Applicative functors and fully transparent higher-order modules

Xavier Leroy
INRIA
B.P. 105, Rocquencourt, 78153 Le Chesnay, France.
Xavier.Leroy@inria.fr

Abstract

We present a variant of the Standard ML module system where parameterized abstract types (i.e. functors returning generative types) map provably equal arguments to compatible abstract types, instead of generating distinct types at each application as in Standard ML. This extension solves the full transparency problem (how to give syntactic signatures for higher-order functors that express exactly their propagation of type equations), and also provides better support for non-closed code fragments.

1 Introduction

Most modern programming languages provide support for type abstraction: the important programming technique where a named type `t` is equipped with operations `f`, `g`, ... then the concrete implementation of `t` is hidden, leaving an abstract type `t` that can only be accessed through the operations `f`, `g`, ... Type abstraction provides fundamental typing support for modularity, since it enables a type-checker to catch violations of the modular structure of programs.

Type abstraction is usually implemented through generative data type declarations: to make a type `t` abstract, the type-checker generates a new type `t` incompatible with any other type, including types with the same structure. From this, it is tempting to explain type abstraction in terms of generativity of type declarations and say for instance that "a type is abstract because it is created each time its definition is evaluated". The *Definition of Standard ML* [14, 8] formalizes this approach as a calculus over type stamps that defines when "new" types are generated and when "old" types are propagated. This approach is adequate for specifying a type-checker, but too low-level and operational in nature to help understanding type abstraction and reason about programs using it.

Independently, Mitchell and Plotkin [16] have proposed a more abstract, less operational account of type abstraction based on a parallel with existential quantification in logic. Instead of operational intuitions about type generativity, this approach uses a precise semantic characterization: representation independence [17, 15], to show that type abstraction is enforced. This abstract approach has since been extended to account for the main features of the Standard ML module system: the "dot notation" as elimination construct for abstract types [3, 4] and the notion of type sharing and its propagation through functors [7, 10].

Unfortunately, some features described by operational frameworks remain unaccounted for in the abstract approach, such as structure sharing and the "fully transparent" behavior of higher-order functors predicted by the operational approach [13]. Also, even though the abstract approach is syntactic in nature and therefore highly compatible with separate compilation [10], code fragments with free functor identifiers could be supported better (see section 2.4 for an example). MacQueen [13, 1] claims that the problem with higher-order functors is serious enough to invalidate the abstract approach and justify the recourse to complicated stamp-based descriptions of higher-order functors and separate compilation mechanisms.

The work presented in this paper is an attempt to solve two of these problems (fully transparent higher-order functors and support for non-closed code fragments) in a syntactic framework derived from [10]. It relies on a modification of the behavior of functors (parameterized modules). In Standard ML and other models based on type generativity, a functor defining an abstract type returns a different type each time it is applied. We say that functors are *generative*. In this work, we consider functors as *applicative*: if the functor is applied twice to provably equal arguments, the two abstract types returned remain compatible. Functors therefore map equals to equals, which enables equational reasoning on functor applications during type-checking. In turn, this allows more precise signatures for higher-order functors, thereby solving the full transparency problem.

Applicative functors are also interesting as an example of a module system that ensures type abstraction (the representation independence properties still hold) without respecting strict type generativity (some applications of a given functor may return new types while others return compatible types). In this approach, type abstraction mechanisms are considered from a semantic point of view (how to make programs robust with respect to changes of implementations?) rather than from an operational point of view (when are two structurally identical types compatible?). This work illustrates the additional expressiveness and flexibility allowed by this shift of perspective.

The remainder of this paper is organized as follows. Section 2 introduces informally the applicative semantics of functors and the main technical devices that implement it. Section 3 formalizes a calculus with applicative functors. Section 4 shows that the representation independence property still holds, and section 5 that higher-order functors are fully transparent in this calculus. Section 6 discusses related work and section 7 gives concluding remarks.

To appear in the proceedings of the 22nd symposium Principles of Programming Languages, january 1995.

2 Functor applications in type paths

2.1 Type paths and the propagation of type equalities

Module systems such as SML's allow type expressions of the form $x.t$, where x is a structure identifier (the name of a module) and t is the name of a type defined in this structure. Since structures can be nested, type expressions such as $x_1 \ldots x_n.t$ are also allowed, denoting the t type component of the x_n substructure of ... the structure x_1. These type expressions are called *type paths* or *long type identifiers*.

When type bindings are transparent (named types are compared by structure), type paths play no role in type-checking, since they can always be replaced by the type expressions to which they are bound. This is not the case with opaque type bindings, where the definitions of type identifiers are hidden and types are compared by name. Opaque bindings are crucial to implement type abstraction and representation hiding. In SML, they are provided by the `abstype` and `datatype` constructs. Type paths play the role of witnesses of opaque types [5]: although the definition of an opaque type is not available, two occurrences of a type path denoting this opaque type are recognized as compatible types. In other terms, syntactic equality between type paths implements the name equivalence that characterizes type abstraction and type generativity [4].

Combined with type definitions in signatures (the ability to specify type equalities in module interfaces), type paths can also express how abstract types are propagated and shared across substructures of a structure, and across the argument and result of a functor. Consider for instance the following `Set` functor that implements sets over any type equipped with a total ordering:

```
signature ORD =
  sig type t; val less: t -> t -> bool end
functor Set(Elt: ORD): SET =
  struct
    type elt = Elt.t
    datatype set = Leaf | Node of set * elt * set
    val empty = Leaf
    ... (* Familiar ordered binary tree
           implementation omitted *)
  end
```

The following signature for the `Set` functor captures the fact that the `elt` component of the result is the same type as the `t` component of the argument:

```
functor Set(Elt: ORD): SET =
  sig
    type elt = Elt.t
    type set
    val empty: set
    val member: elt -> set -> bool
    ...
  end
```

This combination of type paths and dependent functor types has been used to give complete syntactic accounts of type sharing in the SML module system [10, 7, 11].

