LFCS

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

Modules and Persistence in Standard ML

by

Robert Harper

LFCS Report Series ECS-LFCS-86-11

September 1986

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Modules and Persistence in Standard ML

Robert Harper
Computer Science Department
University of Edinburgh
Edinburgh EH9 3JZ
Great Britain

1 Introduction

Standard ML [Mil 85] is an interactive, statically–typed programming language that provides support for higher–order functions, user–defined abstract types, and type–safe exceptions. The ML modules facility [Mac 85] provides the means for organizing ML programs into units, called *structures*. Each structure has a well–defined interface, called its *signature*, that plays a role similar to that of types in the core language. Viewed statically, a program is a hierarchical arrangement of interdependent structures. Support for the dynamics of program construction is provided primarily by *functors*, functions from structures to structures, that provide the means of glueing structures together to form coherent units. By "coherent" we mean that the individual structures comprising the program are combined so as to share information in the intended way.

A distinctive characteristic of the ML modules system is the way in which this sharing between structures is managed. We use the word "sharing" to refer to any of several ways that two structures can come to depend on one another. There are, in essence, two ways in which dependency can arise. If a structure defines a type that is used by another structure in an essential way (say, by defining a function over that type), then the second structure can only be used in an environment in which the first is present as well. A similar form of dependency arises when one structure allocates a variable or defines an exception that is used by a function in another structure. In this case the two usually must be treated as a unit, for if the first structure is recompiled, it generates a *new* location on the heap (or new exception) that cannot be accessed by the second. MacQueen's central observation is that both forms of dependency can be reduced to managing the sharing of common substructures. In particular, functors which are used to build up hierarchies of structures may require, as part of their parameter specification, that two structures be built from the same structure, thereby ensuring that the combination is properly coherent.

These mechanisms appear to form an adequate basis for addressing the theoretical problems that arise in a modular programming environment. In this paper we address some of the more pragmatic issues involved, such as separate compilation, preservation of session context, and program libraries. These issues are traditionally taken to lie outside of the boundaries of the programming language itself, but this seems to be inappropriate for an interactive langauge like ML. Furthermore, recent work [Atk 81,Atk 85,Alb 85] has shown that considerable benefits accrue from integrating these "environmental" issues into the language proper. The key idea that we shall adapt from this work is the idea of a *persistent heap*, whereby the program heap is considered to be a permanent, rather than transient, repository of objects used by a program. Objects are accessed by a generalized address, call a *persistent identifier*, or *PID*.

Facilities for separate compilation and so forth can be naturally viewed as special cases of persistence. Our claim is that although a persistent heap is adequate for our purposes, it does not appear to be necessary. In particular, we show that the sorts of facilities that are most commonly used in a program development environment can be implemented by techniques that, while not as general as a full persistent heap, are considerably simpler. At the same time, however, we demonstrate that viewing these problems from the point of view of persistence is both natural and fruitful.

2 The ML module facility

We begin with an overview of the ML module facility [Mac 85]. Readers familiar with [Mac 85] or [Har 86b] may safely skip to Section 3.

A structure is an encapsulated environment. The basic form of structure expression consists of a declaration between the brackets `struct` and **end**. The declaration may define new types, exceptions, variables, or structures. A structure declared within another structure is called a *substructure* of that structure. Substructures are used to express hierarchical dependency relationships among structures, and are an essential aspect of ML's approach to sharing. The declaration embodying a structure may contain non–local references only to previously–defined structures. This ensures that a program consists *only* of a collection of structures: no stray top–level declarations that are not packaged into a structure are allowed to participate in the construction of a program unit.

Here are some examples of structure declarations:

```
    structure R =
    struct
      datatype t = ...
      val f : t -> t = ...
    end

  structure S =
    struct
      structure R = R
      datatype 'a tree = leaf of 'a | node of 'a tree * 'a tree
      exception nonexistent : R.t
      fun search(t:'a tree,x:'a) = ...
    end
```

The keyword structure introduces structure bindings, much as val introduces a value binding. The structure bound to R defines a new type t and a function f on that type. The structure bound to S imports R as a substructure, and defines a new type 'a tree, an exception nonexistent, and a function search. Notice that the type t defined in the substructure R of S is accessed by a *qualified name*, consisting of a structure name and a component name, separated by a dot.

