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.

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` $= \tau$ as syntactic sugar for the signature `Order` where the specification of `t` is replaced by `type t` $= \tau$, 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` $= \tau$ 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 type components of other structures through the "dot notation" $p.v$ and $p.t$ where p is an access path to a structure with a v_i or t_i component.

The module language

The module language has both structures and functors as first-class module values. We use the word "module" to refer to structures and functors. Functors live at the same level as structure, unlike in Tofte's system [25]. The module language is actually lambda-calculus with one data structure: generalized products. Its dynamic semantics is given by a straightforward translation to untyped lambda-calculus with products, by erasing the type components in structures.

Structures are sequences of bindings for values, types and modules. To keep this paper simple, we do not formalize module type bindings (`signature X = sig ... end`). Simple uses of module type bindings (as in the examples of section 2) can be translated by duplicating module type expressions. Introducing module type bindings as structure components is tempting, as it brings considerable expressive power to the module system: polymorphic modules and even F_ω-like module type operators are definable. For instance, the polymorphic module $\Lambda X.m$ would be expressed as the functor

$$\texttt{functor(x: sig signature X end)}\ (m\{X \leftarrow \texttt{x.X}\}).$$

However, the implications of introducing module types as structure components are not clear yet, especially with respect to decidability of typechecking [10].

Access paths

The main singularity of this calculus is the restriction to paths ($p.v$, $p.x$, $p.t$) when accessing structure components, instead of a more general projection construct ($m.v$, $m.x$, $m.t$) that could be applied to any module expression m, as in DL and XML [15, 12]. For instance, our calculus allows $x_i.t$ in the scope of the binding $x_i = m_1(m_2)$, but not directly $m_1(m_2).t$.

The reason why general projections (and even projections restricted to values, as in Harper and Lillibridge [10]) are inadequate is that we have abstract types and therefore must account for type generativity. For instance, assuming f is a functor returning a structure containing an abstract type t, then two applications of f to a structure m must return two different types t. With general projections, we would be unable to determine whether the two types $f(m).t$ and $f(m).t$ are compatible (if the two occurrences of $f(m)$ correspond to the same application of f) or incompatible (if these are distinct applications of f).

This problem disappears if we restrict projections to paths, and put suitable restrictions on rebindings [5]. Then, the two types $p.t$ and $p.t$ are always compatible, because the two occurrences of p are guaranteed to refer to the same structure: paths do not contain functor applications, hence their evaluation cannot create new types. Similarly, the two types $p.t$ and $p'.t$ where $p \neq p'$ are incompatible (assuming t and t' are abstract types, not manifest types), because p and p' are assumed to be bound to different structures. In other terms, we rely on name equivalence to account for generativity.

3.2 Typing rules

We now give an overview of the typing rules, which assign module types to module expressions ($E \vdash m : M$) and signatures to structures ($E \vdash s : S$). The typing rules for module expressions are mostly standard:

$$E \vdash x_i : E(x_i) \qquad \frac{E \vdash m : M' \quad E \vdash M' <: M}{E \vdash m : M}$$

$$\frac{E \vdash s : S}{E \vdash (\texttt{struct}\ s\ \texttt{end}) : (\texttt{sig}\ S\ \texttt{end})}$$

$$\frac{E \vdash M \text{ module type} \quad x_i \notin \mathrm{Dom}(E) \quad E; \texttt{module}\ x_i : M \vdash m : M'}{E \vdash \texttt{functor}(x_i : M)m : \texttt{functor}(x_i : M)M'}$$

$$\frac{E \vdash m_1 : \texttt{functor}(x_i : M)M' \quad E \vdash m_2 : M}{E \vdash m_1(m_2) : M'\{x_i \leftarrow m_2\}}$$

The application rule is the usual elimination rule for dependent function types. Because only paths are allowed in projections, the substitution $M'\{x_i \leftarrow m_2\}$ is undefined if m_2 is not a path and x_i occurs in M'. In this case, the application $m_1(m_2)$ is ill-typed; an intermediate binding of m_2 to a module identifier must be introduced. Felleisen and Sabry's A-normalization [23] can be used to introduce these bindings in a systematic way before typechecking.

The most unusual rule is the rule for module access:

$$\frac{E \vdash p : (\texttt{sig}\ S_1; \texttt{module}\ x_i : M;\ S_2\ \texttt{end})}{E \vdash p.x : M\{n_i \leftarrow p.n \mid n_i \in \mathrm{Dom}(S_1)\}}$$

Here, n ranges over all three kinds of names. In the premise, we consider the path p as a special case of module expression. The rule says that p in $p.x$ must refer to a structure with a module component named x; the type for this component gives the type for $p.x$. However, the type found in the signature may refer to identifiers bound earlier in the signature, as in

```
p: sig type t_j
       module x_i: sig val v_k: t_j end
   end
```

These identifiers must be prefixed by p when the type of x is extracted from the signature. In the example above, this gives $p.x$ the correct type `sig val` $v_k : p.t$ `end`, with $p.t$ in place of t_j.

