Manifest types, modules, and separate compilation

Xavier Leroy *

Stanford University

Abstract

This paper presents a variant of the SML module system that introduces a strict distinction between abstract types and manifest types (types whose definitions are part of the module specification), while retaining most of the expressive power of the SML module system. The resulting module system provides much better support for separate compilation.

1 Introduction

1.1 Modules and separate compilation

Modularization is the process of decomposing a program in small units (*modules*) that can be understood in isolation by the programmers, and making the relations between these units explicit to the programmers. *Separate compilation* is the process of decomposing a program in small units (*compilation units*) that can be typechecked and compiled separately by the compiler, and making the relations between these units explicit to the compiler and linker. Both processes are required for realistic programming: modularization makes large programs understandable by programmers; separate compilation makes large programs tractable by compilers.

Several languages rely on a common mechanism to provide modules and separate compilation. A typical example is Modula-2 [27], where modules are identified with compilation units composed of an implementation file (source code) and an interface file (specification). However, this identification is limiting. Since compilation units are usually directly mapped onto file system objects, separate compilation tends to keep the structure of compilation units simple, with the dependencies "hard-wired" inside the units. Modern module systems go much farther in their attempts to accurately express the program structure. A well-known example is the module system of SML [14], which is actually a small typed language of its own, with modules (also called *structures*) as the base data structure, module specifications (*signatures*) as types, functions from modules to modules (*functors*) to represent parameterized modules, and function applications to connect modules—all features that cannot be accounted for in the "modules as compilation units" approach.

As a consequence of this tension, SML makes no provision for separate compilation. SML is defined as "an interactive language" [17], implying that users are expected to build their programs linearly in strict bottom-up order. This requirement can be alleviated by systematic use of functors, at the cost of extra declarations (sharing constraints) and late detection of inter-compilation unit type clashes. Recently, Shao and Appel [24] have proposed a more free-form separate compilation mechanism for SML, which infers the required constraints, but delays all type checks between compilation units to the linking phase, which is much too late. Late detection of type errors increases the likeliness of programmers writing large quantities of inconsistent code, only to discover later that major changes are required to bring the parts together.

The work presented in this paper grew out of an attempt to apply the Modula-2 separate compilation techniques (which ensure early detection of inter-compilation unit type clashes) to the SML module system. The starting idea is to abandon the identification of modules and compilation units, and consider compilation units as an additional layer on top of modules: just as Modula-2 compilation units are collections of language objects (types, variables, functions), SML compilation units should be collections of module objects (signatures, structures, functors). These collections of modules can, then, be defined in implementation files and specified (by their signatures) in interface files, and their dependencies can be expressed by Modula-2-style `import` declarations.

1.2 The problem with SML modules

The simple approach outlined above turns out to fail, not because it is inherently flawed, but because it exposes a weakness in the SML module system: a module signature does not express all the typing properties that the remainder of the program can assume about the corresponding structure. In other terms, SML signatures are not complete specifications with respect to typing. This is because type specifications in signatures are "transparent": they do not hide the actual type provided by the structure. For instance, assume a structure `S` has a signature Σ specifying a type component `t`. Even though the signature does not say anything about the implementation of `t`, another structure `S'` can rely on `S.t` being implemented as some particular type, say, `int`. If `S` and `S'` are not defined in the same compilation unit, the implementation defining `S'` cannot therefore be typechecked until the implementation defining `S` has been written: the correspond-

*Dept. of Computer Science, Stanford University, Stanford CA 94305-2140. E-mail: xavier@cs.stanford.edu. Supported by an INRIA post-doctoral grant.

POPL 94- 1/94, Portland Oregon,USA

© 1994 ACM 0-89791-636-0/94/001..$3.50

ing interface, specifying only `structure S : Σ`, does not suffice to determine whether `S'` is correct in assuming `S.t` to be `int`. Hence, typechecking and compilation must be done in bottom-up order, just as in a toplevel-based approach. In contrast, "true" separate compilation, as in Modula-2, allows typechecking and compilation of a program fragment at any time, based only on the interfaces of the fragments it imports.

The fact that type specifications in SML signatures are transparent is no accident: it accounts for a large part of the expressive power of SML modules. Treating type specifications as opaque, that is, making all exported types abstract, would fix the problem with separate compilation but drastically reduce the expressiveness of the module system [15].