2.2 Restriction of projections to paths

An obvious generalization of type paths is to allow projections of type components from arbitrary structure expressions: then, $m.t$ would be a valid type expression for any structure expression m whose result contains a t component [12, 9]. This extension adds considerable expressive power, but raises delicate issues. First, when are two type expressions $m.t$ and $m'.t$ compatible? Clearly, we cannot check that m and m' reduce to the same structure, since equality of structure expressions is undecidable. Second, even if we compare structures by mere syntactic equality ($m.t$ and $m'.t$ are compatible if and only if the structure expressions m and m' are syntactically identical), some type abstraction is lost. For instance, two occurrences of the type expression

```
(struct abstype t = τ with decls end).t
```

would be recognized as compatible, while the two abstract type definitions can come from different parts of the program and should create two distinct abstract types `t`. This problem is particularly apparent in the syntactic module systems [10, 7], which have a typing rule (the "self" rule) that transforms abstract types into types "manifestly equal to themselves": if the structure path `p` has signature

```
p : sig ... type t; ... end
```

then it also has signature

```
p : sig ... type t = p.t; ... end
```

If all structure expressions are allowed in paths, the "self" rule makes abstract types that happen to have the same implementation automatically compatible:

```
structure A = struct abstype t = τ with decls end
structure B = struct abstype t = τ with decls end
```

By application of the "self" rule, we obtain the following signatures for `A` and `B`:

```
A: sig type t =
         (struct abstype t = τ with decls end).t;
       ... end
B: sig type t =
         (struct abstype t = τ with decls end).t;
       ... end
```

Hence `A.t` = `B.t`, which violates type abstraction.

To avoid this problem, all module-language constructs whose evaluation can generate new types (by evaluating an `abstype` or `datatype` definition) must not occur in type projections. This excludes structure construction (`struct...end`) and functor application as well (the body of the functor can generate new types at each application). The only constructions that remains are access to a structure identifier (x) and access to a substructure ($x_1.x_2$), that is, the type paths ($p ::= x \mid p.x$), as in SML. The restriction of type projections to paths is therefore equivalent to the strict notion of type generativity found in SML.

2.3 Functor applications in paths

In spite of these considerations, there are situations where it would be extremely useful to extend slightly the class of paths (the syntactic class p of structure expressions such that $p.t$ is a legal type expression) to include simple cases of functor applications, where the functor and its argument are themselves paths. Let us therefore take

$$p ::= x \mid p.x \mid p_1(p_2)$$

and illustrate the consequences of this choice, on the expressiveness of the language and on the notion of type abstraction.

2.4 Local applications of functors

A first situation where this extension proves useful is to support functors that apply other functors locally. Consider the following `Dict` functor implementing dictionaries:

```
signature DICT =
  sig
    type key
    type 'a dict
    val empty: 'a dict
    val add: key -> 'a -> 'a dict -> 'a dict
    val find: key -> 'a dict -> 'a
  end
functor Dict(Key: ORD): DICT =
  struct
    type key = Key.t
    datatype 'a dict =
      Leaf
    | Node of 'a dict * key * 'a * 'a dict
    ... (* Binary tree implementation omitted *)
  end
```

Assume we need to extend this functor with a `domain` operation that returns the set of keys of a dictionary. To do so, we need a structure implementing sets of keys. The simplest approach is to construct this structure inside the `Dict` functor, using the `Set` functor:

```
signature DICT =
  sig
    ...
    structure KeySet: SET sharing KeySet.elt = key
    val domain: 'a dict -> KeySet.Set
  end
functor Dict(Key: ORD): DICT =
  struct
    ...
    structure KeySet = Set(Ord)
    fun domain dict = ...
  end
```

Unfortunately, the signature above does not reflect that `KeySet` has been obtained by applying `Set` to `Ord`; therefore, the type `KeySet.set` in the result structure of `Dict` is assumed incompatible with other set types obtained by applying `Set` elsewhere to the same ordered type. Continuing the example, assume we have another functor, say an implementation of priority queues, that uses the same trick as `Dict`:

```
signature PRIOQUEUE =
  sig
    type elt
    type queue
    val empty: queue
    val add: elt -> queue -> queue
    val extract: queue -> elt * queue
    structure EltSet: SET sharing EltSet.elt = elt
    val contents: queue -> EltSet.set
  end
functor PrioQueue(Elt: ORD): PRIOQUEUE =
  struct
    ...
    structure EltSet = Set(Elt)
    ...
  end
```

Then, `Dict` and `PrioQueue` cannot be used together, because the `set` types used by the `contents` and `domain` functions are not compatible:

```
structure IntOrder =
  struct type t = int; fun less x y = (x<y) end
structure IntDict = Dict(IntOrder)
structure IntPrioQueue = PrioQueue(IntOrder)
```

The types `IntDict.KeySet.set` and `IntPrioQueue.EltSet.set` are incompatible, therefore the following expression does not type-check:

```
IntDict.domain d = IntPrioQueue.contents q
```

The SML solution to the problem above is to avoid applying locally the `Set` functor and parameterize instead `Dict` and `PrioQueue` by the required `Set` structure:

```
functor Dict(
      structure Key: ORD
      structure Set: SET
      sharing Set.elt = Key.t): DICT =
  struct
    ... structure KeySet = Set ...
  end
functor PrioQueue(
      structure Elt: ORD
      structure Set: SET
      sharing Set.elt = Elt.t): PRIOQUEUE =
  struct
    ... structure EltSet = Set ...
  end
structure IntSet = Set(IntOrder)
structure IntDict =
  Dict(structure Elt = IntOrder
       structure Set = IntSet)
structure IntPrioQueue =
  PrioQueue(structure Elt = IntOrder
            structure Set = IntSet)
```

By hoisting the application `Set(IntOrder)` outside of `Dict` and `PrioQueue`, we have made explicit that the `KeySet` and `EltSet` substructures of `IntDict` and `IntPrioQueue` provide compatible `set` types. Therefore, the following expression now type-checks:

```
IntDict.domain d = IntPrioQueue.contents q
```

However, what appeared to be an incremental change of the program (add some operations to existing functors) has required major changes to the modular structure of the program:

- All other uses of `Dict` and `PrioQueue` in the program must be modified to provide the extra `Set` argument, even if they do not use the new operations.
- Higher-order functors that take `Dict` or `PrioQueue` as arguments must also be modified.
- Hoisting the application `Set(IntOrder)` from the points where it is actually used to a common ancestor of these points in the dependency graph is a non-local program transformation, as in MacQueen's "diamond import" example [12].

