Modules for Standard ML

David MacQueen

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

0. Abstract

The functional programming language ML has been undergoing a thorough redesign during the past year, and the module facility described here has been proposed as part of the revised language, now called Standard ML. The design has three main goals: (1) to facilitate the structuring of large ML programs; (2) to support separate compilation and generic library units; and (3) to employ new ideas in the semantics of data types to extend the power of ML's polymorphic type system. It is based on concepts inherent in the structure of ML, primarily the notions of a declaration, its type signature, and the environment that it denotes.

1. Introduction

ML is a functional language notable for its polymorphic type system [MIL78], which has proven quite successful in combining type security with the flexibility of generic functions. Moreover, automatic type inference in ML makes type declarations largely unnecessary, which is particularly convenient for interactive programming.

However, there are reasons for going beyond the basic polymorphic type system. The first is that certain kinds of parametric generic behavior are not expressible using simple polymorphic functions, because the type parameters must be assumed to carry a certain structure. For instance, it is easy to define a polymorphic function *reverse* of type α *list* $\rightarrow$ α *list* that computes the reverse of a list regardless of the type of the list elements. But it is impossible to define a polymorphic function *sort* of type α *list* $\rightarrow$ α *list*, because to sort a list one must compare the elements of the list and this cannot be done in a uniform fashion for all types α. However, we can define a uniform, parametric procedure for sorting lists whose elements belong to a given *ordering*, that is, a structure consisting of a type with an associated ordering relation. We would like to extend the concept of polymorphism to deal with this more general form of parametric behavior.

The second reason is that as one writes larger ML programs, serious problems of program organization and structure arise, and the type system needs to be augmented to help cope with these problems. As a practical matter, it is desirable that constructs for expressing large-scale program structure should also support type-secure separate compilation of program components and the resulting libraries of generic, precompiled units.

Fortunately, these requirements can all be satisfied by a single design based on the idea of treating declarations and the environments they denote as quasi-first-class entities in the language (*modules* and *instances*). Polymorphism is generalized by allowing abstraction of declarations with respect to their free type and value identifiers (*parameterized modules*). The type of an expression is generalized to the *signature* of a declaration. In order to deal with the important relations of inheritance and sharing between environments, we allow environments to contain other environments as components, giving them a hierarchical structure.

Many modern programming languages (e.g. CLU, Mesa, Ada, Modula-2) contain constructs, variously known as modules, packages, or clusters, designed to help a programmer organize large systems, but ML's semantic simplicity and regularity make it a particularly good base for the design of facilities for modularity. The key to the polymorphic type system is that ML programs are particularly insensitive to the representations of the types that they use, because values are typically handled via pointers; this simplifies the interface between a module and its client programs, making it possible to

abstract cleanly with respect to the module. The result is a greater degree of generality and flexibility than is possible in other typed languages, typically derived from Pascal, where storage allocation issues make client code highly sensitive to the representation of types (even "abstract" types). "Weakly typed" languages like Lisp share the flexibility of ML, but their lack of enforced type consistency or even a standard formalism for expressing type structure and interfaces can be crippling. One of the strengths of the present proposal is that it is a natural extension and generalization of the basic ML type system. It is not an afterthought or an external system description language, but an integral and organic part of ML itself.

This proposal is based on the fruits of a long collaboration with Rod Burstall on prototype designs for modules in Hope [MAC81], and on theoretical investigations with Ravi Sethi and Gordon Plotkin [MAC82, MAC84] that were motivated by those designs. The module designs for Hope were in turn influenced by the Clear specification language of Burstall and Goguen [BUR77]. Many hours of discussion with Luca Cardelli helped to solidify the ideas, and a limited prototype of modules implemented by Cardelli in Pose 3 of his ML system [CAR83a] has made it possible to gain valuable experience programming with modules. This paper is a companion to Robin Milner's proposal for the core of Standard ML [MIL83] and Luca Cardelli's I/O proposal [CAR83b].

The remainder of this section describes the motivation behind our approach to program modularization in ML and sets out the basic design principles. Section 2 describes the language constructs introduced, discusses inheritance and sharing, and indicates the necessary extensions to the ML typing rules. Section 3 presents some derived syntactic forms that streamline the syntax for certain common cases. Finally section 4 is a brief sketch of the underlying type theory.