1.3 This work

This paper proposes a way out of this dilemma: make type specifications opaque (so that the users of a structure can only assume what is declared in its signature), but enrich signatures with *manifest type specifications*. A manifest type specification of the form `type t` $= \tau$ not only declares a type identifier `t`, but also publicizes that it is implemented as the type expression τ. This way, signatures become complete module specifications with respect to typing, making separate compilation feasible while retaining the expressiveness of the SML module system. (Harper and Lillibridge [10] have investigated similar ideas independently.)

The two components of this approach—manifest types and opaque signatures—have already been proposed as extensions to the SML module system: type abbreviation in signatures and MacQueen's `abstraction` construct, respectively. The novelty of this paper is to take these two concepts as the basic mechanisms of a module system, replacing SML's transparent type specifications instead of supplementing them.

The bulk of this paper is devoted to the study of the resulting variant of the SML module system, with opaque type specifications and manifest types in signatures. This module system supports the SML modular programming style in a satisfactory way. It provides a strong type abstraction mechanism, which guarantees interesting representation independence properties [18] and easily accounts for generative datatypes. More surprisingly, the manifest type mechanism subsumes large parts of the SML sharing constraint machinery, an essential part of the SML module system: manifest types in functor argument position express sharing constraints between types, and the simple typing rules for manifest types suffice to check these constraints. The main missing SML feature is sharing constraints between structures (though identity checks on structures can be encoded using abstract types); we argue that this is a small price to pay for the overall simplifications resulting from this restriction.

On the technical side, the main originality of this paper is the use of what is essentially weak sums [19, 7]—albeit with an unusual elimination construct: the "dot notation" [4, 5]—instead of the strong sums that have been used so far to give type-theoretic accounts of SML modules [15, 12, 13]. Unlike strong sums, weak sums provide direct support for type abstraction and make the "phase distinction" [13] obvious. The well-known inadequacies of weak sums for modular programming [15] are here offset by the extra expressiveness brought by manifest types.

The present paper also puts forward a new way to account for type sharing, distinct from the heavy graph-based formalism of the *Definition* [11, 17, 25] and from Aponte's record-based module algebra [1]. Previous approaches to sharing focus on structure generativity and sharing between structures; as a consequence, they require stamps over structures and consistency conditions between structures having the same stamp. In contrast, sharing restricted to types, as in this paper, can be expressed by a standard term algebra without extra consistency checks. More generally, the *Definition* uses semantic objects (richer than signature expressions) in the static semantics, while our type system uses only syntactic objects (signature and module type expressions), in keeping with the typed λ-calculus tradition.

1.4 Outline

The remainder of this paper is organized as follows. Section 2 introduces manifest types and illustrates how they propagate type equalities and express sharing constraints. Section 3 formalizes a small SML-like module calculus with manifest types. Section 4 shows the expressiveness of this calculus by encoding a first-order calculus with strong sums in it. Section 5 mentions some simple extensions of this work, followed by concluding remarks in section 6.

2 Informal development

2.1 Transparency in SML

The SML module language is often presented as a small typed functional language, with structures as base values and data structures, functors as functions, and signatures as types. However, this module language departs significantly from most typed languages on one point: to typecheck a module expression containing a free structure identifier `S`, it does not suffice to know the signature (the type) of `S`; the actual structure (the value) bound to `S` is also needed in some cases. Consider the following code fragment:

```
... S.less 1 2 ...
```

where `S` is assumed to have the following signature:

```
S : sig type t; val less: t -> t -> bool end
```

The code fragment above is well-typed if `S` is bound to a structure that implements `t` as the type `int` of integers. But it is ill-typed if `S.t` has been implemented as another type. Both implementations of `S` satisfy the signature given above, though.

As shown by this example, signatures are not complete type specifications for structures: some information required to typecheck code that uses the structure is missing from the signature, and must be extracted from the structure itself. This is because type specifications in SML signatures are *transparent*: even if the signature only says `type t` without any indication on how `t` is implemented, the actual implementation of `t` "shows through" the signature.