Introducing functor applications in paths enables a much more elegant solution: the functors `Dict` and `PrioQueue` can apply `Set` locally, as in the original attempt, and receive the following signatures:

```
functor Dict(Key: ORD):
  sig
    type key = Key.t
    type 'a dict
    ...
    val domain: 'a dict -> Set(Key).set
  end
functor PrioQueue(Elt: ORD):
  sig
    type elt = Elt.t
    type queue
    ...
    val contents: queue -> Set(Elt).set
  end
```

Since the signatures show explicitly how the `set` types are derived from the functor arguments, the structures obtained by applying `Dict` and `PrioQueue` to the same ordered type now interact correctly:

```
structure IntDict = Dict(IntOrder)
  (* : sig
         ...
         val domain: 'a dict -> Set(IntOrder).set
       end *)
structure IntPrioQueue = PrioQueue(IntOrder)
  (* : sig
         ...
         val contents: queue -> Set(IntOrder).set
       end *)
```

From the signatures above, it follows that the two types `IntDict.KeySet.set` and `IntPrioQueue.EltSet.set` are equal to `Set(IntOrder).set`, and therefore compatible.

2.5 Full transparency in higher-order functors

As previously mentioned, the combination of dependent functor types and type equalities in signatures makes it possible to give syntactic signatures to first-order functors that characterize exactly the "input-output behavior" of functors: how they propagate type components from their argument structure to their result structure. (See for example the signature for `Set` in section 2.1.) This property, in turn, enables simple syntactic descriptions of the module system and simple separate compilation mechanisms [10], with no loss in expressiveness with respect to SML [11].

Unfortunately, this result does not extend straightforwardly to higher-order functors: some higher-order functors do not possess any syntactic signature that characterizes exactly their behavior. Consider for instance the paradigmatic higher-order functor:

```
signature S = sig type t end
functor Apply(functor F(X:S):S; structure A:S) =
  F(A)
```

The expected behavior for this kind of functors, called the "fully transparent" behavior in [13] and predicted by the models based on strong sums [12, 9], is that at application-time all type equalities known about its two arguments are combined to deduce the type equality that holds on the `t` component of the result. For instance, if the `F` argument is the identity and the `A` argument has `t` = `int`, then the result also has `t` = `int`; if `F` is the constant functor returning `t` = `bool`, then the result has `t` = `bool`.

```
functor Identity(X:S) = X
structure Int = struct type t = int end
structure B = Apply(Identity Int)
  (* We get B.t = int *)
functor Constant(X:S) = struct type t = bool end
structure C = Apply(Constant Int)
  (* We get C.t = bool *)
```

With the standard notion of type paths, it is impossible to capture this behavior in a syntactic signature for `Apply`. The most general signature for `Apply`,

```
functor Apply(functor F(X:S):S; structure A:S):
  sig type t end
```

does not propagate any type equalities on the `t` component of the result. `Apply` can be modified to propagate some equalities in special cases, at the cost of making it unapplicable in other cases. For instance, the correct propagation of equalities in the example `Apply(Identity Int)` can be obtained by defining `Apply` as

```
functor Apply(
    functor F(X: S): sig type t=X.t end
    structure A: S)
  = F(A)
```

The signature for `Apply` is then

```
functor Apply(
    functor F(X: S): sig type t=X.t end
    structure A: S)
  : sig type t=A.t end
```

From this signature, it follows that `B = Apply(Identity Int)` has `B.t` manifestly equal to `int`. But then the application `Apply(Constant Int)` is ill-typed, since the functor `Constant` does not meet the specification of the `F` argument of `Apply`. The modified definition of `Apply` propagates more type equations, but makes the higher-order functor less general. At any rate, full transparency is not achieved.

The introduction of functor applications in type paths provides a simple, elegant solution to this full transparency problem, allowing functors such as `Apply` to receive syntactic signatures that capture exactly their type propagation behavior. In the case of `Apply`, this signature is

```
functor Apply(functor F(X:S):S; structure A:S):
  sig type t = F(A).t end
```

This is a correct signature for `Apply`: since the functor body `F(A)` belongs to the extended class of paths, it has not only signature `S`, but also `sig type t = F(A).t end` by application of the "self" typing rule.

Moreover, this signature propagates type equalities correctly, because at application-time `F` and `A` in the result signature are substituted by the actual arguments f and a to the functor, and all known equalities about the result of $f(a)$ will also hold for the result of `Apply`. For instance,

```
structure B = Apply(Identity Int)
    (* : sig type t = Identity(Int).t end *)
```

and `Identity(Int)` has signature `sig type t = Int.t end`, from which it follows `B.t` = `Int.t` = `int`. Similarly,

```
structure C = Apply(Constant Int)
    (* : sig type t = Constant(Int).t end *)
```

and `Constant(Int).t` = `bool`, hence `C.t` = `bool`, as expected.

As in the first-order case, the propagation of type equalities through syntactic functor signatures is limited by the fact that only paths are allowed in type expressions, not type projections from arbitrary structures. Consider the following variant of `Apply`:

```
functor ApplyProd(functor F(X:S):S;
                  structure A:S) =
  F(struct type t = A.t * A.t end)
```

Its natural signature is:

```
functor ApplyProd(functor F(X:S):S;
                  structure A:S):
  sig
    type t = F(struct type t = A.t * A.t end).t
  end
```

Unfortunately, this is not a well-formed signature, since the argument to `F` is not a path. We must therefore revert to the less precise signature

```
functor ApplyProd(functor F(X:S):S;
                  structure A:S):
  sig type t end
```

which does not propagate type equalities as expected. To achieve full transparency, the program must be rewritten so that the argument to `F` is a path. Since this argument contains `A` as a free variable, we must actually "lambda-lift" it as follows:

```
functor G(A:S) =
  struct type t = A.t * A.t end
functor ApplyProd(functor F(X:S):S;
                  structure A:S) = F(G(A))
```

Then, `ApplyProd` is assigned the fully transparent signature

```
functor ApplyProd(functor F(X:S):S;
                  structure A:S):
  sig type t = F(G(A)).t end
```

which ensures the proper propagation of type equations. This trick is an instance of a general normalization technique that transforms an arbitrary program to eliminate applications of functors to non-paths, therefore ensuring that type equations are always propagated as expected (see section 5.2).

2.6 Applicative semantics of functor application

As recalled in section 2.2, the standard notion of paths ($p ::= x \mid p.x$) is the only one that guarantees strict type generativity. In turn, type generativity guarantees type abstraction. The question, then, is: how much type generativity and type abstraction is lost if we allow functor applications in paths? Some generativity, but no abstraction. More precisely, the only difference is that if we have a functor that returns an abstract type, and we apply it twice to syntactically identical paths, then we obtain two compatible abstract types, while with the standard notion of paths we would have generated two distinct types. Consider:

```
structure IntSet1 = Set(IntOrder)
structure IntSet2 = Set(IntOrder)
```

With the standard notion of paths, we have `IntSet1.set` $\neq$ `IntSet2.set`. If functor applications are allowed in paths, by applying the "self" rule, we obtain

```
IntSet1:
  sig ... type set = Set(IntOrder).set ... end
IntSet2:
  sig ... type set = Set(IntOrder).set ... end
```

from which it follows that `IntSet1.set` = `IntSet2.set`.