The typing of structures is straightforward. Structures are dependent products, hence a binding must be pushed in the environment before typing the following bindings.

$$E \vdash \emptyset : \emptyset$$

$$\frac{E \vdash e : \tau \quad v_i \notin \mathrm{Dom}(E) \quad E; \texttt{val}\ v_i : \tau \vdash s : S}{E \vdash (\texttt{val}\ v_i = e;\ s) : (\texttt{val}\ v_i : \tau;\ S)}$$

$$\frac{E \vdash \tau \text{ definable type} \quad t_i \notin \mathrm{Dom}(E) \quad E; \texttt{type}\ t_i = \tau \vdash s : S}{E \vdash (\texttt{type}\ t_i = \tau;\ s) : (\texttt{type}\ t_i = \tau;\ S)}$$

$$\frac{E \vdash m : M \qquad x_\imath \notin \mathrm{Dom}(E) \qquad E; \texttt{module}\ x_\imath : M \vdash s : S}{E \vdash (\texttt{module}\ x_\imath : M = m;\ s) : (\texttt{module}\ x_\imath : M;\ S)}$$

We assume given typing judgements for the base language, $E \vdash e : \tau$ and $E \vdash \tau$ definable type, to assign types to expressions and to check the well-formedness of type expressions, respectively. (In the case of ML, the former is defined by Damas and Milner's type system [9] with the extra requirement that τ is a closed type scheme, and the latter checks that τ is a closed simple type and that all external types $p.t$ in τ are valid.)

The rules above make all type components manifest in the inferred signature. They can be abstracted later, if desired, using a module type constraint.

The rules require identifiers to be renamed so that they are bound at most once. Rebindings lead to incorrect typings in conjunction with name equivalence over paths, as shown by the following example:

```
module x : sig type t; val v:t end
         = struct type t=int; val v=3 end;
val u = x.v;
module x : sig type t; val v:t end
         = struct type t=bool; val v=true end;
val w = x.v;
```

If both modules x have the same stamp, say, $\texttt{x}_\imath$, then the values u and w have the same type $\texttt{x}_\imath$.t, but one is an integer and the other a boolean. The side conditions in the rules above guarantee that the two x modules have different stamps, hence that the types of u and v are incompatible.

3.3 Type inclusion and equivalence

Type equivalence, written $\approx$, takes into account the type equations encoded in manifest type specifications:

$$E_1; \texttt{type}\ t_\imath = \tau; E_2 \vdash t_\imath \approx \tau$$

$$\frac{E \vdash p : \texttt{sig}\ S_1; \texttt{type}\ t_\imath = \tau; S_2\ \texttt{end}}{E \vdash p.t \approx \tau\{n_\imath \leftarrow p.n \mid n_\imath \in \mathrm{Dom}(S_1)\}}$$

The remaining rules for type equivalence depend on the base language considered (For ML, they consist of the usual transitivity and congruence rules.) Inclusion between types ($E \vdash \tau <: \tau'$) is base language-dependent; the only assumption is that type equivalence implies type inclusion. (For ML, subtyping is subsumption between type schemes.)

The inclusion rules between module types are standard. Functor types are contravariant in their domain.

$$\frac{E \vdash M_2 <: M_1 \qquad E; \texttt{module}\ x_\imath : M_2 \vdash M_1' <: M_2'}{E \vdash \texttt{functor}(x_\imath : M_1)M_1' <: \texttt{functor}(x_i : M_2)M_2'}$$

$$\frac{E \vdash S <: S'}{E \vdash \texttt{sig}\ S\ \texttt{end} <: \texttt{sig}\ S'\ \texttt{end}}$$

Inclusion between signatures is defined as follows:

$$E \vdash \emptyset <: \emptyset$$

$$\frac{E \vdash S_c <: S_c' \qquad E; S_c \vdash S <: S'}{E \vdash S_c; S <: S_c'; S'} \qquad \frac{E; S_c \vdash S <: S'}{E \vdash S_c; S <: S'}$$

The rightmost rule allows skipping some components of the richer signature if they have no counterpart in the simpler signature. Inclusion between signature components is defined in the obvious way: val types and module module types must be properly included; manifest type specifications are included in abstract type specifications; and two manifest type specifications are included if and only if the manifest types are equivalent.

$$\frac{E \vdash \tau <: \tau'}{E \vdash \texttt{val}\ v_\imath : \tau <: \texttt{val}\ v_\imath : \tau'}$$

$$\frac{E \vdash M <: M'}{E \vdash \texttt{module}\ x_\imath : M <: \texttt{module}\ x_\imath : M'}$$

$$E \vdash \texttt{type}\ t_\imath = \tau <: \texttt{type}\ t_\imath \qquad E \vdash \texttt{type}\ t_\imath <: \texttt{type}\ t_\imath$$

$$\frac{E \vdash \tau \approx \tau'}{E \vdash \texttt{type}\ t_\imath = \tau <: \texttt{type}\ t_\imath = \tau'}$$

In effect, the inclusion $S_1 <: S_2$ checks that all components in S_2 are present in S_1, possibly with more general types, and possibly interspersed with other components. To keep the rules simple, the components common to S_1 and S_2 must appear in the same order; in practice, it would be desirable to allow permutations of independent components.

Whenever we skip or retain a component of S_1, it is added to the environment for comparing the remainders of S_1 and S_2. This is useful if the component is a manifest type (the type equation might be needed to establish an inclusion later) or a module (whose module type can contain manifest types).