Hierarchical dependency of one structure on another is expressed by the use of substructures. By binding the structure R in as a substructure, the structure S makes its dependence on R explicit, and furthermore makes S self–contained. Were R not incorporated as a substructure of S, the dependency of S on R would be implicit, reflected only in the fact that the type of the exception nonexistent refers to R, and not by the existence of any explicit reference to R within S. Such a situation is undesirable because S can only be meaningfully used in a context in which R is available as well, and if one does not tie the two together in some way, then an essential coherence in the structure of the program is lost.

Structure sharing occurs when two structures incorporate the same structure as a substructure. For example, if some structure Q were to bind in R as a substructure, then S and Q would be related by virtue of the fact that they both depend explicitly on the structure R. It is crucial that both S and Q have the *same* instance of R as substructure, for otherwise coherence is violated. The reason, in this case, is that datatype declarations in ML are *generative* in the sense that each elaboration of a datatype declaration defines a new type. Therefore if S and Q do not refer to the same instance of R, their components will not have compatible types. A similar problem would arise if R declared an exception that is handled or raised by functions in S and Q.

It may strike the reader as peculiar that we are emphasizing the need to maintain version control, especially in the setting that we have developed so far. At present we are describing a static configuration of structures, but later on we will turn to the problem of generating and updating these configurations. From that point of view, preservation of sharing is a crucial problem.

One of MacQueen's central observations is that the problem of dependency between structures can be reduced to preserving appropriate sharing relationships between structures and types. In this paper we will restrict our attention to structure sharing. There is no loss of generality in this assumption (a type can be taken to be a structure consisting of only the type declaration), but in practice this is inconvenient, and the full modules system treats type sharing separately from structure sharing. The reader is referred to [Car 85] for a discussion of type sharing in a persistent environment.

The key to managing structure sharing is to have a suitable notion of equality between structures that is sufficiently fine–grained as to capture the informal idea of "versions" or "instances" of a structure during program development. Each structure expression, when elaborated by the compiler, generates a new structure that is distinct from all other structures previously declared. This generalizes the generativity of data types mentioned above, and also captures the generative character of allocations of references to the heap. Structure equality is then defined to be in terms of a unique identifier (address or timestamp) associated with the structure. This identifier is used as the basis for structure equality tests in functors.

Signatures describe the interface of a structure, describing the names and roles of the identifiers declared within a structure. This information consists of the names of the type constructors defined in the structure, the types of the variables and exceptions, and the signatures of the substructures. Signatures are expressed by writing a *specification* between the brackets `sig` and end. Signature expressions are subject to an even more restrictive closure rule than are structures: they may contain no references to any external identifiers other than the pervasive primitives. They may, of course, be bound to identifiers using a `signature` binding, as follows:

```
signature SIGR =
  sig
    type t
    val f : t -> t
  end

signature SIGS =
  sig
    structure R : SIGR
    type 'a tree
    val search : 'a tree * 'a -> bool
    end
```

The utility of signatures lies in the relation called *signature matching* between structures and signatures. Roughly speaking, if a structure satisfies the specification given in a signature, then that structure is said to *match* that signature. The exact definition of signature matching is somewhat complex, but the idea is quite intuitive. For example, the structure `S` defined above matches the signature `SIGS` because

1. The structure `S.R` matches `SIGR`;
2. The type `S.tree` is a one–argument type constructor;
3. The variable `S.search` has the type given in `SIGS`.

This rough idea of the exact definition of signature matching suffices for our purposes. The interested reader is referred to [Mac 85,Har 86b] for a more precise account.