In other terms, the consequence of adding functor applications in paths is that functors returning abstract types now map equal structure path arguments to equal abstract types. We call this behavior *applicative*, by opposition to the usual *generative* behavior, where each application of such functors generates a new abstract type, whether the arguments are identical or not.

The applicative behavior appears only if the functor arguments are syntactically identical structure paths: in all other cases, the "self" rule does not apply and the abstract types in the functor results are considered different. For instance, if we define

```
structure IntSet3 = Set(IntOrder: ORD)
structure IntSet4 = Set(IntOrder: ORD)
```

we obtain `IntSet3.set` $\neq$ `IntSet4.set`, since `(IntOrder: ORD)` is not a path. This may look unnatural, and more refined syntactic criteria could be used to determine equality of functor arguments; on the other hand, the line has to be drawn somewhere, and equality of structure paths is easily explained and understood.

We claim that the applicative semantics for functor applications does not violate type abstraction and does not weaken the robustness of programs. First, even though the applicative semantics makes some previously incompatible abstract types compatible (`IntSet1.set` and `IntSet2.set` in the example above), the representations of these abstract types are still hidden, and outsiders still cannot forge or inspect directly values of these types. Section 4 formalizes this argument as a representation independence property.

Moreover, several module-level construct still generate new types predictably, because they still do not belong to the extended class of paths: structure construction `struct...end` and restriction of a structure by a signature (*strexp* : *sigexp*). The programmer can rely on these constructs to obtain new, incompatible types, if desired for added safety. Continuing the example above, the types `IntSet1.set` and `IntSet2.set` can be made different by adding a signature constraint:

```
structure IntSet1 = (Set(IntOrder): SET)
structure IntSet2 = (Set(IntOrder): SET)
```

The opaque interpretation of constraints in our module calculus guarantees that all equalities known about the `set` component are forgotten. Moreover, `(Set(IntOrder): SET)` is syntactically not a path, hence the "self" rule cannot be used to derive a type equality between `IntSet1.set` and `IntSet2.set`. In other terms, functor application can no longer be used to force the generation of new types (as is sometimes done in SML using functors with no arguments), but other constructs such as signature constraints can be used for the same purposes.

2.7 Applicative functors and side-effects

In an imperative language such as ML, one may wonder whether the hypothesis that functors map equals to equals is sound. In a language with side-effects and modules as first-class values [3, 7], the applicative semantics for functors is actually unsound:

```
val r = ref false
functor F() : sig type t end =
  (r := not !r;
   if !r then struct type t = int end
         else struct type t = bool end)
structure A = F()
structure B = F()
```

Semantically, `A.t` is `int` and `B.t` is `bool`, but the applicative semantics for functors assumes `A.t` = `B.t`.

The problem is avoided in a stratified language such as SML: type components of structures cannot depend on values, only on types, and the language of types is purely functional; hence, the application of a functor to two structures with identical type components returns two structures with identical type components. The applicative semantics is therefore sound.

It is true, however, that value components of a functor body may depend on the store. It would therefore be incorrect to generalize the applicativity hypothesis to "a functor maps two structures that share (in SML's structure sharing sense) to two structures that share". In this paper, we only consider sharing between type components of structures. (See [6] for a treatment of structure sharing in functor signatures.)

3 A calculus with applicative functors

In preparation for representation independence and expressiveness results, we now define a module calculus with applicative functors, derived from [10, 11].

3.1 Syntax

In the following grammar, v, t, and x are names (for value, type, and module components of structures, respectively), and v_i, t_i, and x_i are identifiers (for values, types, and modules). All identifiers (e.g. x_i) have a name part (here, x) and a stamp part (i) that distinguishes identifiers with the same name. Bound identifiers can be renamed, but α-conversion must preserve the name parts of identifiers and can only change the stamp part. This way, access by name inside structures is meaningful, yet α-conversion can still be performed to avoid name clashes.

Access paths:

$p ::= x_i$	module identifier
$\mid p.x$	access to a module component
$\mid p_1(p_2)$	functor application

Value expressions:

$e ::= v_i$	value identifier
$\mid p.v$	value field of a structure
$\mid \ldots$	base language-dependent

Type expressions:

$\tau ::= t_i$	type identifier
$\mid p.t$	type field of a structure
$\mid$ `int` $\mid \tau \rightarrow \tau \mid \ldots$	base language-dependent

Module expressions:

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

Structure body:

$s ::= \varepsilon \mid d; s$

Structure components:

$d ::=$ `val` $v_i = e$	value definition
$\mid$ `type` $t = \tau$	type definition
$\mid$ `module` $x_i = m$	module definition

Module types:

$M ::=$ `sig` S `end`	signature type
$\mid$ `functor`$(x_i : M_1) M_2$	functor type

Signature body:

$S ::= \varepsilon \mid D; S$

Signature components:

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

Programs:

$P ::=$ `prog` s `end`

We assume given a base language (value expressions e, type expressions τ) that is left mostly unspecified. It can refer to value and type components bound earlier in the same structure through identifiers (v_i and t_i), and to value and type components of other structures through paths ($p.v$ and $p.t$).

At the level of the module language (m), we have structures, functor abstractions and functor applications. Structures are collection of bindings for values, types and modules (either substructures or functors). The corresponding module types M are signatures (collections of declarations for values, types and modules) and functor types (dependent function types). Type components in signatures can be declared either abstractly (`type` t_i) or transparently (`type` $t_i = \tau$).

Signatures are treated as opaque for signature matching, meaning that a transparent type binding (`type` $t_i = \tau$) restricted by an abstract type specification (`type` t_i) becomes abstract: the type equality $t_i = \tau$ is forgotten. SML's generative bindings `abstype` and `datatype` can therefore be expressed as a transparent type binding followed by a restriction by an abstract signature.

Complete programs P are sequences of definitions s that define an integer-valued field named `res`, which is the observable result of the program execution.

3.2 Typing rules

The typing rules for this calculus are shown in figure 1. The rules define the following judgements:

$E \vdash m : M$	The module m has module type M.
$E \vdash M_1 <: M_2$	The module type M_1 is a subtype of the module type M_2.
$E \vdash M$ $\mathsf{modtype}$	The module type M is well-formed.
$E \vdash P$ ok	The program P is well-typed.

