# An extension of Standard ML modules with subtyping and inheritance

John Mitchell\*
Dept. of Computer Science
Stanford University
Stanford, CA 94305
mitchell@cs.stanford.edu

Sigurd Meldal<sup>†</sup>
Dept. of Informatics
University of Bergen
Bergen, Norway
sigurd@eik.ii.uib.no

Neel Madhav<sup>†</sup>
Dept. of Computer Science
Stanford University
Stanford, CA 94305
madhav@cs.stanford.edu

#### **Abstract**

We describe a general module language integrating abstract data types, specifications and object-oriented concepts. The framework is based on the Standard ML module system, with three main extensions: subtyping, a form of object derived from ML structures, and inheritance primitives. The language aims at supporting a range of programming styles, including mixtures of object-oriented programming and programs built around specified algebraic or higher-order abstract data types. We separate specification from implementation, and provide independent inheritance mechanisms for each. In order to support binary operations on objects within this framework, we introduce "internal interfaces" which govern the way that function components of one structure may access components of another. The language design has been tested by writing a number of program examples; an implementation is under development in the context of a larger project.

#### 1 Introduction

This paper describes a general module system which provides approximately the same functionality as a combination of Standard ML modules (signatures, structures and functors) [HMM86],  $C^{++}$  classes [Str86], and Ada packages [US 80], enhanced with a form of specification. Our design effort stems from a larger project to produce a prototyping language and programming environment. However, since our modules most closely resemble Standard ML [Mac85, HMM86], we have chosen to present the module design as an extension of that language. Many of the subtyping ideas we use here have appeared in previous

work, *e.g.*, [CM89]. The primary new features of this language are the generalization of structures to encompass typed "first-class" objects, and inheritance primitives for specifications and structures.

Our module design is a proper extension of Standard ML. We extend signatures to *specifications*, which may contain axioms as well as the types of structure components, and extend the basic typing discipline to include subtyping. A new feature is a set of inheritance primitives for specifications, and a parallel mechanism for structures. A major change is that we essentially allow signatures (specifications) to be used as ordinary types. This has the consequence of allowing structures to be passed as function arguments, returned as function results, or stored in reference cells. In imprecise but suggestive jargon, we generalize Standard ML structures to first-class "objects." To keep the typing problem for this language tractable, we must change our notion of equality for type components of structures. In effect, we make use of the "first class abstract data type" approach of [MP88], also used in the programming language Quest [Car89]. However, this change is applied in a systematic way which coincides with Standard ML typing for "top-level" structures and functors (the cases allowed in Standard ML). Another change is that we allow a specification name to appear in the type of one of its components (a form of recursive type), and provide an expression self for defining such components of structures. A novel feature, in comparison with other object-oriented languages, is the use of internal interfaces in specifications. The need for this new construct arises in part from our separation of specification and implementation, and our flexible forms of inheritance; see Section 6.1. Finally, we allow parameterized specifications (which seem especially useful for object-oriented programming) and use bounded quantification [CW85, CCH+89, CM89] rather than unconstrained ML polymorphism.

One of our language design goals is to support a range of programming styles within a single framework. One style of interest is the traditional program development by "step-wise refinement," using abstract data types and specifications. However, it is recognized that for realistic, large-scale programs, this idealized approach is not always effective. The main drawback is that in the pure style advocated in the early 1970's [Dij72, Par72], a significant amount of effort may be put into the design before any code is written. Therefore, program behavior cannot be used to evaluate the program design until relatively late in the development cycle. It is often better to write a simple

<sup>\*</sup>Supported in part by an NSF PYI Award, matching funds from Digital Equipment Corporation, the Powell Foundation, and Xerox Corporation; NSF grant CCR-8814921 and the Wallace F. and Lucille M. Davis Faculty Scholarship.

<sup>†</sup>Supported by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-90-J1232 and by the Air Force Office of Scientific Research under Grant AFOSR83-0255.

 $<sup>^{\</sup>ddagger}\mbox{Supported}$  by the Norwegian Research Council for Science and the Humanities.

"prototype" early in the life of a project and use properties of the prototype program as a basis for further refinement of either the design or implementation. We therefore envision an approach to program development which involves successive refinement of working code, with substantial code reuse from one prototype to the next. Further discussion of our prototyping model may be found in [BL90].