A signature `SIG` is said to be *more general than* another `SIG'` if every structure that matches `SIG'` also matches `SIG`. For example, the signature `SIGS'` obtained by deleting the specification for `search` from `SIGS` is more general than `SIGS`. This is derived from the fact that signature matching is liberal in the sense that the structure may have "extra" components that are not specified in the signature.

A signature may be (optionally) attached to the structure identifier in a structure declaration, indicating that the structure on the right of the binding must match the given signature. For instance, we may qualify the declaration of structure `S` above with the signature `SIGS` as follows:

`structure S : SIGS =` ... *as above* ...

Since the structure bound to `S` matches the signature `SIGS`, this is an acceptable declaration.

A program consists of a set of interrelated structures, encapsulated into a single root structure that binds the collection together into a unit. Each structure presents a well–defined interface to the others, and sharing is controlled by substructure bindings. Viewed statically, this appears to capture the idea of modular programming. But from the point of view of the dynamics of program development, we are lacking sufficient tools to create and modify such an arrangement of structures. For example, if we wish to modify the code of a function in the structure `R` above, then we need to recompile it and rebuild structures `S` and `Q`, using the same instance of `R` in both cases. It is impractical to insist that we simply recompile the entire program, thereby recreating the static configuration outlined above. What is necessary, of course, is some means of *relinking* `S` and `Q` with the new copy of `R`. This facility is provided by functors.

A functor is a function that, given one or more structures as arguments, yields a structure as result. The main use of a functor is to build a composite structure from argument structures, as we have indicated. Consider the structures R and S defined above. In order for S to be able to sustain a change to R, it must be possible to rebuild S with a new copy of R replacing the old copy. To do this, one defines a functor FS that, given a structure with signature SIGR, yields a structure with signature SIGS. Structure S is built by applying FS to R, yielding a new version of S with the new R incorporated as substructure. Similarly, the structure R itself is generated by a functor, but since R lies at the bottom of the dependency hierarchy, its functor has no parameters.[1]

```
functor FR() : SIGR =
  struct
    datatype t = ...
    val f : t -> t = ...
  end

structure R : SIGR = FR()

functor FS( R : SIGR ) : SIGS =
  struct
    structure R : SIGR = R
    datatype 'a tree = ...
    exception nonexistent : R.t
    val search(t:'a tree,x:'a) = ...
  end

structure S : SIGS = FS( R )
```

The dependence of S on R is flexible because S is built by applying FS to R. Since FS uses R abstractly (knowing only its signature), any instance of R can be plugged in to S by applying FS to that instance.

Now a functor may contain a sharing specification to ensure that two structures are built from a common substructure. For example, consider the structures S and Q discussed above, each with an instance of R in common. Then a functor to build another structure, say W, from S and Q will usually wish to require that it be given *compatible versions* of S and Q, for otherwise the combination will not be coherent. This is achieved by the sharing specification, as follows:

```
functor F( S: SIGS, Q:SIGQ sharing S.R = Q.R ): SIGW = ...
```

[1]The point of defining a nullary functor for R is mainly that in general one may want to obtain multiple instances of R without recompiling the body.

An application of this functor to two structures is acceptable only if the substructures R of each argument are equal in the sense of having the same unique identifier or address.

We shall limit our attention in this paper to "pure" functors, those without non–local references to the environment. The main significance of this restriction is that functors need not be represented as closures, and certain problems related to the signature of an impure functor are avoided. Another consequence of this restriction is that the structure resulting from a functor application can depend only on the parameter structures, and not on any other structures in the environment.

3 Persistence

In this section we discuss the problems of separate compilation, session preservation, and program libraries in the context of the ML modules facility. The essential observation is that each of these problems can be addressed from the point of view of persistent data. We will focus our attention on structures and functors as the units of persistence, as these are the fundamental units of program in ML. A more fine–grained notion of persistence would clearly be useful for some applications, but these issues lie beyond the scope of this paper. First, we discuss a general notion of persistence, and discuss its interaction with sharing and its suitability for support for program development. Then we isolate several special cases of the general notion, each motivated by a particular, limited application, but admitting a much simpler implementation than the most general form.