1.1. The problem: managing environments

In its simplest form, a *module* is just a named collection of declarations whose purpose is to define an environment. It follows that one approach to the design of a module facility is to start with the notions of declarations and environments and consider how they can be made into relatively self-sufficient entities. In other words, what is the most natural way to promote declarations and environments to a quasi-first-class status?

The evaluation of a declaration* produces an environment. However, since a declaration often contains free identifiers of various kinds, its evaluation will require an environment that binds these free identifiers. Thus the meaning of a declaration *dec* is, roughly, a function

$$[\![dec]\!]: Env \rightarrow Env$$

Let us call the argument of this function the *environment of definition*, and the resulting environment the *defined environment*. For the usual case of a declaration embedded in an ML program, the environment of definition defaults to the statically prevailing environment of the declaration, and the defined environment is used to augment the prevailing environment to obtain a new prevailing environment for the scope of the declaration. The prevailing environment is guaranteed to be an appropriate environment of definition if the program as a whole (the declaration and its context) is well typed.

When we consider a declaration in isolation, however, we must explicitly constrain the environment of definition by specifying a *type signature* for it such that the declaration is well typed with respect to that signature. The signature of the environment of definition is not uniquely determined in general, but together with the declaration it determines, by type inference, a most general signature for the defined environment. Furthermore, a signature for the environment of definition is all that is required for compilation of the declaration.

It follows that a minimal form of module would be a declaration together with a signature specifying typing information for its free identifiers. If $M = <dec, sig>$ is such a pair, then it represents a function

$$[\![M]\!]: sig \rightarrow sig'$$

where sig' is the signature inferred for *dec* given the context specification sig, and a signature used as a type represents the collection of all environments satisfying that signature.

The next question is how to use such a module: where should we obtain an appropriate input environment (the environment of definition for *dec*) and what should we do with the resulting defined environment? One possibility is to treat the module as a mere abbreviation for the declaration associated with it. *Instantiating* (i.e. using or applying) the module at a given point in a program would then be equivalent to textually inserting the declaration at that point, and would be valid if the prevailing environment at that point satisfied the signature component of the module. As in the case of an ordinary declaration, the prevailing environment would serve as the environment of definition and the defined environment would be concatenated onto the prevailing environment.

This treatment of modules as essentially precompiled macros for declarations is sound, but it has unfortunate

---

* In Standard ML, there is a rich variety of compound declaration constructs, so a single declaration can result in a collection of bindings of different kinds (types, values, and exceptions).

consequences for program structure. For instance, the usual lexical scoping rules require modules instantiations to be textually grouped together in the same scope if they are to share a common context of definition, making it difficult to disentangle their results if these are to be used by different parts of program.

We could solve part of the problem by allowing the defined environment of a module instantiation to be named, so that the same environment could be used in several places. But complete control requires in addition that the environment of definition be explicitly specified in terms of named environments. Since only the signatures of these named environments are necessary for compilation, it is easy and natural to abstract with respect to them, obtaining a parameterized form of module. The explicit specification of prerequisite environments in terms of named module instances also is a powerful structuring discipline for program design.

1.2. Design principles

The design presented in the following section is based on the following principles and ideas.

1. We strictly separate the environment of definition from the environment of use. We require explicit specification of the complete environment of definition in terms of *antecedent* instances of specified signatures.

2. All modules are viewed as parametric, by abstraction with respect to their antecedent instances (even in the case where they have no antecedents). A module is a function which produces environments of a particular signature when applied to argument instances of specified signatures.

3. We provide for separate definition of interfaces (signatures) and their implementations (modules). This separation is essential for parametric modules.

4. The defined environment of a module must be closed. For example, no types may appear in its signature that are not defined directly or indirectly in the environment. This requirement implies that the defined environment must sometimes inherit certain antecedents.

5. We introduce environments of named signatures, modules, and instances. Such environments may be partially persistent, forming permanent systems or libraries.

6. We minimize the visibility of information by requiring explicit declaration of inheritance (information hiding).

7. Shared antecedents required for coherent interaction between module parameters must be explicitly specified.

2. The basic proposal

This section describes the three constructs making up the module proposal: signatures, modules, and instances. The syntax described here will be for the basic forms; some abbreviations and enhancements that make common idioms more convenient will be described in Section 3. Some familiarity with ML would be helpful, but is not essential. The syntax used in the examples is that of Standard ML [MIL83], which may be viewed as a hybrid of LCF ML [GOR79] and Hope [BUR80].