We write $BV(S)$ for the set of identifiers bound by the signature S, and similarly for typing environments. We assume

Typing of module expressions ($E \vdash m : M$), structures ($E \vdash s : S$) and programs ($\vdash P$ ok):

$$E \vdash x_i : E(x_i) \quad (1) \qquad \frac{E \vdash p : (\texttt{sig}\ S_1;\ \texttt{module}\ x_i : M;\ S_2\ \texttt{end})}{E \vdash p.x : M\{n_i \leftarrow p.n \mid n_i \in BV(S_1)\}} \ (2) \qquad \frac{E \vdash s : S}{E \vdash (\texttt{struct}\ s\ \texttt{end}) : (\texttt{sig}\ S\ \texttt{end})} \ (3)$$

$$\frac{E \vdash M\ \textsf{modtype} \qquad x_i \notin BV(E) \qquad E;\ \texttt{module}\ x_i : M \vdash m : M'}{E \vdash (\texttt{functor}(x_i : M)m) : (\texttt{functor}(x_i : M)M')} \ (4)$$

$$\frac{E \vdash m_1 : \texttt{functor}(x_i : M)M' \qquad E \vdash m_2 : M}{E \vdash m_1(m_2) : M'\{x_i \leftarrow m_2\}} \ (5)$$

$$\frac{E \vdash M\ \textsf{modtype} \qquad E \vdash m : M}{E \vdash (m : M) : M} \ (6) \qquad \frac{E \vdash m : M' \qquad E \vdash M' <: M}{E \vdash m : M} \ (7) \qquad \frac{E \vdash p : M}{E \vdash p : M/p} \ (8) \qquad E \vdash \varepsilon : \varepsilon \ (9)$$

$$\frac{E \vdash e : \tau \qquad v_i \notin BV(E) \qquad E;\ \texttt{val}\ v_i : \tau \vdash s : S}{E \vdash (\texttt{val}\ v_i = e;\ s) : (\texttt{val}\ v_i : \tau;\ S)} \ (10) \qquad \frac{E \vdash \tau\ \textsf{type} \qquad t_i \notin BV(E) \qquad E;\ \texttt{type}\ t_i = \tau \vdash s : S}{E \vdash (\texttt{type}\ t_i = \tau;\ s) : (\texttt{type}\ t_i = \tau;\ S)} \ (11)$$

$$\frac{E \vdash m : M \qquad x_i \notin BV(E) \qquad E;\ \texttt{module}\ x_i : M \vdash s : S}{E \vdash (\texttt{module}\ x_i = m;\ s) : (\texttt{module}\ x_i : M;\ S)} \ (12) \qquad \frac{\emptyset \vdash (\texttt{struct}\ s\ \texttt{end}) : (\texttt{sig val res : int end})}{\vdash (\texttt{prog}\ s\ \texttt{end})\ \textsf{ok}} \ (13)$$

Module subtyping ($E \vdash M <: M'$):

$$\frac{E \vdash M_2 <: M_1 \qquad E;\ \texttt{module}\ x_i : M_2 \vdash M_1' <: M_2'}{E \vdash \texttt{functor}(x_i : M_1)M_1' <: \texttt{functor}(x_i : M_2)M_2'} \ (14)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \qquad \text{for all } i \in \{1, \ldots, m\},\ E; D_1; \ldots; D_n \vdash D_{\sigma(i)} <: D_i'}{E \vdash \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} <: \texttt{sig}\ D_1'; \ldots; D_m'\ \texttt{end}} \ (15)$$

$$\frac{E \vdash \tau \approx \tau'}{E \vdash (\texttt{val}\ v_i : \tau) <: (\texttt{val}\ v_i : \tau')} \ (16) \qquad \frac{E \vdash M <: M'}{E \vdash (\texttt{module}\ x_i : M) <: (\texttt{module}\ x_i : M')} \ (17)$$

$$E \vdash (\texttt{type}\ t_i = \tau) <: (\texttt{type}\ t_i) \ (18) \qquad E \vdash (\texttt{type}\ t_i) <: (\texttt{type}\ t_i) \ (19)$$

$$\frac{E \vdash \tau \approx \tau'}{E \vdash (\texttt{type}\ t_i = \tau) <: (\texttt{type}\ t_i = \tau')} \ (20) \qquad \frac{E \vdash t_i \approx \tau}{E \vdash (\texttt{type}\ t_i) <: (\texttt{type}\ t_i = \tau)} \ (21)$$

Type equivalence ($E \vdash \tau \approx \tau'$):

$$E_1;\ \texttt{type}\ t_i = \tau;\ E_2 \vdash t_i \approx \tau \ (22) \qquad \frac{E \vdash p : \texttt{sig}\ S_1;\ \texttt{type}\ t_i = \tau;\ S_2\ \texttt{end}}{E \vdash p.t \approx \tau\{n_i \leftarrow p.n \mid n_i \in BV(S_1)\}} \ (23)$$

(Rules for congruence, reflexivity, symmetry, transitivity and base language-dependent equivalences omitted.)

Well-formedness of module types ($E \vdash M$ modtype): rules omitted.

Figure 1: Typing rules

given typing judgements $E \vdash e : \tau$ and $E \vdash \tau$ type for the base language. The "self" rule mentioned in section 2.2 (rule 7) uses a type strengthening operation written M/p, which enriches the module type M to reflect that its abstract type components come from the path p, as follows:

$$\begin{aligned}
(\texttt{sig}\ S\ \texttt{end})/p &= \texttt{sig}\ S/p\ \texttt{end} \\
(\texttt{functor}(x_i : M_1)M_2)/p &= \texttt{functor}(x_i : M_1)(M_2/p(x_i)) \\
\varepsilon/p &= \varepsilon \\
(\texttt{val}\ v_i : \tau;\ S)/p &= \texttt{val}\ v_i : \tau;\ S/p \\
(\texttt{type}\ t_i;\ S)/p &= \texttt{type}\ t_i = p.t;\ S/p \\
(\texttt{type}\ t_i = \tau;\ S)/p &= \texttt{type}\ t_i = p.t;\ S/p \\
(\texttt{module}\ x_i : M;\ S)/p &= \texttt{module}\ x_i : M/p.x;\ S/p
\end{aligned}$$