The most general notion of persistence, which we shall call *object persistence*, consists of viewing all objects as existing in persistent storage, with ephemeral storage serving only as a cache for quick access. Each object is identified by a *persistent identifier*, or *PID*, which is the address of that object in persistent storage. In effect, the program heap resides in persistent storage, with local caching in ephemeral storage for the sake of efficiency. The heap is garbage collected as usual, so that only accessible objects are preserved. In order to ensure that all accesses to persistent data are type safe, each object must have its type associated with it, and some sort of run–time typechecking is essential. When types are limited to pervasive primitives like `int` and `bool`, this is a well–understood notion. The situation becomes more interesting when user–defined abstract types are introduced; see [Car 85] for a treatment of this problem.

It has proved fruitful to view a persistent heap as a generalized form of database, and therefore it is normal for a program to access several persistent heaps. This entails that persistent identifiers must be relativized to to a particular heap, and also that primitives be provided for accessing heaps. A program connects to an address space using a primitive called connect. Access to address spaces is mediated by a *handle* [Car 85], typically a file name, that designates the address space to the run–time support system. Thus connect(H) returns some form of descriptor required for accessing the address space identified by the handle H. It is assumed that all changes to the persistent store are automatically preserved, and that any support for simultaneous access to an address space is provided by the underlying support primitives.

In the context of ML structures and functors are persistent objects, and therefore they are assigned PID's at creation time. The PID is used to mediate all access to the structure or functor, and it is also used to define equality between structures: two structures are equal if and only if they have the same PID. This corresponds to the fact that structure expressions are generative, in that each elaboration of a structure expression yields a unique structure. Functors are simililarly assigned PID's when they are created, but since there is no equality test for functors in ML, there is no need to be especially aware of this fact.

Notice that since we required that dependencies between structures be represented by a substructure hierarchy, it is always the case, even in a persistent environment, that related structures cannot be separated from one another. For example, since the structure S defined in Section 2 incorporates R as a substructure, the PID of R is bound to the local identifier R of S, thereby ensuring that the appropriate version of R is properly associated with S in the persistent heap.

It has been observed [Car 85] that it is necessary in a persistent environment to associate the type of a structure with the object in persistent storage in order to ensure type safety. In the context of ML, this means that signatures must be stored with each persistent structure, and a pair of signatures (one for the parameter and one for the result) and the sharing specification must be stored with each functor.[2]

Object persistence is the most general form that we could implement, and therefore is adequate as the basis for supporting the needs of a program development environment mentioned above. Session preservation is completely automatic in such an environment since every heap is implicitly preserved and can be accessed with the connect primitive. Thus we can view a session with ML as being initiated by connecting to a persistent address space, doing some work, and then exiting. To return to that session at some later time, we simply reconnect to that address space, and resume where we left off.

Of course, if all we are interested in supporting is session peristence, then a much simpler implementation strategy is available: simply write a snapshot of the heap to persistent storage. A session is resumed by reloading the snapshot into a virgin ML system. This approach is rather crude and inefficient since we

[2]The generalization to multiple argument functors is straightforward.

have no choice but to treat the entire heap as a monolithic object, without the possibility of treating any fragment of the heap separately, or being able to access it alongside other heaps.

One common application of session persistence is for a special case that we might call *program persistence*. The idea is to regard the significant result of a session to be a single structure consituting the root of some program hierarchy. For example, in the current implementation of ML, the ML compiler itself is merely a checkpointed heap that we reload in order to compile a program. The difficulty with this approach (aside from the aforementioned awkwardness of full checkpointing) is that it amalgamates the programmer's heap with the heap of the compiler itself. Thus if a user wishes to construct an ML program, the entire heap, including the compiler itself, must be written out as a unit.

Of course, if we were to use full object persistence, then there is a much better solution to the problem of program persistence. Since a program consists of a hierarchical arrangement of structures, each with a PID, then we can isolate a program as an independent address space by exploring the set of objects accessible from the root structure, and writing this out as a self–contained heap. Then any program may connect to this heap and use it freely, without having to replace the entire context as with simple–minded session persistence. Thus the problem of program persistence is solved by having persistent structures.