Combining the inclusion rules for signatures and the typing rules for structures, we can type the following three typical examples:

```
(1) struct type t=int; val x=1 end :
                  sig type t; val x:t end
(2) struct type t=int; val x=1 end :
                  sig type t=int; val x:t end
(3) struct type t=int; val x=(1:t) end :
                  sig val x:int end
```

(1) corresponds to an abstract type with associated operations being implemented as some specific type (here, int). (2) is similar, but the type is exported concretely, with its implementation. (3) corresponds to a local type declaration, which is not exported but is taken into account for signature matching.

3.4 Sharing constraints and type strengthening

The calculus presented above contains no special rule for checking sharing constraints at functor applications. The general inclusion and application rules can show that sharing constraints are satisfied in many situations, especially those deriving from the "diamond import" situation. However, they are not always sufficient. First, they are sensitive to the order in which sharing constraints are written: the curried functor

```
module f =
  functor (a: sig type t; ... end)
  functor (b: sig type t = a.t; ... end) ...
```

cannot be applied to structures with the following signatures:

```
a : sig type t = b.t; ... end
b : sig type t; ... end
```

because the signature of `b` is not included in `sig type t = a.t; ... end`, even though `a.t` and `b.t` are known to be equivalent. For the same reasons, the rules fail to recognize that a structure always shares with itself: the functor application `g(c)(c)`, with `g` declared as

```
module g =
  functor (a: sig type t end)
  functor (b: sig type t = a.t end) ...
```

fails if `c.t` is abstract. In both cases, the problem is that we compare signatures without taking into account that they are not signatures of arbitrary structures, but signatures of a given path. Fortunately, this fact can be expressed by introducing more manifest types in the signature: whenever a path p has type

$$p : \texttt{sig type } t_i; \ldots \texttt{ end},$$

it also has type

$$p : \texttt{sig type } t_i = p.t; \ldots \texttt{ end}.$$

If we apply this transformation to the types of `b` and `c` in the examples above, the subtyping rules now recognize that the sharing constraints are satisfied. For instance, when typing the application `f(a)(b)` above, we have to show that `b` has type

```
sig type t = a.t; ... end
```

and this can be done by first considering `b` with type `sig type t = b.t; ... end`, then proving `b.t = a.t` from the type of `a`.

This operation on path types is called *strengthening*. The corresponding typing rule is:

$$\frac{E \vdash p : M}{E \vdash p : M/p}$$

where the strengthening operation M/p is defined by:

$$\begin{array}{rcl}
(\texttt{sig } S \texttt{ end})/p & = & \texttt{sig } S/p \texttt{ end} \\
M/p & = & M \text{ if } M \text{ is not a signature} \\
(\texttt{val } v_i : \tau;\ S)/p & = & \texttt{val } v_i : \tau;\ S/p \\
(\texttt{type } t_i;\ S)/p & = & \texttt{type } t_i = p.t;\ S/p \\
(\texttt{type } t_i = \tau;\ S)/p & = & \texttt{type } t_i = p.t;\ S/p \\
(\texttt{module } x_i : M;\ S)/p & = & \texttt{module } x_i : M/p.x;\ S/p
\end{array}$$

Even in the cases for manifest types, no information is lost by strengthening: if we replace `type` $t_i = \tau$ by `type` $t_i = p.t$ in the type of p, we can still show that the t component is equivalent to τ, by looking up the original type of p in the typing environment. This remark can be formalized as follows: if p has type M in the environment E, then M/p is a subtype of M in E. Hence, it is always safe to apply the strengthening rule before checking type inclusion.

Type strengthening is also useful when taking multiple views of a structure while preserving type compatibility between the views. Assume `x` is bound to a structure with type

```
x : sig type t; val f: τ; val g: σ end
```

and we wish to view `x` without the `g` component. If we define the view as

```
module y : sig type t; val f: τ end = x
```

then, by generativity, `y.t` is a "new" type, incompatible with `x.t`, since `x` and `y` are not the same path. To ensure compatibility between `x.t` and `y.t`, we must make `t` manifest in the signature of `y`:

```
module y : sig type t = x.t; val f: τ end = x
```

Checking the type-correctness of this binding requires the strengthening rule: without prior strengthening, the signature of `x` would not be a subtype of the signature declared for `y`.

3.5 Type inference

The typing rules presented above do not lead directly to a type checking or type inference algorithm, since the rules for subsumption and strengthening are not syntax-directed. However, the applications of these rules can be "floated downwards" and combined with the functor application and module binding rules. More precisely, we define a syntax-directed variant of the calculus above by removing the subsumption and strengthening rules, and replacing the rules for functor application and module binding by:

$$\frac{E \vdash m_1 : \texttt{functor}(x_i : M_1)M \qquad E \vdash m_2 : M_2 \qquad E \vdash M_2/m_2 <: M_1}{E \vdash m_1(m_2) : M\{x_i \leftarrow m_2\}}$$

$$\frac{E \vdash m : M' \qquad E \vdash M'/m <: M \qquad x_i \notin \mathrm{Dom}(E) \qquad E; \texttt{module } x_i : M \vdash s : S}{E \vdash (\texttt{module } x_i : M = m;\ s) : (\texttt{module } x_i : M;\ S)}$$

Here, we write M/m for M if m is not a path, and M/p if m is a path p. The resulting system has the following correctness and completeness properties:

1. if $E \vdash m : M$ in the syntax-directed system, then $E \vdash m : M$ in the original system;
2. if $E \vdash m : M$ in the original system, then there exists M' such that $E \vdash m : M'$ in the syntax-directed system and $E \vdash M'/m <: M$.