We are unsatisfied with previous object-oriented languages for a number of reasons. One is the lack of support for specified algebraic data types, which are familiar to and valued by our intended user community. Another is the way that specifications (typically signature information only) and implementation are identified in a single language construct. (An exception is Emerald [BHEL86], which has several similar features, but lacks inheritance.) We believe that the separation of specification from implementation will prove beneficial to the development of large object-oriented programs, both by allowing independent use of either specification or implementation inheritance (see [Sny86]), and by allowing multiple implementations of a single "class" to coexist peacefully within a single program. Thus we provide explicit language support for what is often achieved by the programming technique of so-called "abstract classes" [GR83, page 66-72]. In addition, our unified approach to abstract data types and objects allows object-oriented concepts to be mixed with other programming styles, and provides inheritance of specification and implementation for algebraic data types. Finally, we provide a compile-time typing discipline which is more flexible than  $C^{++}$ , without the type insecurity of Eiffel [Coo89]. Our type system is an extension of ML polymorphism, based in part on the record calculus of [CM89], and including "F-bounded quantification" [CCH<sup>+</sup>89] for specifying uniform behavior over functionally similar classes of objects. An additional goal of our project, not reported here, is the integration of a form of concurrency based on objects with concurrent methods. An implementation is underway encompassing most of the features discussed in this paper, as well as concurrency at the "method" level and various debugging tools.

In this paper, we will focus primarily on the programstructuring aspects of our language, leaving further discussion of specifications and their use to later publications. The basis for our specification language and development tools is [LvHKBO87, LHM+86]. To a large extent, when restricted to the context of Standard ML, our specifications do not differ substantially from Sanella and Tarlecki's *Extended ML* [ST89]. Since we have not fully investigated the problem of ML-style type inference with subtyping, we will consider ML an explicitly typed language, as explained in [MH88]. However, based on past experience, we do not foresee any serious obstacles in extending ML-style type inference to our language. The reader concerned with type inference and subtyping may consult [Mit84, Wan87, Rém89, JM88].

In Section 2, we give an informal introduction to the module language by describing two simple stack examples. Section 3 follows with a brief glossary of basic terminology. In Section 4, we briefly outline our use of subtyping. The main features of our form of modules are described in Section 5, with motivating examples for certain design decisions given in Section 6. Section 7 places our extension of Standard ML in a type theoretic perspective.

# 2 Examples of abstract data types and classes

To give some feel for the module extension, we will compare two forms of stack specification, one as an abstract data type (ADT) and the other as a class of objects. While these both use the same language constructs, the two programming styles are distinctly different. In both examples, as an expository convenience, we use equality freely without considering its logical interpretation.

In the traditional ADT approach, stacks might be introduced by the following form of *specification*, which is our extension of Standard ML *signature*:

```
specification Stack_spec =
  external
  type S[type t];
  val empty[type t]:S[t];
  fun push[type t]:t*S[t] \rightarrow S[t];
  fun pop[type t]:S[t] \rightarrow S[t];
  fun top[type t]:S[t] \rightarrow t;
  ...
  constraint
  for all type t, x:t, s:S[t].
    pop[t](push[t](x,s)) = s;
    top[t](push[t](x,s)) = x;
  ...
end
```

Except for the explicit type parameters and axioms, this is essentially the usual Standard ML signature. (The keyword external indicates that the named components are accessible outside the structure body; we also have internal interfaces, described later.) A module implementing stacks would then give a representation of stacks (as a function of the type of elements stored), and code for the associated operations. Although our construct is slightly different, we will use the Standard ML term *structure* for a basic module.

For the purpose of writing Standard ML-style programs, our most significant extensions are subtyping and inheritance. Inheritance is useful for defining one module from another, and subtyping allows the programmer to indicate that elements of one type are substitutable for elements of another. For example, we might specify that a type of stacks with an additional operation is a subtype of the stacks implemented as above. (A similar use of subtyping and ML abstract data types was presented in [JM88], where a type inference algorithm is developed.) A pervasive difference between our module system and Standard ML is our treatment of type equality, specifically, equality of type components of structures. We allow structures to be used as first-class values, with the consequence that within certain well-delineated contexts, type components may become "hidden" as opposed to visible types. This is discussed in more detail in Section 5.3 and from a typetheoretic perspective in Section 7.

An object-oriented approach to stacks involves a "class" of objects, each of which implements a single stack. We will have one type of objects for each type of stack entry, given by the following parameterized specification.

```
specification Stack_class[type t] =
  external
  fun is_empty:unit → bool;
  fun push:t → Stack_class[t];
  fun pop:unit → Stack_class[t];
  fun top:unit → t;
  ...
  constraint
  for all type t,s:Stack_class[t], x:t.
    (s.push(x)).top = x;
    (s.push(x)).pop = s;
  ...
end
```

A structure M satisfying the specification Stack\_class[t], for some type t, is essentially an "object" with operations M.push, M.pop, and so on. (The Standard ML type unit is a trivial, one-element type used to indicate that M.pop, for example, has an empty parameter list and returns a value when called, possibly having a side effect.)

Objects are implemented as structures. Following Standard ML terminology, we refer to parameterized structures as *functors*. A general implementation of stack objects would be a functor which creates objects of type Stack\_class[t] when called with type parameter t.