In fact, it is not necessary to have full object persistence in order to implement this strategy. All that is needed is the ability to collect together the accessible portion of the heap as a unit, and write this to persistent storage. Copying garbage collectors already provide this facility, since they work by consolidating the accessible portion of the heap in a contiguous region of virtual memory. If we define "accessible" to mean "accessible from a given structure", then we can use the collector to isolate that portion of the heap relevant to a given program, and write it to persistent storage. The main difficulty that arises is that one must be prepared to relocate this fragment of the heap when it is reloaded. But this is quite simple, using essentially the same technique in reverse. To reload a fragment of the heap, simply use the collector to shift the current heap in virtual memory so provide space for the saved heap, then read the saved heap into the vacated space, binding the address of the root to an identifier. The checkpointed heap fragment is now accessible and available for use. Of course, this implementation strategy is rather crude, but it is considerably easier to implement in an existing system than full object persistence, and is much more flexible than simple–minded checkpointing.

A similar approach to program persistence is used in Poly [Mat 85]. The idea is that one often does not wish to load a program into an existing context, but rather wishes to switch to the context of that program. Rather than shift the existing heap to make room for the imported heap, the current heap is merely replaced by the imported heap. This approach has the advantage over simple–minded checkpointing that it only stores the portion of the heap that is accessible from a given program, rather than storing the entire session context. However, it has the disadvantage that it destroys the current context. Although this is clearly a limitation, it appears that it is not a severe one from the point of view of program persistence.

Program persistence, however it is implemented, is supported by the introduction of two primitives, import and export. The effect of export S to H is to write the fragment of the heap rooted at structure S, together with its signature, to persistent storage, filed under the handle H. The exported heap fragment constitutes an address space in its own right, which we access by using the import primitive. The structure binding

```
    structure S : SIG = import H
```

reloads the heap fragment filed under handle H, and binds to the identifier S. This operation succeeds only if the signature associated with the structure stored at H is no more general than SIG, which guarantees type safety.

It is important to note that sharing *between* two programs (structure hierarchies) is not preserved across calls to export and import. The reason for this is that the structure S and all of the objects on which it depends is culled from the context in which it resides, and written as a single entity. If some other structure incorporates one of the components of the structure being exported, then this connection will be lost when the structure is re–imported. For example, suppose that S is the root structure of some program that incorporates structure R as a substructure, and that some other structure T, unrelated to S also incorporates R. Then after exporting S to persistent storage, and re–importing it at some later time, the relationship between T and S no longer holds. In fact, T may not even exist in the context into which S is imported. From the point of view of supporting program persistence, this is perfectly alright, since only the sharing *within* a program is relevant. Sharing between programs is purely an accident of no particular importance.

In Section 2 we discussed the role of functors as providing the means of relinking programs, and therefore lie at the heart of the dynamics of program development. The key idea is that the dependency between structures can be expressed functionally, provided that we have sharing clauses. A change to a program component is made by changing the corresponding functor, recompiling that functor (by rebinding it), and then relinking the entire program. Thus functors are the "object modules" of ML, and functor application performs linking. Separately–compiled program units and program libraries are naturally viewed as persistent (pure) functors.

Since a functor is an object, object persistence provides the means of supporting separately compiled program units and program libraries. But since functors have no non–local references to the heap, they may be isolated from the rest of the heap, much as we isolated structure hierarchies above. As a result, a simpler implementation strategy is available for the support of persistent functors based on the `import` and `export` primitives defined above. Here the non–destructive form of `import` must be used, for otherwise there would be no structures available to apply it to once we had retrieved it!