3.3 Denotational semantics

The dynamic semantics of the module calculus is obtained by erasing all type information and mapping structures to records and functors to functions. This is formalized as a standard denotational semantics shown in figure 2. We assume given a domain B of values for the base language, including a constant **wrong** denoting run-time type errors, and a meaning function $[\![\cdot]\!]_\rho$ for expressions of the base language. Writing $\stackrel{fin}{\to}$ for partial functions with finite domain, the domain V of values for the module language is defined as:

$$\begin{aligned}
V &= (\mathit{ValName} \stackrel{fin}{\to} B) \times (\mathit{ModName} \stackrel{fin}{\to} V) \\
&\quad + (V \to V) + \texttt{wrong}_\perp
\end{aligned}$$

Meaning of module expressions and programs:

$$
\begin{array}{rcl}
[\![x_i]\!]_\rho & = & \text{if } x_i \in \mathrm{Dom}(\rho) \text{ then } \rho(x_i) \text{ else } \mathtt{wrong} \\
[\![p.x]\!]_\rho & = & \text{let } d = [\![p]\!]_\rho \text{ in if } x \in \mathrm{Dom}(d) \text{ then } d(x) \text{ else } \mathtt{wrong} \\
[\![\mathtt{struct}\ s\ \mathtt{end}]\!]_\rho & = & [\![s]\!]_\rho \\
[\![\mathtt{functor}(x_i : M)m]\!]_\rho & = & \lambda d.\ \text{if } d = \mathtt{wrong} \text{ then } \mathtt{wrong} \text{ else } [\![m]\!]_{\rho+\{x_i \mapsto d\}} \\
[\![m_1(m_2)]\!]_\rho & = & \text{if } [\![m_1]\!]_\rho \in V \to V \text{ then } [\![m_1]\!]_\rho([\![m_2]\!]_\rho) \text{ else } \mathtt{wrong} \\
[\![(m : M)]\!]_\rho & = & [\![m]\!]_\rho \\
[\![\varepsilon]\!]_\rho & = & \{\} \\
[\![\mathtt{val}\ v_i = e;\ s]\!]_\rho & = & \text{let } d = [\![e]\!]_\rho \text{ in if } d = \mathtt{wrong} \text{ then } \mathtt{wrong} \text{ else} \\
& & \text{let } d' = [\![s]\!]_{\rho+\{v_i \mapsto d\}} \text{ in if } d' = \mathtt{wrong} \text{ then } \mathtt{wrong} \text{ else } \{v \mapsto d\} + d' \\
[\![\mathtt{type}\ t = \tau;\ s]\!]_\rho & = & [\![s]\!]_\rho \\
[\![\mathtt{module}\ x_i = m;\ s]\!]_\rho & = & \text{let } d = [\![m]\!]_\rho \text{ in if } d = \mathtt{wrong} \text{ then } \mathtt{wrong} \text{ else} \\
& & \text{let } d' = [\![s]\!]_{\rho+\{x_i \mapsto d\}} \text{ in if } d' = \mathtt{wrong} \text{ then } \mathtt{wrong} \text{ else } \{x \mapsto d\} + d' \\
[\![\mathtt{prog}\ s\ \mathtt{end}]\!] & = & \text{let } d = [\![s]\!]_\emptyset \text{ in if } \mathtt{res} \in \mathrm{Dom}(s) \text{ then } s(\mathtt{res}) \text{ else } \mathtt{wrong}
\end{array}
$$

Extension of the meaning function on value expressions:

$$[\![p.v]\!]_\rho = \text{let } d = [\![p]\!]_\rho \text{ in if } v \in \mathrm{Dom}(d) \text{ then } d(v) \text{ else } \mathtt{wrong}$$

Figure 2: Denotational semantics

4 Representation independence

We now show that applicative functors ensure a suitable notion of type abstraction. Following Reynolds [17] and Mitchell [15], we use representation independence as the semantic characterization of type abstraction: a language enforces type abstraction if two implementations of an abstract type can be observationally equivalent (substituting one by the other in any program does not change the outcome of the program), yet use different representation types to implement this abstract type. The proof is an adaptation of Mitchell's proof for the SOL calculus [15], which relies on logical relations.

We first define binary logical relations for the values and type expressions of the module calculus. Logical relations are presented as the judgement $\Gamma \models v \approx v' : M \Rightarrow T$, read: "under the type interpretation Γ, the values v and v' are equivalent when observed under type M and produce the type interpretation T". The type interpretation Γ assigns a meaning to the type identifiers and type paths appearing in M. The T component records the relations we used to prove the equivalence of v and v'. We need T because the elimination construct we use for abstract types (type paths) has open scope, in contrast with Mitchell's `open` construct, which has closed scope.

Let *BaseRel* be the set of admissible relation between base values (binary relations over B that do not relate `wrong` with `wrong`). Define

$$
\begin{array}{l}
\mathit{ModInterp} = \\
\quad (\mathit{TypeName} \xrightarrow{fin} \mathit{BaseRel}) \times (\mathit{ModName} \xrightarrow{fin} \mathit{ModInterp}) \\
\qquad + (\mathit{ModInterp} \to \mathit{ModInterp}) \\
\mathit{InterpEnv} = \\
\quad (\mathit{TypeIdent} \xrightarrow{fin} \mathit{BaseRel}) \times (\mathit{ModIdent} \xrightarrow{fin} \mathit{ModInterp})
\end{array}
$$

Module interpretations $T \in \mathit{ModInterp}$ are either structure interpretations (a record mapping type names to base relations and module names to module interpretations), or functor interpretations (functions from interpretations to interpretations). Interpretation environments $\Gamma \in \mathit{InterpEnv}$ map type identifiers to base relations and module identifiers to module interpretations. An interpretation environment Γ provides a meaning to type paths as admissible relations between base values in the obvious way:

$$
\begin{array}{rcl}
\Gamma(p.t) & = & (\Gamma(p))(t) \\
\Gamma(p.x) & = & (\Gamma(p))(x) \\
\Gamma(p_1(p_2)) & = & (\Gamma(p_1))(\Gamma(p_2))
\end{array}
$$

We first define a family (R^τ_Γ) of logical relations over values of the base language in the usual way:

- $(b_1, b_2) \in R^{\mathtt{int}}_\Gamma$ iff b_1 and b_2 are equal integers
- $(b_1, b_2) \in R^{t_i}_\Gamma$ iff $t_i \in \mathrm{Dom}(\Gamma)$ and $(b_1, b_2) \in \Gamma(t_i)$
- $(b_1, b_2) \in R^{p.t}_\Gamma$ iff $\Gamma(p.t)$ is defined and $(b_1, b_2) \in \Gamma(p.t)$
- $(b_1, b_2) \in R^{\tau\to\sigma}_\Gamma$ iff b_1 and b_2 are functions, and for all base values $(a_1, a_2) \in R^\tau_\Gamma$, we have $(b_1(a_1), b_2(a_2)) \in R^\sigma_\Gamma$

We then define the judgement $\Gamma \models v \approx v' : M \Rightarrow T$ by induction over M, as follows.

