for proving program correctness or for program documentation. It is natural to combine the information in the signature with the axioms specifying `Array` to form a more complete interface, as follows:

```
signature ARRAYSIG =
  sig
    type array
    val empty: array
    val retrieve: int * array -> int
    val put: int * int * array -> array
    axiom put(n,v,put(n,w,l)) = put(n,v,l)
```

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

Now, if we want to express the fact that `ARRAYSIG` is the interface of `Array` we simply write `Array:ARRAYSIG`. As usual, this can be combined with the declaration of `Array` as follows:

```
structure Array:ARRAYSIG=
  struct
    type array = int list;
    . . .
  end;
```

By adding axioms to a signature as in `ARRAYSIG`, we have formed what is known as a *theory*, and it would be appropriate to change the notation accordingly. However, we will continue to use the term "signature" to refer to a signature with axioms as well. One could imagine extending the ML compiler to allow signatures to include axioms, but unfortunately the compiler cannot be expected to check that an axiom is satisfied by a function the way that it can check that types are correct (for this would involve proofs like the one in the last section), so axioms would have to be treated as comments.

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 (recall that a histogram is a statistical device for maintaining a count of the number of data elements encountered according to their values, typically displayed as a "bar graph" — for example, a graph of scores on an examination vs. the number of students obtaining those scores):

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

```
    axiom A.retrieve(n,display h) = count(n,h)
    end
  end;
```

This specification uses an auxiliary function `count` in order to specify the function `display`[4]. We do not want this to become part of the signature `HISTOGRAMSIG` because there may be implementations of `HISTOGRAMSIG` which do not involve a function like `count`. In the specification above we adopted the syntax `local` ... `in` ... `end` used in structures and other ML declarations to express that we want `count` to be local to the specification of `display` (this is technically a change to the syntax of signatures). This is a bit clumsy and difficult to read, so from now on we will use a pictorial representation whereby the specification of types and functions which are intended to be strictly local to the specification of the other types and functions in the signature are enclosed in a box like so:

```
signature HISTOGRAMSIG =
  sig
    structure A:ARRAYSIG
    type histogram
    val create:  histogram
    val incrementcount:  int * histogram -> histogram
    val display:  histogram -> A.array
    val count:  int * histogram -> int
    axiom count(n,create) = 0
    axiom count(n,incrementcount(n,h)) = 1 + count(n,h)
    axiom n<>m ==> count(n,incrementcount(m,h)) = count(n,h)
    axiom A.retrieve(n,display h) = count(n,h)
  end;
```

This specification is implemented by the following structure in which histograms are represented as arrays and `display` is the identity function:

```
structure Histogram:HISTOGRAMSIG =
  struct
    structure A:ARRAYSIG = 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;
```

[4]It is also possible to specify `display` directly without using `count` (exercise).

Axioms in signatures can also be used to specify functors. The only difference is that a functor has both a parameter signature (actually, zero or more parameter signatures, depending on how many parameters it takes) and a result signature, and so each functor is associated with two specifications (more or less, depending on the number of parameters).

As an example we will consider a functor which provides a function for sorting a list of objects using a Quicksort algorithm, given an ordering on the objects as a parameter. The signature of the parameter (without axioms) will be as follows:

```
signature ORDSIG =
  sig
    type obj
    val le: obj * obj -> bool
  end;
```

The idea is that `le` will be an order relation (*l*ess than or *e*qual) on values of type `obj`. The signature of the result (again, without axioms) will be:

```
signature SORTSIG =
  sig
    structure OBJ:ORDSIG
    val partition: OBJ.obj * OBJ.obj list
                               -> OBJ.obj list * OBJ.obj list
    val sort: OBJ.obj list -> OBJ.obj list
  end;
```

The `partition` function is used in the Quicksort algorithm to split a list into two lists relative to a given element; the first list contains all objects in the given list which are less than or equal to the element and the second contains the objects which are greater than the element.