This makes sense in the context of a toplevel-based system: because of static scoping, the user must provide a definition for S before being able to enter code that mentions S; hence the typechecker has access to the actual structure bound to S when typing expressions referring to S. This is no longer true in the context of separate compilation: S can be defined in a compilation unit A and used in another unit B, and A might not yet be written at the time we wish to typecheck and compile B. Hence, the fact that type specifications are transparent precludes Modula-2-style separate compilation, where program fragments are typed and compiled independently, relying only on their export interfaces.

In spite of these difficulties with separate compilation, transparent type specifications are an important feature of the SML module system, one that accounts for a large part of its expressive power. In the traditional view of structures as types equipped with operations over them [14], transparent type specifications makes it possible to add operations to a preexisting type, and apply these operations to preexisting values. A stricter interpretation of type specifications would generate a new type, incompatible with the original type, therefore compromising the usefulness of the additional operations. Consider for instance the following signature for a type equipped with a total ordering function:

```
datatype order = Less | Equal | Greater;
signature Order =
  sig
    type t
    val cmp: t -> t -> order
  end
```

We can define an Order structure for a base type such as int:

```
structure intOrder: Order =
  struct
    type t = int
    fun cmp i1 i2 =
      if i1 = i2 then Equal else
      if i1 < i2 then Less else Greater
  end
```

Since the type specification in Order is transparent, intOrder.t is compatible with int, hence intOrder.cmp can be applied to any integer. If type specifications were opaque, intOrder.t would be an abstract type, incompatible with any other type, and intOrder.cmp could not be applied to any value, making the structure useless.

Transparency also works across functors. Consider the following functor that takes an ordered type and produces an ordering over lists of elements of that type:

```
functor listOrder(base: Order): Order =
  struct
    type t = base.t list
    fun cmp [] [] = Equal
      | cmp [] _  = Less
      | cmp _  [] = Greater
      | cmp (h1::t1) (h2::t2) =
          case base.cmp h1 h2 of
            Equal => cmp t1 t2
          | c => c
  end
```

The application of listOrder to intOrder produces an Order structure whose type t is compatible with int list, hence whose cmp function can be applied to preexisting lists of integers. Again, functors such as listOrder would be useless without transparency.

2.2 Manifest types

So far, we have seen two interpretations of type declarations in signatures: the opaque interpretation, which supports separate compilation but is too restrictive, and the transparent interpretation, which is expressive enough but causes difficulties with separate compilation. We now propose a third approach, which combines expressiveness and separate compilation. We consider type declarations as opaque, but allow two kinds of type declarations: *abstract type declarations*, of the form type t, which give no clue on how t is implemented and therefore makes t incompatible with any other type (opaque interpretation); and *manifest type declarations*, of the form type t = τ, which require that t be implemented as the type expression τ, and therefore makes t compatible with τ.[1] This way, signatures become expressive enough to capture the required type equivalences, and there is no need to refer to the structures to establish these equivalences. Consider again the intOrder example above. The structure

```
structure intOrder =
  struct
    type t = int
    fun cmp i1 i2 =
      if i1 = i2 then Equal else
      if i1 < i2 then Less else Greater
  end
```

now has signature

```
intOrder: sig
            type t = int
            val cmp: t -> t -> order
          end
```

From this signature, we can deduce that intOrder.t and int are compatible; hence, the application of intOrder.cmp to integer values is well-typed. Notice that we have established this by looking at the signature only, but not at the actual structure bound to intOrder. We can show that intOrder.cmp 1 2 is well-typed even if intOrder is defined in another compilation unit and all we know about it is its signature, as provided by the interface of the unit.

Manifest types also work across functors. Consider again the listOrder functor above. With manifest types, we can define it as:

```
functor listOrder(base: Order):
  sig
    type t = base.t list
    val cmp: t -> t -> order
```

[1] We do not consider generative datatype declarations in signatures, since they can be viewed as declarations of abstract types plus injection and projection operations. For instance, the type specification sig type t = A | B of int end is equivalent to sig type t; val inj_A: t; val inj_B: int->t, val elim_t: t -> (unit->'a) -> (int->'a) -> 'a end.

```
  end
= struct
    type t = base.t list
    fun cmp l1 l2 = ...
  end
```

The result signature for `listOrder` makes it apparent that the component `t` in the result structure is compatible with `base.t list`, where `base` is the argument structure. This is a dependent function type: the type of the result depends on the value of the argument. Then, consider the application

```
structure intListOrder = listOrder(intOrder)
```

This application is well-typed, even though the signature of `intOrder` is different from `Order`, the argument signature of the functor: `Order` specifies `type t` (an abstract type) but the signature of `intOrder` says `type t = int` (a manifest type). However, a manifest type is a special case of an abstract type: we can always make a manifest type abstract by forgetting the additional information. We shall formalize this idea as a subtyping relation between signatures. This relation will show that the signature of `intOrder` is a subtype of `Order`, hence the application `listOrder(intOrder)` is well-typed.