The list pointer contents is private to an object since it does not appear in the Stack\_class external interface. For the reader unfamiliar with Standard ML, we note that the operator! returns the contents of an assignable reference cell. The body of push deserves explanation. This is a sequence of expressions. The value returned is the value of the second expression, self, after evaluating the first for side effect. Since the value of self, which is not part of Standard ML, is the entire structure, the effect of s.push(x) is to add x to the contents list of s and return the resulting stack structure. Although without self we could write a similar push function with only a side effect, this would make compound expressions such as s.push(x).top meaningless.

Since we separate the interface to an object from the function which creates objects, it is straightforward to give several implementations of a single class. This leads to a difficulty with binary operations [Mit90a], which is resolved using internal interfaces, as discussed in Section 5

#### 3 Basic concepts and terminology

**structure:** Syntactically, a structure is an encapsulated set of declarations, including types, functions (procedures)

and other values. Semantically, a structure is a composite value determined by a set of such declarations.

**functor:** A functor is a parameterized definition of a structure, as in Standard ML.

specification: A specification defines two interfaces, called the *external interface* and the *internal interface*. Each interface consists of a signature, giving names of components and their types, and an optional set of constraints (axioms) restricting the behavior of structure components. The external interface defines the visible components of the specification, while the internal interface may impose additional demands on all implementations of the specification. These are described in greater detail in Sections 5.1 and 6.1. We consider a specification a form of type, namely, the type of all structures meeting this specification. Our use of *specification* corresponds to Standard ML *signature* or Ada *package specification*, extended with an internal interface and constraints.

type: A type is a collection defined by a type expression or specification. Types are separated into two kinds, which we call *small types* and *large types*. This distinction corresponds to the Standard ML distinction between signatures and types. The small types are basic types such as int, bool, string, types obtained from these using constructors such as record and array, and specifications with only abstract type components. Values from any small type may be stored in a reference cell, for example, or returned as the result of a conditional expression. Specifications, in general, define large types. Implicit coercion from large to small types is discussed in Section 5.3. The names *small* and *large* are related to constructive type theory (see, *e.a.*, [Mar84, C+86, Mac86, Mit90b]).

subtype: A type A is a subtype of B if every expression of type A may be used in any context requiring an expression of type B, without type error. In fact not all subtypes are recognized by the compiler, just as most languages do not recognize all possible type equalities. The compiler only recognizes A as a subtype of B if this follows from the form or declaration of at least one of these types.

**inheritance:** We use "inheritance" informally to refer to various mechanisms for using one declaration in writing another. We have separate inheritance mechanisms for specifications and for structures.

# 4 Extending ML with subtypes

Subtyping is a general extension to Standard ML which effects both the so-called "core language" of basic expressions and declarations, as well as the module system. Since the subtyping ideas we use, outside of the module system, have been proposed in previous papers [CW85, Mit84, JM88] [CM89], we will only give a brief summary of subtyping. More detailed discussion of subtyping for specifications and structures appears in Section 5. We use the notation A <: B to indicate that A is a subtype of B.

There are five ways that a type A may become a subtype of B, depending on the forms of A and B.

basic types: If A and B are basic, predefined types, then
 A <: B iff this is given as part of the language definition.</pre>

**type constructors:** If A and B are types defined using the same type constructor, then A <: B iff this follows from the subtyping properties for this constructor. For example, if  $A = A_1 \times A_2$  and  $B = B_1 \times B_2$  are both cartesian products, then A <: B iff  $A_1 <: B_1$  and  $A_2 <: B_2$ .

specifications: If A and B are specifications, then A <: B iff specification A includes all of the component declarations and constraints of B, and the declaration of either A or B makes it clear that subtyping is intended.</p>

**declared type identifier:** If the declaration of a type identifier t specifies that t is a subtype of A, then t <: B iff A <: B. A similar rule holds for **formal parameters:** a formal parameter t is a subtype of B only if this follows from a subtype or sharing constraint in the formal parameter list containing t.

**predefined supertypes:** Several designated basic types are supertypes of all types of a given form. For example, record is a supertype of all record types.

Subtyping is essentially a generalization of type equality. In Standard ML, and most typed languages, we generally assume that if A and B are equal types, then any expression with one of these types also has the other. With subtyping, we augment type equality with the so-called "subsumption" rule:

If A <: B and expression e has type A, then expression e has type B.

Subtyping may also be used in parameter lists, to restrict the possible values of formal parameters. This is illustrated by example in Section 6.2.

A point worth mentioning briefly is the lack of connection between subtyping of specifications and subtyping of abstract data types defined by these specifications. Consider, for example, a specification Q\_spec introducing a type Q of queues with the usual operations, and a similar specification DQ\_spec introducing a type DQ of double-ended queues. Since any structure satisfying DQ\_spec also satisfies Q\_spec, it is reasonable to consider DQ\_spec <: Q\_spec. In common terms, this means that any implementation of doubly-ended queues is also an implementation of queues. However, if M:Q\_spec is a structure satisfying Q\_spec and DM:DQ\_spec is a structure satisfying DQ\_spec, the component type DM.DQ is *not* necessarily a subtype of M.Q.