Moreover, the type equivalence and type subsumption relations between module types are obviously decidable (by structural induction on the type expressions) as soon as the corresponding relations on base types are decidable. The decidability problems encountered in Harper and Lillibridge's system [10] are avoided here because we do not have module types as structure components.

From these remarks, we easily obtain a type inference algorithm which, given an environment E and a module expression m, either returns the most general type of m in E, or fails if m is ill-typed in E. At the level of the core language, it assumes given algorithms to infer the principal type of a value expression and to check equality and subsumption between core type expressions. The algorithms for the core language must take into account the extra type equalities induced by the manifest types in the current typing environment. In the case of ML, the principal type property and the existence of a type inference algorithm still hold when equations between types are introduced to take into account manifest type declarations. This follows from Rémy's general results on type inference modulo an equational theory [21].

4 Expressiveness

The aim of this section is to show that our calculus (with opaque signatures and propagation of type equalities through manifest types) is at least as expressive as the SML module system (with transparent signatures and propagation of type equalities through structures). Ideally, we should present a type-preserving encoding of a significant fragment of SML (e.g. without structure sharing) into our calculus. Unfortunately, the size and complexity of the SML definition [17] are such that defining this encoding and reasoning about it is hopeless. Instead, we will start from a much simpler calculus based on strong sums, similar to MacQueen's DL calculus and Harper and Mitchell's XML calculus [15, 12]. This calculus accounts for the basic features of the SML module system except generativity and sharing.

To keep the encoding simple, the target of the encoding will not be the module system presented in the previous section, but a simpler calculus, closer in syntax to the strong sums calculus, but with weak sums and manifest types instead of strong sums. The target calculus accounts for most of the features of our module system except generativity. The fragment of the target calculus actually used by the translation can easily be encoded into the system of section 3.

4.1 Strong sums and manifest sums

The source and target calculi have the following syntax:

Terms:

$$\begin{aligned} m ::=\ & x \mid \lambda x{:}M.\, m \mid m_1(m_2) \\ & \mid [t = \tau,\, m] \mid \mathtt{ops}(m) \\ & \mid \langle x = m_1,\, m_2\rangle \mid \mathtt{fst}(m) \mid \mathtt{snd}(m) \end{aligned}$$

Simple types:

$$\tau ::= \mathtt{int} \mid \mathtt{bool} \mid \ldots \mid \tau_1 \rightarrow \tau_2 \mid t \mid \mathtt{typ}(m)$$

Module types:

$$M ::= \tau \mid \exists t.\, M \mid \exists t = \tau.\, M \mid \Sigma x{:}M_1.\, M_2 \mid \Pi x{:}M_1.\, M_2$$

Structures are replaced by two simpler constructs, $[t = \tau,\, m]$ to package a type τ with a module m, and $\langle x = m_1,\, m_2\rangle$, which is the dependent pair of two modules. Access in packages and pairs is by position (projections `typ` and `ops` for packages, `fst` and `snd` for pairs) instead of by field names. Similarly, signatures are replaced by existential types (for packages) and Σ-types (for dependent pairs). Functors are presented by λ-abstractions and Π-types (dependent function types).

A difference with DL and XML is that we keep existential types distinct from Σ-types, instead of injecting simple types into module terms, which turns packages into special cases of pairs and existential types into special cases of Σ-types. In our system, we have manifest existential types but no manifest Σ-types, hence we cannot identify these two notions.

The base language used here is simply-typed lambda-calculus (with constants represented as predefined identifiers). Instead of explicitly injecting the base language into the module language, we simply consider the base language as a subset of the module language: λ-abstraction and application at the level of the base language are merged with λ-abstraction and application at the level of modules. This makes no semantic difference [12] but further simplifies the calculus. The calculus is still stratified at the type level, however: base-language functions and module-level functors have distinct types, and existentially quantified type variables range over simple types, not over module types.

The dynamic semantics for the calculus are given by the following reduction rules:

$$\begin{aligned} (\lambda x{:}M.\, m)(m') &\rightarrow m\{x \leftarrow m'\} \\ \mathtt{typ}[t = \tau,\, m] &\rightarrow \tau \\ \mathtt{ops}[t = \tau,\, m] &\rightarrow m\{t \leftarrow \tau\} \\ \mathtt{fst}\langle x = m_1,\, m_2\rangle &\rightarrow m_1 \\ \mathtt{snd}\langle x = m_1,\, m_2\rangle &\rightarrow m_2\{x \leftarrow m_1\} \\ \Gamma[m] &\rightarrow \Gamma[m'] \text{ if } m \rightarrow m', \text{ for any context } \Gamma \end{aligned}$$

We write $\overset{*}{\rightarrow}$ for the reflexive and transitive closure of the reduction rules.

Figure 1 shows the typing rules for the two variants of the calculus that will serve as source and target for the translation. The first variant, named S for "strong sums", does not use manifest existential types $\exists t = \tau.\, M$ and treats packages as transparent. The second variant, named M for "weak sums with manifest types", uses manifest existential types to propagate type equalities and treats packages as opaque.

The difference between S and M is apparent in the type equivalence rules for $\mathtt{typ}(m)$. In M, if m does not have a manifest existential type, the type expression $\mathtt{typ}(m)$ is only equivalent to $\mathtt{typ}(m')$ where m and m' are syntactically identical. In S, rule 6 says that $\mathtt{typ}(m)$ is equivalent to the type part of whatever package m reduces to.