According to the standard elimination rule for dependent function types (substitute the actual parameter for the formal parameter in the result type), the signature of `intListOrder` is:

```
intListOrder:
  sig
    type t = intOrder.t list
    val cmp: t -> t -> order
  end
```

From this signature, it follows that `intListOrder.t` is equivalent to `intOrder.t list`, and we already know that `intOrder.t` is equivalent to `int`. Since type equivalence is transitive and a congruence, it follows that `intListOrder.t` is equivalent to `int list`, which is the result we need to be able to apply `intListOrder.cmp` to integer lists. Again, we have reached the same conclusions as with the SML module system, but the reasoning is completely different: we have reasoned only at the level of signatures, while in SML we had to look inside structures.

2.3 Avoiding signature duplication

An apparent weakness of the approach presented above is the duplication of signatures: `intOrder`, `intListOrder` and the result of the `listOrder` functor all have different signatures, while in SML they share the same signature `Order`. Worse, the result signature for the `listOrder` functor cannot be declared and named before, since it depends on the argument of the functor.

To factor out the common parts between these signatures (the `val` declarations, usually), one solution is to introduce signatures parameterized by type expressions:

```
signature ManifestOrder(type τ) =
  sig type t = τ; val cmp: t -> t -> order end
```

so that the signature of `intOrder` is `ManifestOrder(int)`, and `listOrder` can be declared as:

```
functor listOrder(base: Order):
  ManifestOrder(base.t) = ...
```

The remaining problem is that the generic `Order` signature, with `t` left abstract, cannot be obtained by application of `ManifestOrder` and must therefore be declared separately.

Another approach is to introduce the notation `Order with type t = τ` as syntactic sugar for the signature `Order` where the specification of `t` is replaced by `type t = τ`, that is:

```
sig type t = τ; val cmp: t -> t -> order end
```

This style of "after the fact" parameterization, reminiscent of SML's syntax for sharing constraints, makes it possible to write the signature only once and use it in both abstract and manifest contexts. (Tofte [26] has proposed a similar notation to express type abbreviations in signatures, though for different purposes.)

The `with` construct is just a notational convenience: it can always be expanded before typing as described above, as long as signatures can be named but not abstracted over nor stored in structures. A typechecker would certainly avoid this expansion for the sake of efficiency, but the point is that the `with` construct does not complicate the formalism.

This is no longer true if signatures can appear as structure components or as functor parameters: if `S` is a functor parameter, `S with type t = τ` cannot be expanded before typing. In this context, the unrestricted `with` construct seems to require a type system similar to those for polymorphic extensible records [6]. A more reasonable alternative is to restrict `with` to situations where the left-hand side can be statically reduced to a `sig ... end` expression.

2.4 Sharing constraints for free!

So far, we have seen that manifest types in toplevel position or functor result position can replace SML's transparent type specifications. We shall now see that manifest types in functor argument position can replace SML's sharing constraints. The idea is that a functor of the form

```
functor F
   (structure S1: sig type t; ... end
    structure S2: sig type t = S1.t; ... end) ...
```

can only be applied to structures `S1` and `S2` for which we can prove that `S1.t` is the same type as `S2.t`—just like the corresponding SML functor with a sharing constraint:

```
functor F
   (structure S1: sig type t; ... end
    structure S2: sig type t; ... end
    sharing type S1.t = S2.t) ...
```

Sharing constraints are an essential feature of the SML module system: they guarantee that a functor combining operations from several structures will only be applied to consistent sets of structures—typically, structures derived from one common structure by addition of operations. This programming situation, known as the "diamond import problem" [15], arises often in practice. The following "diamond import" example shows that manifest types suffice to express and check the required sharing properties. We start by a structure implementing some abstract data type, say, integer lists:

```
signature Intlist =
  sig
    type t
    val nil: t
    val cons: int -> t -> t
  end
```