Finally, here is the definition of the functor itself:

```
functor Sort(X:ORDSIG):SORTSIG =
  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;
```

Now we can consider what the specification of `Sort` should be. The first thing to do is to augment the parameter signature `ORDSIG` by axioms stating the properties required of any structure used as a parameter of `Sort`. It turns out that the sorting program above will not give the expected results unless the `le` function is a *total order*[5], that is:

- `le` is *transitive*, i.e. for any `a,b,c:obj` if `le(a,b)` and `le(b,c)` then `le(a,c)`;
- `le` is *anti-symmetric*, i.e. for any `a,b:obj` if `le(a,b)` and `le(b,a)` then `a=b`; and
- `le` is *total*, i.e. for any `a,b:obj` either `le(a,b)` or `le(b,a)` (or both).

Examples of total orders are `>=` and `<=` on integers and the dictionary ordering on strings. If we add these requirements in the form of axioms to `ORDSIG`, we obtain the following parameter specification (or import interface) for `Sort`:

```
signature ORDSIG =
  sig
    type obj
    val le: obj * obj -> bool
    axiom le(a,b) and le(b,c) ==> le(a,c)
    axiom le(a,b) and le(b,a) ==> a=b
    axiom le(a,b) or le(b,a)
  end;
```

Now we have to augment the result signature `SORTSIG` by axioms specifying `partition` and `sort`. The function `sort` is a bit tricky to specify; the idea of the specification below is that `sort` takes a list and permutes (rearranges) it in such a way that the result is an ordered list. This specification of `sort` requires several auxiliary functions:

```
signature SORTSIG =
  sig
    structure OBJ:ORDSIG
    val partition:  OBJ.obj * OBJ.obj list
                                -> OBJ.obj list * OBJ.obj list
    axiom (l1,l2)=partition(a,l) and member(b,l)
                              ==> member(b,l1) or member(b,l2)
    axiom (l1,l2)=partition(a,l) and member(b,l1) ==> OBJ.le(b,a)
```

[5]The reader may disagree with this statement depending on the interpretation of "expected results"; at least it should be clear that a total order is sufficient.

```
    axiom (l1,l2)=partition(a,l) and member(b,l2) ==> not(OBJ.le(b,a))
    val isordered:  OBJ.obj list -> bool
    axiom isordered l = forall l1,a,l2,b,l3 =>
                             (l=l1@[a]@l2@[b]@l3 ==> OBJ.le(a,b))
    val remove:  OBJ.obj * OBJ.obj list -> OBJ.obj list
    axiom remove(a,a::l) = l
    axiom a<>b ==> remove(a,b::l) = b::remove(a,l)
    val ispermutation:  OBJ.obj list * OBJ.obj list -> bool
    axiom ispermutation(nil,nil) = true
    axiom ispermutation(nil,b::l) = false
    axiom ispermutation(a::l1,l2) = member(a,l2) and
                                    ispermutation(l1,remove(a,l2))
    val sort:  OBJ.obj list -> OBJ.obj list
    axiom sort(l1) = l2 ==> isordered l2 and ispermutation(l1,l2)
  end;
```

Notice that the function `remove` is local to the specification of `ispermutation` and so the above specification could have used two concentric boxes.

Now that we have written the parameter and result specification of `Sort`, the fact that they are the import and export interfaces is expressed by defining `Sort` with explicit parameter and result signatures (as before):

```
functor Sort(X:ORDSIG):SORTSIG =
  struct
    . . .
  end;
```

# 5 Proving that structures and functors meet their specifications

The problem of verifying that a "flat" structure (one without substructures) satisfies its specification is just the same as the problem of proving that all the functions it contains meet their specifications.

**Exercise** Prove that the structure `Array` in the last section meets its specification.

The problem of showing that a structure with substructures meets its specifications has two stages. First, it is necessary to show that all the substructures meet their specifications. Of course, the substructures may themselves have substructures, so in general this requires descending to the most deeply nested substructures and working upwards. Second, the functions in the structure must be shown to satisfy their specifications. Since these functions may make use of functions in substructures, these proofs will often need to use facts about the functions in the substructures. 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. For one thing, 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). For another, 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.