1. $\Gamma \models s_1 \approx s_2 : \mathtt{sig}\ S\ \mathtt{end} \Rightarrow T$ iff s_1 and s_2 are semantic signatures such that $\Gamma \models s_1 \approx s_2 : S \Rightarrow T$
2. $\Gamma \models f_1 \approx f_2 : \mathtt{functor}(x_i : M)N \Rightarrow T$ iff f_1, f_2 and T are functions such that for all values v_1, v_2 and interpretations X satisfying $\Gamma \models v_1 \approx v_2 : M \Rightarrow X$, we have
$$\Gamma + \{x_i \mapsto X\} \models f_1(v_1) \approx f_2(v_2) : N \Rightarrow T(X)$$
3. $\Gamma \models s_1 \approx s_2 : \varepsilon \Rightarrow \{\}$ for all semantic signatures s_1 and s_2

4. $\Gamma \models s_1 \approx s_2 : (\texttt{val } v_i : \tau;\ S) \Rightarrow T$ iff $(s_1(v), s_2(v)) \in R^\tau_\Gamma$ and $\Gamma \models s_1 \approx s_2 : S \Rightarrow T$

5. $\Gamma \models s_1 \approx s_2 : (\texttt{type } t_i;\ S) \Rightarrow T$ iff there exists an admissible base relation R and an interpretation T' such that $\Gamma + \{t_i \mapsto R\} \models s_1 \approx s_2 : S \Rightarrow T'$ and $T = \{t \mapsto R\} + T'$

6. $\Gamma \models s_1 \approx s_2 : (\texttt{type } t_i = \tau;\ S) \Rightarrow T$ iff $\Gamma + \{t_i \mapsto R^\tau_\Gamma\} \models s_1 \approx s_2 : S \Rightarrow T'$ and $T = \{t \mapsto R^\tau_\Gamma\} + T'$

7. $\Gamma \models s_1 \approx s_2 : (\texttt{structure } x_i : M;\ S) \Rightarrow T$ iff there exists interpretations T' and T'' such that $\Gamma \models s_1(x) \approx s_2(x) : M \Rightarrow T'$ and $\Gamma + \{x_i \mapsto T'\} \models s_1 \approx s_2 : S \Rightarrow T''$ and $T = \{x \mapsto T'\} + T''$

Notice that in the fifth case (declaration of an abstract type t_i), we do not require t_i to be implemented by the same type expression in the two structures denoted by s and s': we can interpret t_i by any admissible relation, not only R^τ_Γ for some τ, as long as the relation makes the remainder of the two structures related [15].

We define similarly an equivalence relation $\Gamma \models \rho \approx \rho' : E \Rightarrow \Gamma'$ between evaluation environments ρ and ρ' viewed under the typing environment E. The Γ' component is an extension of Γ with interpretations for the identifiers declared in E.

The fundamental lemma of logical relations for the module calculus is as follows:

Proposition 1 *If $E \vdash m : M$ and $\emptyset \models \rho \approx \rho' : E \Rightarrow \Gamma$, then there exists an interpretation T such that*

$$\Gamma \models [\![m]\!]_\rho \approx [\![m]\!]_{\rho'} : M \Rightarrow T.$$

Proof: standard inductive argument on the derivation of $E \vdash m : M$. □

As a corollary, we obtain the representation independence property: two closed module expressions whose meanings are related can be substituted one for the other in any program. The first formulation is as follows:

Proposition 2 *Let m_1 and m_2 be two closed module expressions such that $\emptyset \vdash m_1 : M$ and $\emptyset \vdash m_2 : M$. Assume $\emptyset \vdash [\![m_1]\!]_\emptyset \approx [\![m_2]\!]_\emptyset : M \Rightarrow T$ for some T. For all program contexts $C[\,]$ such that $x_i : M \vdash C[x_i]$ ok, we have $[\![C[m_1]]\!] = [\![C[m_2]]\!]$.*

Proof: Let $D[\,]$ be the structure context such that $C[\,] = \texttt{prog } D[\,] \texttt{ end}$. We have $[\![C[m_1]]\!] = [\![D[m_1]]\!]_\emptyset(\texttt{res})$ and similarly for m_2. Let $\rho_1 = \{x_i \mapsto [\![m_1]\!]_\emptyset\}$ and $\rho_2 = \{x_i \mapsto [\![m_2]\!]_\emptyset\}$ and $E = (x_i : M)$. By hypothesis on m_1 and m_2, we have $\emptyset \models \rho_1 \approx \rho_2 : E \Rightarrow \Gamma$ where $\Gamma = \{x_i \mapsto T\}$. By hypothesis on C, we have $E \vdash (\texttt{struct } D[x_i] \texttt{ end}) : (\texttt{sig val res : int end})$. From the fundamental lemma, it follows that $\Gamma \models [\![\texttt{struct } D[x_i] \texttt{ end}]\!]_{\rho_1} \approx [\![\texttt{struct } D[x_i] \texttt{ end}]\!]_{\rho_2} : (\texttt{sig val res : int end}) \Rightarrow T'$ for some T'. It is easy to check that $[\![D[x_i]]\!]_{\rho_1} = [\![D[m_1]]\!]_\emptyset$, and similarly for m_2 and ρ_2. Extracting the `res` integer field from the denotations of $D[m_1]$ and $D[m_2]$, it follows that $([\![D[m_1]]\!]_\emptyset(\texttt{res}), [\![D[m_2]]\!]_\emptyset(\texttt{res})) \in R^{\texttt{int}}_\Gamma$, which means $[\![C[m_1]]\!] = [\![C[m_2]]\!]$ since $R^{\texttt{int}}_\Gamma$ is the equality relation over integers. □

Proposition 2 is a relatively weak result, because the assumption $x_i : M \vdash C[x_i]$ ok requires the program context $C[\,]$ to be parametric with respect to all possible implementations of the signature M and prevents $C[\,]$ from taking advantage of more typing hypothesis that could be derived about a particular implementation. The following reformulation of proposition 2 shows that even without the hypothesis $x_i : M \vdash C[x_i]$ ok, the applicative functor calculus prevents $C[\,]$ from depending too closely on a particular implementation of M.