Then, we define two functors that take an `Intlist` structure and equip its type `t` with derived operations.

```
signature Interval =
  sig type t; val interval: int -> int -> t end

functor interval(intlist: Intlist):
        Interval with type t = intlist.t
= struct
    type t = intlist.t;
    fun interval i j = ...
  end

signature Sumlist =
  sig type t; val sumlist: t -> int end

functor sumlist(intlist: Intlist):
        Sumlist with type t = intlist.t
= struct
    type t = intlist.t;
    fun sumlist l = ...
  end
```

Finally, we define a functor that combines the structures returned by the functors `interval` and `sumlist`.

```
functor main
   (structure i: Interval
    structure s: Sumlist with type t = i.t)
= struct
    fun f n = s.sumlist(i.interval 1 n)
  end
```

The application of `s.sumlist` to the result of `i.interval` is well-typed because the signature of `s` guarantees that the types `s.t` and `i.t` are compatible. Now, we can show that the application

```
main(interval(list) sumlist(list))
```

is well-typed, given a structure `list` of type `Intlist`. First, the signature of `interval(list)` is

```
interval(list): Interval with type t = list.t
```

which is included in the expected signature for `i` in `main`. Then, following the typing rule for functor application, we substitute the actual parameter `interval(list)` for the formal parameter `i` in the remainder of the functor arguments:

```
s: Sumlist with type t = interval(list).t
```

We must now prove that the signature of the second argument:

```
sumlist(list): Sumlist with type t = list.t
```

is included in the signature for `s`. According to the subtyping rules in section 3, this amounts to showing that the types `list.t` and `interval(list).t` are identical. This immediately follows from the signature of `interval(list)`; again, only the signature is used. Hence the application of `main` is well-typed. On the other hand, we will correctly reject applications of `main` to inconsistent `i` and `s` structures, such as

```
main(interval(list) sumlist(list2))
```

where `list2` is another implementation of `Intlist` with a type `t` incompatible with `list.t`. Typing proceeds as above, but fails because `interval(list).t` and `list2.t` are not compatible, hence the signature of `sumlist(list2)` is not included in the signature specified for `s`.

Notice that we have checked the sharing constraint using only the general rules for subtyping and functor application: no special typing rule is required—at least for this simple diamond import problem; section 3.4 shows that an additional "type strengthening" rule is sometimes necessary to establish the expected sharing properties.

2.5 Expressible sharing constraints

The sharing constraints expressible with manifest types are less general than those expressible in the SML module system. First, manifest types can only express constraints of the form *type identifier = type expression*, which are both asymmetrical and local (a constraint over a type `t` must appear in the signature that declares `t`). In contrast, SML allows sharing constraints of the form *long identifier = long identifier* (e.g. `p.t = q.x.t`), more symmetrical and non-local. This difference is mostly cosmetic, however: SML-style sharing constraints can be compiled into manifest types by choosing a representative for each equivalence class of shared types, and pushing the constraints down the constrained signatures.

A more substantial difference is that manifest types can only express the equality of two types, while SML sharing constraints can also express the equality of two structures. Manifest types can account for the most common use of sharing constraints over structures: to specify sharing between all type components of two structures in a compact way. A more advanced use of sharing constraints over structures is to ensure that the value components of the structures are also identical, which is useful to deal with structures that have a local state [11]. This can be encoded to some extent in our calculus, by introducing an abstract type to act as a structure stamp. For instance, the SML specification

```
functor F
 (structure A: sig val r: int ref ... end
  structure B: sig val r: int ref ... end
  sharing A = B)
```

becomes

```
functor F
 (structure A: sig type stamp;
                   val r: int ref ... end
  structure B: sig type stamp = A.stamp;
                   val r: int ref ... end)
```

If the `stamp` type fields are abstract types in all structures, then the equality of stamp types guarantees the equality of the structures, by generativity of abstract types. This relies on programmer's discipline, however; hence the type system cannot infer that all components of these two structures are

themselves shared. On the other hand the absence of sharing constraints over structures greatly simplifies the formalism: since structures have no "identity", there is no need to represent them by unique stamps, as in [17]; simple record-like terms suffice.