As an example, consider the structure `Histogram` from the last section. For convenient reference, here is the structure and its specification `HISTOGRAMSIG` again:

```
signature HISTOGRAMSIG =
  sig
    structure A:ARRAYSIG
    type histogram
    val create:  histogram
    val incrementcount:  int * histogram -> histogram
    val display:  histogram -> A.array
    val count:  int * histogram -> int
    axiom count(n,create) = 0
    axiom count(n,incrementcount(n,h)) = 1 + count(n,h)
    axiom n<>m ==> count(n,incrementcount(m,h)) = count(n,h)
    axiom A.retrieve(n,display h) = count(n,h)
  end;

structure Histogram:HISTOGRAMSIG =
  struct
    structure A:ARRAYSIG = 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;
```

where

```
signature ARRAYSIG =
  sig
    type array
    val empty: array
```

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

is the specification of the structure `Histogram.A`.

Assuming that you have done the above exercise, we know that `Histogram.A` satisfies the specification `ARRAYSIG`. In order to prove that `Histogram` satisfies `HISTOGRAMSIG`, we then have to show that the functions and constants in `Histogram`, namely `create`, `incrementcount` and `display`, satisfy the axiom in `HISTOGRAMSIG`, namely:

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

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

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

**Proof** (`Histogram` **satisfies** `A.retrieve(n,display h) = count(n,h)`) The proof is by induction on the structure of histograms.

**Base case** Suppose `h = create`. Then:

`A.retrieve(n, display h)`
  `= A.retrieve(n,A.empty)` (by the definitions of `display` and `create` in `Histogram`)
  `= 0` (by the third axiom in `ARRAYSIG`)
  `= count(n,h)` (by the first axiom for `count`).

**Step case** We assume `A.retrieve(n,display h) = count(n,h)` and show that then

```
A.retrieve(n,display(incrementcount(m,h)))
                        = count(n,incrementcount(m,h))
```

for any integer `m`. According to the definitions of `display` and `incrementcount` in `Histogram`,

```
A.retrieve(n,display(incrementcount(m,h))) =
              A.retrieve(n,A.put(m,1 + A.retrieve(m,h),h))
```

Now in order to complete the proof we must consider two cases:

**Case 1** (`n = m`) In this case,

`A.retrieve(n,A.put(m,1 + A.retrieve(m,h),h))`

`= 1 + A.retrieve(n,h)` (by the fourth axiom in `ARRAYSIG`)

`= 1 + count(n,h)` (by the definition of `display` and the inductive assumption)

`= count(n,incrementcount(m,h))` (by the second axiom for `count`).

**Case 2** (`n <> m`) In this case,

`A.retrieve(n,A.put(m,1 + A.retrieve(m,h),h))`

`= A.retrieve(n,h)` (by the fifth axiom in `ARRAYSIG`)

`= count(n,h)` (by the definition of `display` and the inductive assumption)

`= count(n,incrementcount(m,h))` (by the third axiom for `count`).□

Proving that a functor meets its specification is very similar to showing that a structure with substructures satisfies its specification, since the functor parameters may be regarded more or less 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.

**Exercise** Show that the functor `Sort` in the last section satisfies its specification.

# 6 Conclusion

A number of important topics in the specification of ML programs have not been covered. For example, the examples do not make use of exceptions or assignment — we have been dealing with purely functional programs only. The use of exceptions and assignment turn out to introduce complications which have not yet been resolved in a satisfactory way. Another point which is important to be aware of is that we have been taking a slightly idealised view of the world where for example integers can be of unbounded size and real numbers are of unbounded precision. Since we have glossed over the sordid realities of the situation, it is actually possible to prove that a program satisfies a specification when in fact it does not (because e.g. of arithmetic overflow). Capturing the consequences of the finiteness of real machines in specifications without making them overly complicated is a difficult job.

The use of specifications in the program development process has not been mentioned except in the introduction. The gradual evolution of programs from specifications by means of verified refinement steps so that a correct result is guaranteed is perhaps the most exciting potential application of formal specifications.

For more about this and about the specification of ML programs, see: “Program specification and development in Standard ML” by D. Sannella and A. Tarlecki in Proceedings ACM Conference on Principles of Programming Languages, New Orleans, 1985.

**Acknowledgements** Some of the examples used are originally due to Rod Burstall. Thanks to Andrzej Tarlecki for continuing collaboration on the subject of these notes, to Joan Ratcliff for typing and to Paul Taylor, George Cleland and David Walker for helping with the proofreading and production.