The introduction rules for Σ differ accordingly. In S, the second component m_2 of the pair $\langle x = m_1,\, m_2\rangle$ is typed after textual substitution of x by m_1, so that m_2 can rely on specific implementations of abstract types in m_1. In M, only the type of m_1, not m_1 itself, is taken into account for the typing of m_2.

The system M also has subsumption and strengthening rules similar to those of the full module calculus. Strengthening M/m is here defined as:

$$\begin{aligned} \tau/m &= .\ \tau \\ (\exists t.\, M)/m &= \exists t = \mathtt{typ}(m).\, M/\mathtt{ops}(m) \end{aligned}$$

$$E \vdash x : E(x) \quad (1)$$

$$\frac{E, x : \tau_1 \vdash m : \tau_2}{E \vdash \lambda x{:}\tau_1.\, m : \tau_1 \to \tau_2} \qquad \frac{E \vdash m_1 : \tau \to \tau' \quad E \vdash m_2 : \tau}{E \vdash m_1(m_2) : \tau'}$$

$$\frac{E, x : M_1 \vdash m : M_2}{E \vdash \lambda x{:}M_1.\, m : \Pi x{:}M_1.\, M_2} \qquad \frac{E \vdash m_1 : \Pi x{:}M.\, M' \quad E \vdash m_2 : M}{E \vdash m_1(m_2) : M'\{x \leftarrow M\}} \quad (2)$$

$$\frac{E \vdash_s m\{t \leftarrow \tau\} : M\{t \leftarrow \tau\}}{E \vdash_s [t = \tau,\, m] : \exists t.\, M} \quad (3) \qquad \frac{E \vdash_m m\{t \leftarrow \tau\} : M\{t \leftarrow \tau\}}{E \vdash_m [t = \tau,\, m] : \exists t = \tau.\, M}$$

$$\frac{E \vdash_s m_1 : M_1 \quad E \vdash_s m_2\{x \leftarrow m_1\} : M_2\{x \leftarrow m_1\}}{E \vdash_s \langle x = m_1,\, m_2\rangle : \Sigma x{:}M_1.\, M_2} \quad (4) \qquad \frac{E \vdash_m m_1 : M_1 \quad E, x : M_1 \vdash_m m_2 : M_2}{E \vdash_m \langle x = m_1,\, m_2\rangle : \Sigma x{:}M_1.\, M_2}$$

$$\frac{E \vdash m : \exists t.\, M}{E \vdash \mathtt{ops}(m) : M\{t \leftarrow \mathtt{typ}(m)\}} \qquad \frac{E \vdash m : \Sigma x{:}M_1.\, M_2}{E \vdash \mathtt{fst}(m) : M_1} \qquad \frac{E \vdash m : \Sigma x{:}M_1.\, M_2}{E \vdash \mathtt{snd}(m) : M_2\{x \leftarrow \mathtt{fst}(m)\}}$$

$$\frac{E \vdash_s m : M \quad E \vdash_s M \approx M'}{E \vdash_s m : M'} \quad (5) \qquad \frac{E \vdash_s m : M \quad E \vdash_s M <: M'}{E \vdash_s m : M'} \qquad \frac{E \vdash_m m : M}{E \vdash_m m : M/m}$$

$$\frac{E \vdash_s m : \exists t.\, M \quad m \xrightarrow{*} [t = \tau,\, m']}{E \vdash_s \mathtt{typ}(m) \approx \tau} \quad (6) \qquad \frac{E \vdash_m m : \exists t = \tau.\, M}{E \vdash_m \mathtt{typ}(m) \approx \tau} \quad (7) \qquad \frac{E \vdash_m M \approx M'}{E \vdash_m M <: M'}$$

$$E \vdash_m \exists t = \tau.\, M <: \exists t.\, M$$

$$\frac{E \vdash_m M <: M'}{E \vdash_m \exists t.\, M <: \exists t.\, M'} \qquad \frac{E \vdash_m \tau \approx \tau' \quad E \vdash_m M\{t \leftarrow \tau\} <: M'\{t \leftarrow \tau'\}}{E \vdash_m \exists t = \tau.\, M <: \exists t = \tau'.\, M'}$$

$$\frac{E \vdash_m M_1 <: M_1' \quad E, x : M_1 \vdash M_2 <: M_2'}{E \vdash_m \Sigma x{:}M_1.\, M_2 <: \Sigma x{:}M_1'.\, M_2'} \qquad \frac{E \vdash_m M_1' <: M_1 \quad E, x : M_1' \vdash M_2 <: M_2'}{E \vdash_m \Pi x{:}M_1.\, M_2 <: \Pi x{:}M_1'.\, M_2'}$$

Plus the standard congruence, transitivity and symmetry rules for $\approx$, and transitivity rule for $<:$

Figure 1: Typing rules for strong sums (S) and weak sums with manifest existentials (M). Left: S-specific rules; right: M-specific rules; center: common rules. $\vdash$ stands for either $\vdash_s$ or $\vdash_m$.

$$\begin{aligned}
(\exists t = \tau.\, M)/m &= \exists t = \mathtt{typ}(m).\, M/\mathtt{ops}(m) \\
(\Sigma x{:}M_1.\, M_2)/m &= \Sigma x{:}M_1/\mathtt{fst}(m).\, M_2/\mathtt{snd}(m) \\
(\Pi x{:}M_1.\, M_2)/m &= \Pi x{:}M_1.\, M_2/m(x)
\end{aligned}$$