The central concept in this proposal is that of an environment structure (called an *instance* here) consisting of a set of bindings of types, values, and exceptions. An instance has two main functions: (1) it is a hybrid collection of entities incorporating related types, values, and exceptions; and (2) it provides names for its constituents.

The notion of an instance is actually somewhat more complex than just indicated, because of the problem of modeling inheritance. It will be essential to build new environments in terms of existing ones, and in such cases the new environment will often depend overtly on its antecedents (typically the type of a value bound in the new environment will involve types inherited from the antecedents). To express these dependencies, we allow environments to contain their required antecedent environments as components (i.e. as instance bindings). In a sense, each instance carries with it its own family tree, or at least as much of its family tree as is necessary to make use of the instance (see Section 2.2 below).

Instances, as environment structures, will have their own kind of type, called a signature. Basically, a signature gives appropriate type specifications for each of the bindings making up an instance (the arity of a type constructor, the type of a value or exception, and the signature of an antecedent instance). From another point of view, however, instances themselves can often be considered a generalized form of type, an *interpreted* type, wherein a type (or types) is combined with operations and exceptions with which to manipulate its values.

2.1. Basic forms: signatures, modules, and instances

As mentioned above, a signature is a type specification for an environment. The notation for signatures is

```
sig
    instance specs
    type specs
    val specs
    exception specs
```

```
      end
```

The various kinds of specifications may be interspersed, as long as names are introduced before being used (except in the case of recursive type specifications). The following signature specifies instances with a type `elem` and a binary predicate `eq` (presumably representing an equality relation) over `elem`:

```
sig
    type elem
    val eq: elem * elem -> bool
end
```

Of course, writing out all signatures in full soon becomes very cumbersome, so as usual we introduce a new kind of declaration for naming signatures. Thus we can declare

```
signature EQ =
  sig
    type elem
    val eq: s * s -> bool
  end
```

As another simple example, here is a declaration of a signature for stacks as a unary type constructor with appropriate operations and exceptions.*

```
signature STACK =
  sig
    type 'a stack
    val nilstack: 'a stack
    and push: 'a * 'a stack -> 'a stack
    and empty: 'a stack -> bool
    and pop: 'a stack -> 'a stack
    and top: 'a stack -> 'a
    exception pop: unit and top: unit
  end
```

A *module* is a declaration supplied with an explicit context in the form of a set of parameter instances with specified signatures. The purpose of the declaration is to create a new environment structure (that is, an *instance* of the module) relative to the particular context provided by a set of actual instance parameters. The declaration part of the module is type checked relative to the signatures of the parameters, which must bind all the free identifiers occurring in the declaration, yielding an inferred signature that must agree (in a sense to be defined later) with any explicitly declared signature for the module result.

Here is the rather tired but still serviceable example of a module for stacks that implements the signature given above.**

```
module StackMod ( ) : STACK
    type 'a stack = nilstack' ! push' of 'a * 'a stack
    exception pop: unit and top: unit
    val nilstack = nilstack'
    and push = push'
    and empty(nilstack') = true !
        empty _ = false
    and pop(push'(_,s)) = s !
        pop _ = escape pop
    and top(push'(x,_)) = x !
        top _ = escape top
```

---

* Identifiers beginning with an apostrophe such as `'a` are used as type variables. The asterisk (*) and arrow (->) represent the product and function space type constructors. `unit` is a primitive type analogous to Algol 68's void type, containing a single defined value `` `( )'' ``. The keywords `type` and `val` introduce type and value declarations and specifications.

** The type declaration defines `stack` as a unary *type constructor* representing, for each `'a`, a disjoint union tagged by the two *data constructors* `nilstack'` and `push'`, which can be used in patterns matching against stack values. Functions are defined as a list of rules with an argument pattern on the left and a body expression on the right. The underscore (_) is a special wild-card pattern element that is often used as a default.

```
end
```

In this particular example the declaration making up the body of `StackMod` is entirely self-contained (i.e. there are no free identifiers), so the module has no parameters. Nevertheless, the module must still be applied (to a trivial, empty parameter set) to produce an instance:

```
instance Stack = StackMod ()     { Stack: STACK }
```

Multiple Instances. It is possible to instantiate such a module more than once, producing several instances, and the stack type constructors in each instance will be distinct, as will the types of the operators such as `push`. Multiple instances of a "pure" module such as Stack are sometimes useful, but usually a single instance will do, since all instances provide essentially the same resource and a single instance could be shared by all "clients" without interference.