#### 5 General description of specifications and structures

# 5.1 Specifications

Syntactically, a specification has three main parts: an inheritance list, an external interface and an internal interface. The inheritance list may give one or more specifications which may be extended, restricted or copied, as described in Section 5.2 below. Each interface consists of a signature, giving names of components and their types, and an optional set of constraints restricting the behavior of structure components. For the purpose of this paper,

the constraints may be regarded as axioms about the components, written in a version of multi-sorted first-order logic whose sorts are the types of the programming language. The external interface describes the structure components that are available outside the structure and the internal interface describes components that are visible only to other instances of that specification and its subtypes. The motivation for having separate internal and external interfaces is illustrated in Section 6.1. When a type component is listed as part of a specification, a list of subtypes and supertypes of this type may be given.

#### 5.2 Specification inheritance

There are three "inheritance" operations on specifications. The first is extend. If specification A is defined by extending B, then the declaration of A need only mention components and axioms that are not given in B. The result is that specification A is equivalent to the union of the declarations of A and B, and the type A becomes a subtype of B. The converse operation is restrict. If A is defined by restricting B, then the declaration of A lists components and axioms of B that are to be omitted from A. The result is that A is a supertype of B. A specification may extend several specifications, defining a subtype of each.

The third operation is copy, which is essentially equivalent to inserting a textual copy of a specification, and does not result in a subtype or supertype. Either an entire specification may be copied, or the two forms

copy \langle specification\_name \rangle except ...
copy \langle specification\_name \rangle only ...

may be used. In the first case, all components of a specification are copied, except those listed. The second form only includes the listed components, which may be more convenient if only a relatively small part of a specification is required.

An added feature of extend and restrict is the ability to rename components. This adds some complexity to subtyping for specifications, since the names of the components in one type may not correspond directly to the names of components in a supertype. However, the conversion from one type to another is straightforward (and easily implemented), and this feature seems to provide a very effective way for the programmer to resolve name conflicts in multiple inheritance. Renaming with copy is straightforward since copy-ing a specification is not required to result in a subtype.

# 5.3 Structures

Our structures are similar to Standard ML structures, with inheritance primitives added. The most significant difference is in our treatment of type equality.

The desire for decidable compile-time type checking imposes certain restrictions on computations with ML modules ([MH88, HMM90]). For instance, ML signatures, structures and functors may only be declared at "top level," and functor application is restricted to top level. This is because the type part of a functor application is required at compile-time [MTH90, HMM90]. This conflicts with our desire to support object-oriented programming, since we would like to allow arbitrary computation on structures

such as the object-oriented style stacks mention in Section 2.

Our solution to this problem is to allow arbitrary computation with structures, but to limit type equality to what may be determined at compile time. In a sense, where Standard ML restricts the language, we choose instead to restrict the degree of "dataflow analysis" used in type checking. If a structure has no type components, then computing type equality simply does not enter into the picture: A structure without type components is essentially a record, and arbitrary computation may be allowed without complication. Similarly, as shown in [MP88], if all type components are abstract types, there is no need to restrict computation on structures. The only problem arises when a structure has a type component which we expect to be equal to another type. In this case, we cannot expect to test equality at compile time, if the structure is a result of an arbitrary function call or conditional expression. Using a systematic criterion which is related to the distinction between small and large types (see [Mac85, MH88, HMM90]), we identify a class of compile-time evaluable structure expressions. These include the application of a top-level functor to a top-level structure. If usage dictates that the type component of a structure must be equal to another type, then the type checker determines whether the structure is given by a compile-time expression. If so, then the test for equality is based on the actual structure value. If not, then the type is considered "abstract," and distinct from other types, and the test may therefore fail. This mechanism is supplanted by the sharing constraints of Standard ML, which we generalize to subtype assertions. If a structure is a formal parameter, then the type relationships guaranteed by sharing constraints in the parameter list are acknowledged within the function body.

Although the above description of type equality has an operational, algorithmic quality, the basic method has a straightforward explanation using standard type-theoretic concepts. This is summarized in Section 7. Essentially, when a structure is used in a conditional expression, we view this as an implicit coercion of visible type components to abstract types.

Since structures have many similarities to records, it might seem natural to identify objects with ML records instead of structures. If ML typing were extended with a flexible form of record typing, as in [Wan87, Rém89, CM89] for example, this would allow much of the object-oriented capability we provide. One important difference, however, is inheritance. Rather than extend ML records with inheritance, we have chosen to integrate objects into the module system. This has the beneficial effect of allowing the same primitives to be used for specification and implementation of abstract data types, "objects," and various hybrids.