4.2 The first-order fragment

We now show that the first-order fragment of S, the calculus with strong sums, is included in the first-order fragment of M, the calculus with weak sums and manifest existentials. By "first-order", we mean that functors cannot take functors as arguments, as in SML. Unlike SML, we will still allow functors as structure components, as long as such structures are not passed to functors. Consider the following subsets of module types:

Functor argument types:
$F ::= \tau \mid \exists t.\, F \mid \Sigma x{:}F.\, F$

Abstract first-order types:
$A ::= \tau \mid \exists t.\, A \mid \Sigma x{:}A.\, A \mid \Pi x{:}F.\, A$

Concrete first-order types:
$C ::= \tau \mid \exists t = \tau.\, C \mid \Sigma x{:}C.\, C \mid \Pi x{:}F.\, C$

We are going to show that any term that is typable in S using only A types is also typable in M using C types. Moreover, the C types used in the M derivation correspond, in a sense to be made precise below, to the A types used in the S derivation.

The correspondence between A and C types is captured by the following "stripping" operation, written $\overline{C}$, which removes all manifest type information from a C type, turning it into an A type:

$$\begin{aligned}
\overline{\tau} &= \tau \\
\overline{\exists t = \tau.\, C} &= \exists t.\, \overline{C} \\
\overline{\Sigma x{:}C_1.\, C_2} &= \Sigma x{:}\overline{C_1}.\, \overline{C_2} \\
\overline{\Pi x{:}F.\, C} &= \Pi x{:}F.\, \overline{C}
\end{aligned}$$

This operation is extended pointwise to typing environments.

Proposition 1 *Let E be a typing environment containing F and C types.*

1. *If $\overline{E} \vdash_s m : A$, then there exists C such that $\overline{C} = A$ and $E \vdash_m m : C$.*

2. *If* $\overline{E} \vdash_s A \approx A'$ *and* $A = \overline{C}$, *then there exists* C' *such that* $A' = \overline{C'}$ *and* $E \vdash_m C \approx C'$.

Part (2) is the main difficulty of the proof: apparently, system S can derive more type equalities (using reductions during typing whenever necessary) than system M (where the only information available is the one recorded in manifest types). We will show that this is not the case, at least for the first-order fragment.

The proof makes use of the following properties of system M: subject reduction (if $E \vdash_m m : M$ and $m \xrightarrow{*} m'$, then $E \vdash_m m'$) and uniqueness of typings (if $E \vdash_m m : M_1$ and $E, E' \vdash_m m : M_2$, then $E, E' \vdash_m M_1 \approx M_2$).

Proof: by induction on the derivations and case analysis on the last rule used. We show the main cases.

(2), rule 6. From the left premise and the induction hypothesis (1), we get $E \vdash_m m : \exists t = \sigma . C$ with $\exists t. \overline{C} = A$. Since $m \xrightarrow{*} [t = \tau, m']$, it follows that $E \vdash_m [t = \tau, m'] : \exists t = \sigma . C$ by subject reduction. By uniqueness of typings, $E \vdash_m \sigma \approx \tau$. Hence $E \vdash_m \mathtt{fst}(m) \approx \tau$ by rule 7 and transitivity, and we can take $C' = \tau$.

(1), rule 1. If $E(x)$ is a C type, we can take $C = E(x)$ Otherwise, $E(x)$ is a F type and we can take $C = E(x)/x$.

(1), rule 3. We have $A = \exists t. A'$. By induction hypothesis (1), we have $E \vdash_m m\{t \leftarrow \tau\} : C'\{t \leftarrow \tau\}$ and $\overline{C'} = A'$. We can take $C = (\exists t = \tau . C')$.

(1), rule 2. Follows immediately from the induction hypothesis (1) and the fact that $E \vdash_m C <: \overline{C}$ for all C.

(1), rule 4. By induction hypothesis (1) applied to the premises, we get $E \vdash_m m_1 : C_1$ and $E \vdash_m m_2\{x \leftarrow m_1\} : C_2\{x \leftarrow m_1\}$ (3), with $\overline{C_1} = A_1$ and $\overline{C_2} = A_2$. We must prove that $E, x : C_1 \vdash_m m_2 : C_2$ (4). Consider each occurrence of x substituted by m_1 in the derivation of (3). These occurrences correspond to sub-derivations of the format $E, E' \vdash_m m_1 : C$ (5) for some E' and C. By uniqueness of typings, we have $E, E' \vdash_m C \approx C_1$. Hence we can derive $E, x : C_1, E' \vdash_m x : C$ (6). By substituting (6) for (5) in the derivation of (3) for each occurrence of x, we obtain a derivation of (4).