However, when instances have state it is often appropriate to create more than one instance of a module even though that module has no parameters. For example, we could define a module implementing stacks of integers such that each separate instance of the module was a stack.*

```
module StackMod' ()
    local
        val stack: int list ref = ref nil
          {local stack data structure}
    in
        val push x = (stack := x :: !stack)
        and pop () = (stack := tl(!stack))
        and top () = hd(!stack)
    end
end
```

Note that this version of stacks does not introduce a new stack type with each instance. Instead, the module itself (or more precisely its result signature) plays the role of a type, and is in fact quite similar to a class in Simula or Smalltalk, with its instances corresponding to objects of the class. This stack module can be used to create several instances, each of which serves as a separate stack object with its own internal stack data structure.

Type Propagation. When defining a module that implements a signature like STACK, it is natural to declare the type component `stack` as a new type, so that each instantiation of the module creates a new, unique type constructor. But consider the signature EQ of a type with an equality predicate. In this case it is probably not interesting to create instances with entirely new types. Rather the typical use of this signature and its instances is to impose structure on an existing type.

For example, we might like to define an instance of EQ wherein the type component is `int` and the `eq` predicate is ordinary integer equality. We can do this with the aid of a new form of "transparent" type declaration which simply binds a given type to a name. The syntax for such declarations is:

```
type elem is int
```

With this new form of declaration, we can use the following module declaration to produce the desired instance.

```
module IntEqMod () : EQ
    type elem is int
    val eq = (op =)
end

instance IntEq = IntEqMod ()
```

Another interesting example of the use of this form of declaration is given by the following module for producing lexicographic orderings on lists. This example also shows a module with a nontrivial parameter.**

```
signature ORDSET = sig
    type s
    val le: s * s -> bool
```

---

* Here `ref` constructs updateable references to values, `!` dereferences, and `hd`, `tl`, and `::` (infix cons) are the usual prim-itive functions for lists. Note that module declarations may omit the result signature specification, as in this example; type inference is used to determine an anonymous signature for the module's instances.

** Note the use of qualified names such as `O.s` to refer to components of instances. Declarations for "opening" an instance for unqualified naming of its components are discussed in Section 3.

202

```
            end

module LexOrdMod(O: ORDSET): ORDSET
     type s is O.s list        {derived from type component of parameter}
     val le(nil,_) = true | le(_,nil) = false | ...
end

module IntOrdMod ( ) : ORDSET
     type s is int
     val le = op <=
end

instance IntOrd = IntOrdMod( )
{ IntOrd: ORDSET
  IntOrd.s = int
  IntOrd.le: int * int -> bool }

instance LexOrdInt = LexOrdMod(IntOrd)
{ LexOrdInt: ORDSET
  LexOrdInt.s = int list
  LexOrdInt.le: int list * int list -> bool }
```

It is important to note that *identity* of component types is preserved, so that one can write expressions such as

```
LexOrdInt.le([1;x+1], (2*y)::1)
```

that mix operations on lists and integers with the `le` operation from `LexOrdInt`. In fact, `LexOrdInt.s` is just another name for the type `int list`.

2.2. Inheritance

We can distinguish two classes of instance parameters. Parameters in the first class provide some utilities used internally to implement the desired environment, but do not affect the result signature of the module and are irrelevant to its users.

The other class consists of those instance parameters that are relevant to the *use* of the derived instance as well as its definition. Consider the following example of a module that, given an instance of `EQ` (a type with an equality function), defines membership in and equality between lists of elements of that type.*

```
signature LISTEQ = sig
     instance E: EQ
     val member: E.elem * E.elem list -> bool
     and eqlists: E.elem list * E.elem list -> bool
end

module ListEqMod (E': EQ) : LISTEQ
     instance E = E'
     val member(e,nil) = false |
          member(e,e'::1) = E.eq(e,e') orelse member(e,1)
     and eqlists(nil,nil) = true |
          eqlists(e1::11,e2::12) = E.eq(e1,e2) andalso eqlists(11,12)
end
```

In this case, the types of `member` and `eqlists` in `LISTEQ` are clearly dependent on the type `elem` inherited from the instance `E`, so without this instance as a component, the signature (and the corresponding instances) would not be self-contained. But there is a subtle issue here: why not inherit just the type `E.elem` from the instance `E` instead of the whole instance, since only the type is involved in the signature `LISTEQ`? The reason is that the membership and equality functions for lists over a given type are predicated on a particular equality function for the elements, and so proper use of the list functions may require knowledge of that equality function. Another kind of dependence involves exceptions, since functions in a derived instance may raise exceptions declared in a parameter instance, so the parameter instance would have to be inherited if we wished to selectively handle those exceptions. In short, an instance parameter should be inherited whenever it is required as context for the proper use or interpretation of the derived instance.