#### 5.4 Structure inheritance

One motivation for allowing separate inheritance of specification and implementation comes from the fact that these two quite often work most naturally in "opposite directions" ([Sny86]). This may be illustrated by considering queues and double-ended queues. It is natural to specify double-ended queues by extending the specification of queues, since half of the specification of double-ended queues is already present in the specification of queues. On the other hand, if we have an implementation of double-

ended queues, then it is easy to implement queues by simply hiding some operations. This implementation of queues does not depend on how the implementation of double-ended queues works, and so we could change the latter without needing to redefine the former. However, if we tried to implement double-ended queues by adding operations to the implementation of queues, we lose this abstraction. A change in the implementation of queues might require us to change the implementation of double-ended queues. In this case, separate inheritance allows us to define both our specifications and implementations in the most natural way.

There is one inheritance operation on structures, copy, with several options. The options allow inheritance of any or all components, renaming, and visibility restrictions. The following examples, using both options, include all but one of the basic keywords.

```
copy \langle specification_name \rangle only ... rename ...
hide ...
copy \langle specification_name \rangle except ... rename
... show ...
```

In the first example, only the listed components are copied. A list of renaming clauses, of the form \( \component\_name \)\ to \( \component\_name \)\, indicate which components are renamed, and how. All of the copied components become visible components of the new structure, except those listed in the hide clause. The names except and show are complements of only and hide.

The simplest semantic description is textual copy. To give an example, an operation copy A only f, g in the declaration of structure B indicates that B will have visible components B.f and B.g derived from A. The behavior of f and q is exactly the same as if both of their declarations in A (if that is where they are declared) had been textually copied into the declaration of B. In particular, if f is a function calling g, then this call in the body of f refers to the inherited g. If f also calls a function h, then this call will go to the function h declared in B, not A. This is essentially the usual form of inheritance, as in Smalltalk, for example, except that we do not require all of a structure to be inherited. We are able to provide this flexibility as a result of the fact that we do not force the two structures to have related types (signatures or specifications). The more complicated example copy A only f,g rename f to k may be understood as a textual copy, but with f renamed to k, including renaming all occurrences of f in the bodies of f and q. The effect of hide is straightforward.

# 6 Motivating Examples

#### 6.1 Internal interface

We present two examples that illustrate the use of internal interfaces.

#### 6.1.1 Time-stamped events

Consider time-stamped events. A specification for these might look as follows:

```
specification time_stamped_event =
  external
  fun precedes:time_stamped_event → bool;
  fun event_name:unit → string;
  ...
  end
```

Our implicit intention is that each structure of type time\_stamped\_event will have a local real variable which stores the time of the event. Since the only time-related operation on time-stamped events is precedes, this variable in not part of the external interface. However, if ev1, ev2: time\_stamped\_event are two events, then ev1.precedes(ev2) should compare the times of the two events, returning a boolean value. This requires ev1 to have access to the time of ev2. If the function precedes must be implemented using only public operations on its argument, then the only solution is to make the time of an event public. This seems undesirable, since only other time-stamped events need to see the time of an event.

The internal interface solves this problem appropriately by letting us specify that every time-stamped event must have a time component. By putting this requirement in the internal interface, we make the time component of one event accessible to the function components of other events, without making time generally available. With an internal interface added, the specification will now appear as follows:

```
specification time_stamped_event( ... ) =
  external
  fun precedes:time_stamped_event → bool;
  fun event_name:unit → string;
  ...
  internal
    val time:ref real;
  end
```

# 6.1.2 Sets with union

Suppose we wish to specify and implement a class of set objects with a union operation. In object-oriented style, the union of two sets is computed by sending one set as an argument to another. A first attempt at specifying set objects might be as follows. For expository convenience, we assume that equality on type elem is provided in some way, and that an equation e1 = e2 has a meaningful interpretation, even when evaluating one of these expressions might ordinarily side-effect the other. We may explicitly require equality on elem using means similar to those de-

scribed in Section 6.2.

```
specification SET[type elem] =
external
  fun empty
                     unit → bool;
  fun element
                     elem \rightarrow bool;
                  : elem → SET[elem];
  fun insert
  fun delete
                  : elem → SET[elem];
                     SET[elem] \rightarrow SET[elem];
  fun union
  fun intersect :
                     SET[elem] → SET[elem];
constraint
 SET[elem].empty;(* all sets initially empty *)
 for all o1,o2:SET[elem], e1,e2:elem.
    o1.empty => not o1.element(e1);
    o1.insert(e1).element(e2) <=>
             e1 = e2 or o1.element(e2);
    o1.delete(e1).element(e2) <=>
             e1 \neq e2 and o1.element(e2):
    o1.union(o2).element(e1) <=>
             o1.element(e1) or o2.element(e1);
    o1.intersect(o2).element(e1) <=>
             o1.element(e1) and o2.element(e1);
end SET;
```

If we try to implement the above specification, we run into difficulties with union(s:SET[elem]). The problem is that the natural, efficient implementations of union need to iterate over the list of the elements in the argument s, but there is no apparent way to do this using only the public operations allowed on s. One not quite satisfactory approach is to implement sets in a *lazy* style. While this makes union easy to compute, it becomes difficult to implement empty properly in the presence of intersection. A more realistic approach is to extend the external interface with a function element\_list which returns a list of all the elements in the set. This has the undesirable effect of making more operations public than required. A more subtle reason for keeping element\_list out of the external interface is that we expect equality to be a congruence relation, but we do not require equal sets to have the same ordered element list.