(1), rule 5. Follows immediately from the induction hypothesis (2). □

4.3 Higher-order functors

The expressiveness result above does not extend to higher-order functors. The reason is that higher-order functors are "more polymorphic" in system S than in system M. Consider the following example (we revert to SML-like syntax for clarity):

```
signature S = sig type t ... end;
module G =
  functor (F: functor(X:S)S) functor (X:S) F(X);
module A =
  G(functor(X:S) X)
   (struct type t=int ... end);
module B =
  G(functor(X:S) struct type t=int ... end)
   (struct type t=bool ... end);
```

In SML with higher-order functors [25], `A` and `B` are well-typed, and moreover `A.t` and `B.t` are both compatible with `int`. In our calculus, if we do not introduce any manifest type in the definition of `G`, `A` and `B` are well-typed but `A.t` and `B.t` are incompatible with `int`. To make `A.t` compatible with `int`, we can define `G` as:

```
module G =
  functor (F: functor(X:S) S with type t=X.t)
  functor (X:S) F(X);
```

but then the definition of `B` is ill-typed, because the constant functor given as first argument has the wrong type. To make `B.t` compatible with `int`, we can similarly define `G` as.

```
module G =
  functor (F: functor(X:S) S with type t=int)
  functor (X:S) F(X);
```

but then the definition of `A` is ill-typed. Hence, our system makes it impossible to define a functor `G` that can be used in all contexts where its SML counterpart can.

From the discussion above, one might conclude that higher-order functors are strictly more expressive in SML than in our system. This is not so: in our system, higher-order functors can specify all the sharing properties of their functorial arguments, such as "the parameter `F` is a functor that takes a structure `X` with a type `t` and returns a structure with a type equal to `t list`":

```
functor(F: functor(X:S) S with type t = X.t list)
```

This is not supported in Tofte's proposal [25]: sharing constraints in the functor specification can require that the `t` component of the result structure is identical to another type constructor such as `X.t`, but they cannot express more general dependencies as in the example above. Consequently, Tofte's system does not allow abstracting over any functor: the following code fragment

```
signature S = sig type t; val x:t; val f:t->t end;
functor F(X:S) = struct type t=X.t list; ... end;
structure G =
  struct
    structure A = struct type t=int; ... end;
    structure R = F(A);
    val y = R.f [A.x]
  end;
```

is well-typed, but we cannot abstract over `F` in `G`, because no functor signature for `F` can specify that `F(X).t` is equal to `X.t list`, nor to `int list`. In contrast, our system allows abstracting over any functor—which is an important motivation for higher-order functors: allow arbitrary program fragments to be "closed" by abstracting over all free identifiers.

5 Extensions

5.1 Type systems with kinds

A useful extension of the module calculus presented in this paper is the introduction of kinds at the level of core types. Kinds are required to ensure the well-formedness of type expressions in the presence of type constructors, as in full ML, or type operators, as in F_ω. Kinds also provide an elegant treatment of bounded quantification [3, 4].

Introducing kinds in the module calculus is straightforward: abstract type specifications in signatures now have the form **type** $t :: \kappa$, where κ is a kind. The modified typing rules check that the implementation types are of the expected kinds, and that kinds are properly included when checking signature inclusion. (See [4] for a similar calculus.)

Having kinds in type specifications offers an opportunity to simplify the treatment of manifest types. The idea is to introduce the kind $\texttt{EQUIV}(\tau)$ of all types that are equivalent to τ. This way, a manifest type declaration **type** $t = \tau$ in a signature can be expressed as the type specification **type** $t :: \texttt{EQUIV}(\tau)$. This trick parallels Cardelli's treatment of bounded quantification using the kind $\texttt{POWER}(\tau)$ of all subtypes of τ [3, 4]. With the EQUIV kind, there is only one syntactic construct to declare a type in a signature: **type** $t :: \kappa$. The kind κ says whether **t** is manifest (if $\kappa = \texttt{EQUIV}(\tau)$) or abstract (if $\kappa = \texttt{TYPE}$, where TYPE is the kind of all types).

Besides simplifying the syntax, the EQUIV kind also clarifies the typing rules, by moving all manifest type-specific rules up to the kind level, making them orthogonal to the rules for structures and signatures. The properties of manifest types are captured by the rules below for the EQUIV kind (left column). They are surprisingly similar to the rules for the POWER kind (right column).

$$\frac{E \vdash \tau \approx \sigma}{E \vdash \tau :: \texttt{EQUIV}(\sigma)} \qquad \frac{E \vdash \tau <: \sigma}{E \vdash \tau :: \texttt{POWER}(\sigma)}$$

$$\frac{E \vdash \tau :: \texttt{EQUIV}(\sigma)}{E \vdash \tau \approx \sigma} \qquad \frac{E \vdash \tau :: \texttt{POWER}(\sigma)}{E \vdash \tau <: \sigma}$$

$$E \vdash \texttt{EQUIV}(\tau) <:: \texttt{TYPE} \qquad E \vdash \texttt{POWER}(\tau) <:: \texttt{TYPE}$$

$$\frac{E \vdash \tau \approx \sigma}{E \vdash \texttt{EQUIV}(\tau) <:: \texttt{EQUIV}(\sigma)} \qquad \frac{E \vdash \tau <: \sigma}{E \vdash \texttt{POWER}(\tau) <:: \texttt{POWER}(\sigma)}$$

As demonstrated by the rules above, once kinds are introduced in a type system, it is then straightforward to extend it with manifest types, bounded quantification, or both at the same time.

5.2 First-class modules

Another natural extension of the work presented here is to merge the module language and the base language, by treating modules as first-class values. This approach brings additional expressive power and simplifies the formalization.

In the case of a module system based on strong sums, such as SML's, first-class modules raise major difficulties [12]: simply-typed lambda-calculus with strong sums is logically inconsistent (i.e. non-normalizing) and has no phase distinction (i.e. arbitrary reductions are required during typechecking); as a consequence, typechecking is undecidable. Stratification into a base language and a module language is essential to ensure the phase distinction and decidability of typing [13].