The instance declaration

```
instance E = E'
```

in `ListEqMod` is necessary to cause the inheritance of the parameter `E'` as a component of the result of the module. This is a rather cumbersome form, however (especially if we had used `E` instead of `E'` as the formal parameter name), and we will provide a derived form for conveniently specifying which parameter instances are to be inherited. An inherited instance component of a module may also be a instance component of a parameter.

2.3. Sharing

The inheritance relation between instances defines a dependency hierarchy, and since several instances may be built on the same antecedents, the form of this hierarchy is a directed acyclic graph rather than a tree. The sharing of antecedents among instances is not just incidental, for the common antecedents form the basis for communication between instances.

In the absence of parametric modules, any required sharing between instances of modules can be insured "by construction". That is, the hierarchy is built from the bottom up, and later instance constructions refer to specific shared instances created earlier.

When parametric modules are introduced, it sometimes becomes necessary to place explicit sharing constraints on parameter instances. These sharing constraints express assumptions about common antecedents that are essential for integrating the resources provided by the different parameters; they insure that the parameters "fit together" properly. In ordinary polymorphic types, sharing is expressed by repeated occurrences of a type variable, as in

```
val map: ('a -> 'b) * 'a list -> 'b list
```

Our problem is to express sharing of component instances as well as types. The solution is to introduce a new kind of declaration that indicates sharing by equating *paths* through the inheritance hierarchy, where a path is a sequence of subinstance names separated by "." and terminating with either a subinstance name or a type name. The syntax for these declarations is

```
sharing path1 = path2 = ... = pathn
```

To illustrate the problem and its solution in detail, consider the following set of signature and module definitions which might be part of a facility for bit-mapped graphics. {*Example abbreviated for summary.*}

```
signature POINT = sig
    type point
    . . .
end


signature RECT = sig              signature CIRCLE = sig
    instance P: POINT                 instance P: POINT
    . . .                             . . .
end                               end


module RectMod(P: POINT) : RECT   module CircleMod(P: POINT) : CIRCLE
    instance P = P                    instance P = P
    . . .                             . . .
end                               end


signature GEOMETRY = sig
    instance R: RECT
    instance C: CIRCLE
    sharing R.P = C.P
    . . .
end


module GeometryMod(R: RECT, C: CIRCLE) : GEOMETRY
    instance R = R
    instance C = C
    sharing R.P = C.P
    . . .
end
```

Note the sharing declarations in both the GEOMETRY signature and the GeometryMod module. These indicate that the parameters R and C should be based on the same POINT component (named P in both RECT and CIRCLE).

Now suppose we define two different instances of POINT corresponding to two coordinate systems (e.g. transformed versions of an underlying screen coordinate system). The sharing constraints in GeometryMod will prevent us from confusing the different sorts of points.

```
instance Point1: POINT = ...    {expresses one coordinate system}
instance Point2: POINT = ...    {a different coordinate system}


instance Geom1 = GeometryMod(RectMod(Point1), CircleMod(Point1))  {OK}
instance GeomX = GeometryMod(RectMod(Point1), CircleMod(Point2))  {WRONG}
```

It is important to note that the sharing specification in GeometryMod is essential for proper type checking of the module, since the functions from the parameter instances R and C will attempt to interact in terms of the type point that they inherit from their respective POINT components R.P and C.P. The sharing declaration will allow the type checker to identify these two versions of the type point (i.e. R.P.point = C.P.point).

2.4. Type checking

Signature Matching. The type checking of module definitions and instantiations involves one way matching of a *candidate* signature against a *target* signature. When checking the body of a module against its declared signature, the declared signature is the target and the inferred signature of the body is the candidate. When checking a module application, the declared parameter signature is the target and the signature of the corresponding actual parameter is the candidate. The matching is based on the assumption that polymorphic types appearing in a signature are generic, that is, implicitly universally quantified. This means that a polymorphic function imported from a parameter instance can be used in a module with several different instantiations of its polymorphic type. This is in contrast to the rules for ordinary functions, which cannot use polymorphic parameters generically in their bodies.