We may solve this problem using internal interfaces. An internal interface requiring an element\_list function forces each set to define a function element\_list. However, this function is not publicly visible – it may only be used by other set objects. The following example illustrates the use of constraints to state correctness properties of internal interface functions.

```
internal
  fun element_list : unit → List[elem];
constraint
  local in_list : elem * List[elem] → bool
  for all s:SET[elem], l:List[elem], e:elem.
    in_list(e, l) <=> e = hd(l) or in_list(e, tl(l));
    s.element(e) <=> in_list(e, s.element_list())
end
```

The declaration <code>local in\_list</code> ... indicates that the predicate identifier <code>in\_list</code> is local to the constraint, and not part of the interface. With this internal interface, implementing SET is straightforward.

We should emphasize the fact that the internal interface makes it easier for a single program to contain several different implementations of the same specification. In

the pure object-oriented style, any operation in a specification that needs an argument of the same type (binary operations, in essence) cannot make any assumptions about the argument, since it could have an arbitrary internal representation. The internal interface makes it easier to write binary operations by allowing the specification to impose restrictions in addition to the external interface.

Internal interfaces are used in a variant of the  $C^{++}$  friend concept. If elements of type A need access to the internal interface of B structures then the specification of B may list A as a friend type.

# 6.2 Polymorphism over structurally similar types of objects

In many situations, it is useful to define structures or functions which work uniformly over all types of objects with certain operations. Although it is a simple example, sorting illustrates the main points. Using abstract data types, a polymorphic sorting function would require a comparison (order) relation on the type of list elements to be sorted. This form of sort function would have type

```
sort[type\ t]: (t*t \rightarrow bool)*List[t] \rightarrow List[t]
```

In an object-oriented style, we would expect each object in the input list to have a comparison function component. This requirement must be incorporated into the type of sort, as a restriction on the type of list elements. Based on previous investigation [CCH $^+$ 89], the appropriate way to restrict the type of list element appears to be through the introduction of a parameterized specification ordered[type t]. A parameterized specification ordered[type t] is essentially a function which, given any type t, returns a specification requiring each x:t to have a comparison function x.less\_than of type x.less\_than: t  $\rightarrow$  bool. Intuitively, the function x.less\_than tells, for any y:t, whether x is less than y. Given this parameterized specification, we can write the type of sort as

```
sort[type t <: ordered[t]] : List[t] \rightarrow List[t]
```

Intuitively, if t is a type of objects, each having a comparison function component, then sort maps lists of t's to lists of t's. Although the reader might see the restriction type t<:ordered[t] on the type of list elements as "recursive," it is not; see [CCH+89] for further discussion. Note that the same form of parameterization would be required if we wish to parameterize a binary search tree specification or implementation by the type of ordered records inserted into trees.

If specifications of objects are simply signatures, then the straightforward form of qualified polymorphism described above is sufficient. However, since we wish to give constraints in addition to signature information, a problem arises. This may be seen by trying to define the parameterized specification ordered[type t]. A naive first attempt might be the following.

```
specification ordered[type t] =
  external
   fun less_than:t → bool
  constraint
  for all x,y,z:t.
   not x.less_than(x)
   x.less_than(y) and y.less_than(z) implies
        x.less_than(z)
  end
```

Although this specification might at first appear to express the intended restriction on t, it is *not* syntactically well-formed. The problem is that the constraint section assumes that if x:t, then x has a function component x.less\_than. However, there is no assumption about the type t which justifies this. The inquisitive reader may wish to ponder this point, and attempt to devise a solution before reading further.

One solution to this problem is to use two specifications, one giving the signature part of this specification, and the other (depending on the first) expressing the constraint. More specifically, we may write the following two specifications.

```
specification order_sig[type t] =
  external
    fun less_than:t→bool
  end

specification
order_constraint[type t<:order_sig[t]] =
  constraint
  for all x,y,z:t.
    not x.less_than(x)
    x.less_than(y) and y.less_than(z) implies
        x.less_than(z)
  end</pre>
```

The important change from the single specification is that the parameter t of the order\_constraint is explicitly required to be a subtype of order\_sig[t], and so it is clear within the specification body that any x:thas a less\_than component of the appropriate type.

Since the parameterized specification we originally desired is a kind of "sequential conjunction" of two specifications, we may define ordered by the following declaration.

We also provide syntax for defining **ordered** and similar parameterized specifications directly.