First-class modules cause less difficulties in our approach because it is based on weak sums [19, 7]. No reductions are needed during typechecking, hence the phase distinction is obvious, whether modules are first-class or not.

As shown by Harper and Lillibridge [10], first-class modules still make typechecking undecidable in the presence of manifest types, but for different reasons than for strong sums: what is undecidable is the subtyping relation, as in $F_{<:}$ [20], and this is due to the combination of subtyping, manifest types and dependent function types at the same level. Stratification avoids this problem by allowing different combinations of these features at the two levels: in the system presented above, manifest types but no subtyping nor dependent function types at the base level, and subtyping and dependent function types but no manifest types at the module level.

6 Conclusions

We have presented a variation on the SML approach to modularity that propagates type equations explicitly through signatures and module types, instead of implicitly through structures and module values. While retaining the expressiveness of the SML module system, our variant provides much better support for Modula-style separate compilation and, more generally, makes it easier to understand code fragments with free structure identifiers. The underlying type theory is also simpler.

A prototype batch compiler integrating the main ideas in this paper (type abbreviations in signatures and opaque interpretation of signatures) has been derived from SML/NJ 0.93 by Pierre Crégut [8]. The main difference with the work presented here is that both opaque and transparent signatures are supported, via two distinct module binding constructs. Another implementation, based on the author's Caml Light system and closer to the calculus introduced in sections 2 and 3, is in progress.

On the practical side, some concern has been expressed about the additional verbosity brought by declaring manifest types in signatures. Previous attempts at programming in a fully functorized style, with all type equations explicit, have demonstrated a major increase in program size due to the extra sharing declarations required [2]. We expect this problem to be less acute in our approach, since manifest types provides better support for non-fully functorized code; moreover, one manifest type declaration (in the definition of the signature of a module) sometimes replace several sharing declarations (one for each functor that imports this module). More practical experience is required to assess this issue.

Finally, the general idea of making the definitions of some structure components explicit in the signature, here applied to the type components, could also be extended to other kinds of components: values and sub-structures. Defining a value in a module signature does not make much sense at first sight, but is actually a common programming practice (definition of constants in Modula interfaces, of macros and in-line functions in C and C++) and provides a simple yet highly practical approach to user-controlled function inlining.

Acknowledgments

The author is grateful for conversations with Luca Cardelli, Robert Harper, Mark Lillibridge, Pierre Crégut, Didier Rémy and María-Virginia Aponte. John Mitchell, Stefan Kahrs and Lars Birkedal suggested a number of improvements in earlier drafts.

References

[1] M.-V. Aponte. Extending record typing to type parametric modules with sharing. In *20th symposium Principles of Programming Languages*, pages 465–478. ACM Press, 1993.

[2] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML kit, version 1. Technical report 93/14, DIKU, 1993.

[3] L. Cardelli. Structural subtyping and the notion of power type. In *15th symposium Principles of Programming Languages*, pages 70–79. ACM Press, 1988.

[4] L. Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, 1989.

[5] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. IFIP TC2 working conference on programming concepts and methods*. North-Holland, 1990.

[6] L. Cardelli and J. C. Mitchell. Operations on records. In *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 22–52, 1989.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing surveys*, 17(4):471–522, 1985.

[8] P. Crégut. Separate compilation in SML. Working note, Magic group, ECRC, 1993.

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[10] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*. ACM Press, 1994.

[11] R. Harper, R. Milner, and M. Tofte. A type discipline for program modules. In *TAPSOFT 87*, volume 250 of *Lecture Notes in Computer Science*, pages 308–319. Springer-Verlag, 1987.

[12] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. Prog. Lang. Syst.*, 15(2):211–252, 1993.

[13] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *17th symposium Principles of Programming Languages*, pages 341–354. ACM Press, 1990.

[14] D. MacQueen. Modules for Standard ML. In R. Harper, D. MacQueen, and R. Milner, editors, *Standard ML*. University of Edinburgh, technical report ECS LFCS 86-2, 1986.

[15] D. MacQueen. Using dependent types to express modular structure. In *13th symposium Principles of Programming Languages*, pages 277–286. ACM Press, 1986.

[16] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

[17] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, 1990.

[18] J. C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial intelligence and mathematical theory of computation*, pages 305–330. Academic Press, 1991.

[19] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Prog. Lang. Syst.*, 10(3):470–502, 1988.

[20] B. C. Pierce. Bounded quantification is undecidable. In *19th symposium Principles of Programming Languages*, pages 305–315. ACM Press, 1992.

[21] D. Rémy. Extending ML type system with a sorted equational theory. Research report 1766, INRIA, 1992.

[22] J. C. Reynolds. The essence of Algol. In de Bakker and van Vliet, editors, *Algorithmic languages*, pages 345–372. North-Holland, 1981.

[23] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Lisp and Functional Programming 1992*, pages 288–298, 1992.

[24] Z. Shao and A. Appel. Smartest recompilation. In *20th symposium Principles of Programming Languages*, pages 439–450. ACM Press, 1993.

[25] M. Tofte. Principal signatures for higher-order program modules. In *19th symposium Principles of Programming Languages*, pages 189–199. ACM Press, 1992.

[26] M. Tofte. Type abbreviations in signatures. Message sent to the `sml` mailing list, Jan. 1992.

[27] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.