The matching of signatures is based on matching corresponding components with identical names. Thus a value component "foo: ty" in the candidate signature must match a corresponding value component "foo: ty'" in the target, with ty' being an instance of ty. There must be a one-to-one correspondence between components in the candidate and target signatures, but this strict requirement is mitigated by the ability to define *views* (Section 3) of an instance with a restricted and possibly renamed set of components.

Typing instance components. The type specifications for values and exceptions in a signature are specifications relative to the actual type components of instances of the signature, so to get the true type of a value or exception component we must combine the type schema of the signature with the type bindings in the instance itself. The types in the signature schema (and the instance components as well, which behave like types in this respect) are really bound dummy variables, and the nature of their binding is a form of existential quantification (see Section 4).

In summary, the rule for determining the type of a value or exception component of an instance is to take the type schema for that component in the signature and replace all type constructor names bound in that instance or its antecedents with their bindings in the instance. (When the instance is a module formal parameter, we create dummy type components for the parameter and use them to instantiate the schema.) This is the justification for the type propagation phenomena discussed in Section 2.1.

3. Derived forms and extensions

This section presents some derived forms which make the module facilities considerably less cumbersome to use.

Global references. In the basic module facility, any instance that is to be used in a module must be passed as a parameter. This is a simple and uniform convention, but it is also somewhat unnatural in a case where the same instance will be passed as a parameter every time the module is instantiated. In such situations it would be convenient to simply refer to the particular instance as a global instance name. We have in fact already been following this practice of using global identifiers in the case of signatures.

The use of such global (or free) identifiers implies that there must be some context in which they are bound, and the fact that we want to separately compile modules implies that this context should be persistent, i.e. that it should exist independently of any particular invocation of ML. These requirements can be satisfied by introducing the notion of a *system* or *library*, that consists of a permanent collection of named signatures and (precompiled) modules and directions for reconstituting certain named instances "on demand" (alternatively, it may be possible to make the instances themselves persistent). The system concept could also provide programming support features such as the ability to automatically recompile all modules which are affected by a change in one module (like the "make" command in Unix).

Direct instance definitions. In the majority of cases, a module will only be instantiated once, creating a single

instance. Since this usage is so common, it is worthwhile to provide a special syntax direct definition of a single instance:

```
inst
     declarations
end
```

This form of definition does not allow parameters, of course. Any other instances used in the body must be referenced as globals.

Inheritance declarations. To avoid the awkward form "instance E = E'" used in the `ListEqMod` example to cause the parameter E' to be inherited as an instance component, we introduce the declaration

```
inherit instance-name-seq
```

This declaration can only be used in a module, and it indicates which of the module's parameters are to be inherited as instance components of the module's instances.

Opened instances. The use of qualified names (or "paths") to refer to components of instances can become very tedious in deeply nested hierarchies such as the Standard ML compiler. In that example we find specifications like

```
val LookupVar : E.T.L.Ide * E.Env * int * (E.T.L.Ide * int) E.T.A.Association
                     -> E.T.TypeExp * Displacement
```

where a module high in the hierarchy is referring to types introduced several levels below. Equally cumbersome names will occur in expressions.

Often these qualified names are not essential, because there is only one binding of a given identifier among the antecedents and so no danger of ambiguity. To gain direct rather than qualified access to the identifiers bound in an instance, we can declare that instance to be "open". In a signature, we use an instance specification of the form

```
open instance name: signature
```

while in module and instance definitions we use the declaration

```
open instance-name1 . . instance-namen
```

to gain direct access to the bindings of the named instances.

The effects of "open" specifications in signatures and "open" declarations in modules are independent. An open specification in a signature has effect with respect to a client using instances of that signature, flattening out the signature from the client's point of view.

The open declaration in a module makes the bindings of the named instances directly available *within that module*. In other words, the scope of bindings made accessible by an open declaration in a module is limited to the module body -- they are not exported along with the module's bindings. In order to cause an instance component of a module to be open in an inferred signature, one must use declarations such as

```
open instance E = E'

open inherit E
```

Open declarations can also be used in ordinary ML expressions and declarations, where the scoping of the revealed bindings follows the usual rules.