#### 7 Type-theoretic perspective

Our extension of Standard ML may be understood within the type-theoretic framework used in [Mac86, MH88, HMM90] to explain the Standard ML module system. While the indexed categorical view of [HMM90] provides some interesting insight into the distinction between compile-time and run-time, we will restrict ourselves to the simpler and more accessible framework of [MH88], based on a predicative typed lambda calculus *XML* with two "universes" of types (see also [Set89]). In this section, we will briefly review

XML and then describe an extension  $XML^+$  with subtyping and constructs illustrating the essential features of our extended module system. Since the basic subtyping features of  $XML^+$  are subsumed by the more ambitious system of [CM89], we emphasize the modules as opposed to subtyping in this document.

The language XML is a typed lambda calculus with two universes of types,  $U_1$  and  $U_2$ , which we have referred to informally as the "small" and "large" types, respectively. Since the language may be defined with respect to any choice of basic types, type constructors and expression constants, the small types of XML may include basic types such as integers and booleans, and type constructors such as list. The expressions of these types may be polymorphic, depending on type variables, or may depend on structures.

The modules of XML are constructs of the second universe  $(U_2)$ , which contains general products and sums over the small types, and  $U_1$  itself. For those unfamiliar with basic type theory, we review the notion of general sums and general products. If *A* is a collection (such as a type) and B(x) is an expression defining a collection for each  $x \in A$ , then the *general product type*  $\Pi x : A.B$  is the collection (or type) of functions f from A to the union  $\bigcup_{x \in A} B$ of all B(x) such that for each  $x \in A$ , we have  $f(x) \in B(x)$ . For example, if list(t) is the type of t lists, for any small type  $t: U_1$ , and nil(t) is the empty list of type list(t), then we can say that the function *nil* has type  $\Pi t : U_1.list(t)$ . For collection *A* and *A*-indexed family of collections B(x)as above, the *general sum type*  $\Sigma x$  : A. B is the collection (or type) of pairs  $\langle a, b \rangle$  such that a : A and b : B(a). For example,  $\Sigma t: U_2$ . list(t) is the type of pairs  $\langle t_{\ell}, \ell \rangle$  with list  $\ell$  belonging to type *list*( $t\ell$ ).

As explained in [Mac86, MH88], the signatures, structures and functors may be regarded as syntactic sugar for general products and sums, and their members. More specifically, the signature of a Standard ML structure may be regarded as a general sum type, and a structure itself as an element of such a type, since a structure is essentially a kind of tuple. Although functor signatures are omitted from Standard ML, the type of a Standard ML functor is a general product type  $\Pi x:A.B$ , with type A giving the type of the functor parameter and type B defining the type of the resulting structure. The reason a functor does not have an ordinary function type is that the type of structure produced may depend on the actual structure parameter, as in the following example:

```
functor M(S:spec type t; val x:t end) :
    spec
        type s; val x:list[S.t]
    end
```

In our extension of Standard ML modules, we have both general sum types, as specifications of structures, and general product type specifications for functors. In addition, we have parameterized specifications, which actually have types of the form  $U_2 \rightarrow U_2 \rightarrow ... \rightarrow U_2$ , technically taking us beyond  $U_2$ .

One novel feature of our module system, in comparison with Standard ML, is an impredicative treatment of existential types and a related implicit coercion from  $U_2$  to  $U_1$ . As described in [MP88], see also [CW85], existentially quantified types correspond to the programming language notion of abstract types. Like Quest [Car89], our existential types are impredicative, so that a specification with

only abstract (or, in Ada terminology, "private") types may be considered a  $U_1$  type instead of a  $U_2$  type.

For example, the following Standard ML specification

```
specification S =
spec
  type t;
  val x:t
end
```

would ordinarily be regarded as a general sum type

$$S ::= \Sigma t : U_1 . t$$

Since this type is in  $U_2$  and not  $U_1$ , a function such as conditional could not be applied to structures of this type. More precisely, conditional is an operation with type (in curried form)

cond: 
$$\Pi t: U_1. bool \rightarrow t \rightarrow t \rightarrow t$$
.

The reason for restricting conditional to  $U_1$  types is that a higher level conditional would interfere with compile-time type checking by allowing, for example, type expressions such as cond (e > 5)  $int\ bool$ , for arbitrary integer expression e. Since conditional only applies to  $U_1$  arguments, we cannot apply cond to a structure M:S.

However, if the type component of some structure M:S is hidden, then there is no problem with using a conditional expression. Some examples of this are discussed in [MP88], including an example choosing between sparse and dense matrix implementations at run time. In order to achieve this flexibility, we recognize that if the type component of M:S is to be considered abstract, then we may think of S as the existential (as opposed to general sum) type

$$S_{\exists}$$
 ::=  $\exists t : U_1.t$ 

belonging to  $U_1$  instead of  $U_2$ . In effect, the typechecker recognizes that there is a canonical map

$$hide: \Sigma t: U_1.t \rightarrow \exists t: U_1.t$$

which does nothing but "hide" the identity of the type component of a structure by mapping a pair into a type which does not have a first projection function. By automatically coercing structures from  $U_2$  to  $U_1$ , we allow structures to be treated as first-class values, essentially losing only the ability to test equality between type components.