We extend the `import` and `export` primitives to functors as follows. The command `export F to H` writes the functor `F` to persistent storage at handle `H`. The parameter and result signatures and the parameter sharing clause are stored with the functor at `H`. A stored functor is re–imported with the following variant of a functor binding:

```
    functor G( R : SIGR ) : SIG = import H
```

This bindind succeeds only if the signature stored with the functor at `H` matches the given signature of `G` in the following sense. Suppose that the functor stored at `H` has the signature ⟨`SIGR'`,`SIG'`⟩. Then the declaration of `G` succeeds only if

1. Signature `SIGR` is no more general than `SIGR'`;
2. Signature `SIG` is at least as general as `SIG'`.

Note carefully the inversion between the two conditions. Recall that we say that a signature `SIG` is more general than `SIG'` if every structure that matches `SIG'` also matches `SIG`. An argument to the functor `G` must match the signature `SIGR`. If `SIGR` is no more general than `SIGR'`, then any argument to `G` matches `SIGR'` as well, and hence is acceptable as an argument to the stored functor. Conversely, the stored functor yields a structure that matches signature `SIG'`. This structure also matches `SIG` if `SIG` is at least as general as `SIG'`.

It is interesting to compare our approach to persistence with the persistent abstractions of Cardelli and MacQueen [Car 85]. Their interest is primarily with persistent abstract data types, and the problem of ensuring type safety in a persistent environment for a language with user–defined abstract types. Signatures are similar to their existential types, and structures are similar to their implementations of an existential type. However, we do not have any notion of a value with abstract type (which is an object that is "really" an object of the implementation type, but is treated as having a type distinct from the hidden implementation), and therefore most of the complications do not arise. They have no analog of functors or functor types in their paper. They do define some primitive operations for a persistent programming language, and we can cast our operations on structures in their terms as follows. The command `export S to H` can be expressed in their notation as `extern(dynamic(S,SIG),H`, where `SIG` is a signature matched by `S`. The declaration form `structure S:SIG = import H` can be expressed as `structure S = coerce intern(H) to SIG`.

4 Conclusion

The general notion of object persistence, adapted to the context of modular programming in Standard ML, provides an adequate framework for addressing some of the pragmatic aspects of program development such as session preservation, separate compilation, and program libraries. Three particularly useful forms of persistence, session persistence, structure persistence, and pure functor persistence, address certain special problems that arise in a program development environment, and admit simpler implementations than a persistent heap. Whether or not full persistence is necessary for our purposes remains unclear, though it appears that the real advantages of persistence lie in the context of database programming.

5 Acknowlegements

Kevin Mitchell and Dave MacQueen carefully read a draft of this paper and made many valuable suggestions.

References

[Alb 85] A. Albano, G. Ghelli, and R. Orsini, *The Implementation of Galileo's Values Persistence*, Proc. of the Persistence and Data Types Workshop, August, 1985.

[Atk 81] Malcolm Atkinson, Ken Chisholm, and Paul Cockshott, *PS–Algol: an Algol with a Persistent Heap*, Internal Report CSR–94–81, Computer Science Department, University of Edinburgh, 1981.

[Atk 85] Malcolm Atkinson and Ronald Morrison, *Types, Bindings, and Parameters in a Persistent Environment*, Proc. of the Persistence and Data Types Workshop, August, 1985.

[Atk 86] *Procedures as Persistent Data Objects*, ACM TOPLAS, October, 1985.

[Car 85] Luca Cardelli and David MacQueen, *Persistence and Type Abstraction*, Proc. of the Persistence and Data Types Workshop, August, 1985.

[Har 86a] Robert Harper, David MacQueen, and Robin Milner, *Standard ML*, Computer Science Department, University of Edinburgh, March 1986.

[Har 86b] Robert Harper, *Introduction to Standard ML*, Computer Science Department, University of Edinburgh, September 1986.

[Mac 85] David MacQueen, *Modules for Standard ML*, in [Har 86a].

[Mat 85] D.C.M. Matthews, *An Overview of the Poly Programming Language*, Proc. of the Persistence and Data Types Workshop, August, 1985.

[Mil 85] Robin Milner, *The Standard ML Core Language*, in [Har 86a].