Views. Sometimes one wants to make an instance X of some signature SIG1 masquerade as an instance of some other, presumably simpler signature SIG2, so that X can be passed to a module requiring a SIG2 parameter. Often this can be accomplished by restricting and renaming the bindings of X. This process can be thought of as creating a new "view" of the instance [GOG83], or as applying a "signature morphism" to the instance. Such a signature transformation can easily be expressed as a parameterized module, or as an *ad hoc* definition of a new instance derived from the old.

It is debatable whether any additional syntactic sugaring is needed for this process of creating new views of instances, but we have considered some additional forms.

4. Foundations

Here we give a very brief sketch of the type theory underlying instances, signatures and parameterized modules. The basic problem is how to model hybrid objects such as instances that include types as well as values related to the type components. The solution is to consider the type components of a simple instance signature (one which does not involve instance components) as being existentially bound. For instance, the signature

```
sig
   type elem
   val eq: elem * elem -> bool
end
```

can be considered to be a sugared form of the existential type

$$\exists elem \,.\, elem * elem \rightarrow bool$$

Conversely, by analogy with constructive logic the values of an existential type such as $\exists t.\sigma(t)$ are pairs $<\tau, v:\sigma(\tau)>$ where τ is a *witness* for the bound type variable t and v is a value type $\sigma(\tau)$. These values correspond roughly to instances.

To obtain the type of a parameterized module it sometimes suffices to simply build functional types over the existential types of the parameter and result signatures. But in the case where the result instance inherits types from the parameters, it is necessary to change the sense of the quantification and universally quantify the shared type variable over the functional type of the module.

These ideas are closely related to the dependent type structures of Per Martin-Löf's intuitionistic type theory [MAR75], and recent work by John Mitchell, Gordon Plotkin, and the author to explain type abstraction in terms of the "quantification theory" of types.

5. Conclusion

The design described here is the latest in a long series of approximations to the ideal of a module facility that is an organic development of basic principles of language structure and type theory. It is based on the functional language ML because of ML's particularly clear and simple structure. The underlying foundations are the lambda calculus and the very natural polymorphic type structure for the lambda calculus discovered by Curry, Hindley, and Milner. The most important novelties in the design deal with inheritance and sharing, notions which become critical when extending the basic philosophy of ML to the realm of programming in the large.

References

[BUR77]R. M. Burstall and J. A. Goguen, *Putting theories together to make specifications*, Proc. 5th Int. Joint Conf on Artificial Intelligence, Cambridge, Mass., August, 1977, pp. 1045-1058.

[BUR80]R. M. Burstall, D. B. MacQueen, and D. T. Sannella, *Hope: an experimental applicative language*, Conf. Record of the 1980 LISP Conference, Stanford, August 1980, pp. 136-143.

[CAR83a]L. Cardelli, *ML under Unix*, Polymorphism, I.3, December 1983.

[CAR83b]L. Cardelli, *Stream Input/Output*, Polymorphism, I.3, December 1983.

[GOG83]J. A. Goguen, *Parameterized programming*, Proceedings of Workshop on Reusability in Programming, A. Perlis, ed.

[GOR79]M. J. Gordon, R. Milner, and C. P. Wadsworth, **Edinburgh LCF**, LNCS Vol. 78, Springer-Verlag, New York, 1979.

[MAC81]D. B. MacQueen, *Structure and parameterization in a typed functional language*, Symp. on Functional Languages and Computer Architecture, Gothenburg, Sweden, June, 1981, pp. 525-537.

[MAC82]D. B. MacQueen and R. Sethi, *A semantic model of types for applicative languages*, 1982 ACM Symp. on Lisp and Functional Programming, Pittsburgh, August 1982, pp. 243-252.

[MAC84]D. B. MacQueen, G. Plotkin, and R. Sethi, *An ideal model for recursive polymorphic types*, 11th Annual ACM Symp. on Principles of Programming Languages, Salt Lake City, January 1984, pp. 165-174.

[MAR75]P. Martin-Löf, *An intuitionistic theory of types: predicative part*, Logic Colloquium 73, ed. H. E. Rose and J. C. Shepherdson, North-Holland, Amsterdam, 1975, pp. 73-118.

[MIL78]R. Milner, *A theory of type polymorphism in programming*, JCSS, 17(3), December 1978, pp. 348-375.

[MIL83]R. Milner, *A proposal for Standard ML*, Polymorphism I.3, December 1983.