#### 8 Conclusion

We have developed an extension of Standard ML modules which incorporates subtyping, inheritance and a form of "object" in a uniform way. The language design has been tested by writing a number of examples, and an implementation is under development. While the language design does not solve all of the problems discussed in [Mit90a], for example, the current design seems practically useful, and more flexible in certain respects than existing typed object-oriented languages. In particular, the separation of specification from implementation and the use of internal interfaces to allow efficient, type-safe implementation of binary operations seem useful. We have found "F-bounded polymorphism," introduced in [CCH+89], to be a useful

primitive, and sufficiently expressive for describing families of classes satisfying certain axiomatic specifications. A direction we are currently exploring is an extension of objects (Standard ML structures) with concurrent methods. In particular, the design and use of an appropriate specification language seem to require some consideration.

Acknowledgements: This module design is part of the design of a prototyping language and system, under development in collaboration with a team lead by David Luckham (Stanford), Frank Belz (TRW), Sigurd Meldal and John Mitchell. We are grateful to members of this team for discussion, guidance and feedback. We also thank Dinesh Katiyar for a number of helpful discussions and suggestions.

#### References

- [BHEL86] A. Black, N. Hutchinson, E.Jul, and H. Levy. Object structure in the Emerald system. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 78–86, October 1986.
- [BL90] F. Belz and D.C. Luckham. A new approach to prototyping ada-based hardware/software systems. In *Proc. ACM Tri-Ada'90 Conference*, December 1990. To appear.
- [C+86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*, volume 37 of *Graduate Texts in Mathematics*. Prentice-Hall, 1986.
- [Car89] L. Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989. presented at IFIP Advanced Seminar on Formal Descriptions of Programming Concepts.
- [CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273– 280, 1989.
- [CM89] L. Cardelli and J.C. Mitchell. Operations on records. In *Math. Foundations of Prog. Lang. Semantics*, 1989. To appear. Also available as DEC SRC Technical Report 48, August 1989, 60 pages.
- [Coo89] W.R. Cook. A proposal for making Eiffel typesafe. In European Conf. on Object-Oriented Programming, pages 57–72, 1989.
- [CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
- [Dij72] E.W. Dijkstra. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, Structured Programming. Academic Press, 1972.

- [GR83] A. Goldberg and D. Robson. *Smalltalk-80:* The language and its implementation. Addison Wesley, 1983.
- [HMM86] R. Harper, D.B. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, Lab. for Foundations of Computer Science, University of Edinburgh, March 1986.
- [HMM90] R. Harper, J.C. Mitchell, and E. Moggi. Higherorder modules and the phase distinction. In *Proc. 17-th ACM Symp. on Principles of Programming Languages*, pages 341–354, January 1990.
- [JM88] L. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 198–212, July 1988.
- [LHM+86] D. C. Luckham, D. P. Helmbold, S. Meldal, D. L. Bryan, and M. A. Haberler. Task sequencing language for specifying distributed ada systems. In *Lecture Notes in Computer Science*, Number 275, pages 249–305. Springer-Verlag, May 1986.
- [LvHKBO87] David Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. Anna
   - A Language for Annotating Ada Programs. In Lecture Notes in Computer Science, Number 260. Springer-Verlag, July 1987.
- [Mac85] D.B. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2), 1985. 35 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.
- [Mac86] D.B. MacQueen. Using dependent types to express modular structure. In *Proc. 13-th ACM Symp. on Principles of Programming Languages*, pages 277–286, 1986.
- [Mar84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
- [MH88] J.C. Mitchell and R. Harper. The essence of ML. In *Proc. 15-th ACM Symp. on Principles of Programming Languages*, pages 28–46, January 1988.
- [Mit84] J.C. Mitchell. Coercion and type inference (summary). In *Proc. 11-th ACM Symp. on Principles of Programming Languages*, pages 175–185, January 1984.
- [Mit90a] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17-th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.
- [Mit90b] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990. (To appear.).

- [MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12-th ACM Symp. on Principles of Programming Languages*, 1985.
- [MTH90] Robin Milner, Mads Tofte, and Robert Harper. The Definition of Standard ML. MIT Press, 1990.
- [Par72] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.
- [Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In 16-th ACM Symposium on Principles of Programming Languages, pages 60-76, 1989.
- [Set89] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.
- [Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. 1-st ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.
- [ST89] D. Sanella and A. Tarlecki. Towards formal development of ML programs: foundations and methodology. Technical Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, February 1989.
- [Str86] B. Stroustrop. *The C*<sup>++</sup> *Programming Language*. Addison-Wesley, 1986.
- [US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language*. GPO 008-000-00354-8, 1980.
- [Wan87] M. Wand. Complete type inference for simple objects. In *Proc. 2-nd IEEE Symp. on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in *Proc. 3-rd IEEE Symp. on Logic in Computer Science*, page 132, 1988.