2.6 The problem with type abbreviations in signatures

Manifest types are similar to an often proposed extension of SML called "type abbreviations in signatures". This extension has been excluded from the Standard because it is known to cause serious difficulties [16]: if type abbreviations are allowed in signatures, signature elaboration becomes undecidable. It is worth pointing out that this problem is not inherent to type abbreviations in signatures, but stems from their interaction with sharing constraints. In the simple approach suggested in [16], sharing constraints may involve abbreviated type constructors, as in:

```
sig
  type t = τ
  type s = σ
  sharing type t = s
end
```

In this approach, sharing constraints are therefore no longer restricted to equalities between type constructors: they can now express arbitrary equations between type expressions ($\tau = \sigma$ in the example above). Since type equations may involve abstract type constructors (as in `int t = int` where `t` is declared as `type 'a t`), second-order unification is required to elaborate these sharing constraints.

Our approach avoids this difficulty: since sharing constraints are expressed in terms of manifest types, all expressible sharing constraints are of the format *long identifier* = *type expression*, where *long identifier* refers to an abstract type. Hence there is no way to equate two arbitrary type expressions. For instance, the pathological signature given above is not expressible in our system: assuming `t` is chosen as representative for the equivalence class of `s` and `t`, then `s` would have to be declared as equal to σ and equal to `t` also, which is syntactically impossible.

3 A calculus of modules

We now formalize the ideas presented above in a simple module calculus built on top of a typed base language.

3.1 Syntax

In the following grammar, v ranges over value names, t over type names and x over module names. Identifiers v_i, t_i and x_i are composed of a name plus a stamp i taken from some infinite set of stamps.

Stamps are used to distinguish identifiers having the same name. We cannot allow arbitrary renamings on identifiers, since the calculus relies on the names to extract structure fields. Instead, we will use renamings that only change the stamp parts of identifiers, but preserve the name parts of identifiers. This causes no difficulties with structure access, since access is by name, not by name plus stamp.

Stamps are needed only during typechecking. In particular, they can be omitted from program texts, since they can be recovered by applying the standard scoping rules (each binding generates a new stamp, each reference to an identifier is given the stamp of its most recent binding). We will follow this convention to make examples more legible.

Value expressions:

$e ::= v_i$	value identifier
$\mid p.v$	value component of a structure
$\mid \ldots$	depends on the base language

Type expressions:

$\tau ::= t_i$	type identifier
$\mid p.t$	type component of a structure
$\mid \ldots$	depends on the base language

Module expressions:

$m ::= x_i$	module identifier
$\mid p.x$	module component of a structure
$\mid$ `struct` s `end`	structure construction
$\mid$ `functor`$(x_i : M)\, m$	functor
$\mid m_1(m_2)$	functor application

Module types:

$M ::=$ `sig` S `end`	signature type
$\mid$ `functor`$(x_i : M)\, M'$	dependent function type

Structure body:

$s ::= \emptyset \mid s_c; s$

Structure components:

$s_c ::=$ `val` $v_i = e$	value binding
$\mid$ `type` $t_i = \tau$	type binding
$\mid$ `module` $x_i : M = m$	module binding

Signature body:

$S ::= \emptyset \mid S_c; S$

Signature components:

$S_c ::=$ `val` $v_i : \tau$	value declaration
$\mid$ `type` t_i	abstract type declaration
$\mid$ `type` $t_i = \tau$	manifest type declaration
$\mid$ `module` $x_i : M$	module declaration

Access paths:

$p ::= x_i \mid p.x$

Typing environments:

$E ::= \emptyset \mid E; S_c$

Terms are identified up to alpha-conversion. The binding constructs are `functor` (with scope the functor result part) and `val`, `type` and `module` (with scope the remainder of the structure or signature). Alpha-conversion can rename the stamp part of identifiers, but is required to preserve the name part. The components of a structure or signature are assumed to have distinct names.

The base language

The base language (value and type expressions) is left mostly unspecified, since the module calculus makes few assumptions about it and should accommodate a variety of base languages. (We have experimented with two base languages: ML and a more Algol-like language derived from [22].) The base language can access values and types bound earlier in the same structure (v_i and t_i). It can also refer to value and