In the following statement, we say that a module type M is principal for a module expression m in an environment E if $E \vdash m : M$ and for all types M' and environments E' such that $BV(E') \cap BV(E) = \emptyset$, if $E; E' \vdash m : M'$, then $E; E' \vdash M <: M'$.

Proposition 3 *Let m_1 and m_2 be two closed module expressions and M be a module type. Assume that M is a principal type for m_1 and for m_2 in the empty environment. Then, for all program contexts $C[\,]$, the program $C[m_1]$ is well-typed if and only if $C[m_2]$ is, and if so, $[\![C[m_1]]\!] = [\![C[m_2]]\!]$.*

Proof: Assume $\emptyset \vdash C[m_1]$ ok. Using the fact that M is principal for m_1, we can build a derivation of $x_i : M \vdash C[x_i]$ ok. Applying proposition 2, it follows that $[\![C[m_1]]\!] = [\![C[m_2]]\!]$. □

Two closed modules m_1 and m_2 that have a common principal type and are equivalent at that type are therefore observationally equivalent. Moreover, this condition does not require that the two modules implement their type components by the same representation types. Consider the typical situation:

$$\begin{array}{lcl} M & = & \texttt{sig type t; val x: t end} \\ m_1 & = & \texttt{(struct type t = } \tau_1\texttt{; val x = ... end : M)} \\ m_2 & = & \texttt{(struct type t = } \tau_2\texttt{; val x = ... end : M)} \end{array}$$

If m_1 and m_2 are well-typed, then M is principal for m_1 and m_2. (Since signature constraints are not paths, the "self" rule does not apply to m_1 and m_2, and therefore no type smaller than M can be derived.) Moreover, by definition of logical relations, the denotations of m_1 and m_2 can be related even if τ_1 is incompatible with τ_2. This would not be the case if the class of paths were extended further, e.g. by allowing any module expression in paths. Then, the "self" rule would apply to the definitions of m_1 and m_2, allowing the derivation of the following typings:

$$\begin{array}{lcl} m_1 & : & \texttt{sig type t = } m_1\texttt{.t; val x: t end} \\ m_2 & : & \texttt{sig type t = } m_2\texttt{.t; val x: t end} \end{array}$$

and M would no longer be the principal type of m_1 and m_2. The hypothesis that m_1 and m_2 have the same principal type would force m_1 and m_2 to be syntactically identical, hence τ_1 and τ_2 to be identical as well.

This situation cannot happen in the applicative functor calculus: m_1 and m_2 in proposition 3 cannot be paths since they must be closed, while paths $p ::= x_i \mid p.x \mid p_1(p_2)$ always have at least one free structure identifier. Hence, the "self" rule does not apply to m_1 and m_2, thus m_1 and m_2 can have the same principal type while implementing abstract type components differently.

This discussion shows that introducing functor applications in paths does not compromise representation independence as characterized by proposition 3, but further extensions of the class of paths could.

Typing rules:

$$E \vdash x_i : E(x_i) \quad (24) \qquad \frac{E;\ x_i : M_1 \vdash m : M_2}{E \vdash (\lambda x_i : M_1.\, m) : (\Pi x_i : M_1.\, M_2)} \quad (25) \qquad \frac{E \vdash m_1 : \Pi x_i : M.\, M' \qquad E \vdash m_2 : M}{E \vdash m_1(m_2) : M'\{x_i \leftarrow m_2\}} \quad (26)$$

$$\frac{E \vdash m : M \qquad E \vdash M <: M'}{E \vdash m : M'} \quad (27) \qquad \frac{E \vdash m : M}{E \vdash m : M/m} \quad (28) \qquad \frac{E \vdash e : \tau}{E \vdash \iota_v(e) : \mathtt{V}(\tau)} \quad (29) \qquad \frac{E \vdash \tau\ \mathsf{type}}{E \vdash \iota_t(\tau) : \mathtt{EQ}(\tau)} \quad (30)$$

$$\frac{E \vdash m_1 : M_1 \qquad E;\ x_i : M_1 \vdash m_2 : M_2}{E \vdash \langle x_i = m_1,\, m_2 \rangle : (\Sigma x_i : M_1.\, M_2)} \quad (31) \qquad \frac{E \vdash m : \Sigma x_i : M_1.\, M_2}{E \vdash \pi_1(m) : M_1} \quad (32) \qquad \frac{E \vdash m : \Sigma x_i : M_1.\, M_2}{E \vdash \pi_2(m) : M_2\{x_i \leftarrow \pi_1(m)\}} \quad (33)$$

Subtyping and type equivalence: (rules omitted: congruence, transitivity and symmetry for $\approx$, transitivity for $<:$)

$$\frac{E \vdash m : \mathtt{EQ}(\tau)}{E \vdash \pi_t(m) \approx \tau} \quad (34) \qquad \frac{E \vdash M \approx M'}{E \vdash M <: M'} \quad (35) \qquad \frac{E \vdash \tau\ \mathsf{type}}{E \vdash \mathtt{EQ}(\tau) <: \mathtt{TYPE}} \quad (36)$$

$$\frac{E \vdash M_1 <: M_1' \qquad E;\ x_i : M_1 \vdash M_2 <: M_2'}{E \vdash (\Sigma x_i : M_1.\, M_2) <: (\Sigma x_i : M_1'.\, M_2')} \quad (37) \qquad \frac{E \vdash M_1' <: M_1 \qquad E;\ x_i : M_1' \vdash M_2 <: M_2'}{E \vdash (\Pi x_i : M_1.\, M_2) <: (\Pi x_i : M_1'.\, M_2')} \quad (38)$$

Type strengthening:

$$\mathtt{V}(\tau)/m = \mathtt{V}(\tau) \qquad \mathtt{TYPE}/m = \mathtt{EQ}(\pi_t(m)) \qquad \mathtt{EQ}(\tau)/m = \mathtt{EQ}(\pi_t(m))$$
$$(\Sigma x_i : M_1.\, M_2)/m = \Sigma x_i : (M_1/\pi_1(m)).\, (M_2/\pi_2(m)) \qquad (\Pi x_i : M_1.\, M_2)/m = \Pi x_i : M_1.\, (M_2/m(x_i))$$

Figure 3: Typing rules for the manifest sums calculus

5 Full transparency for higher-order functors

In this section, we prove that the module calculus with applicative functors has fully transparent higher-order functors, in the sense of MacQueen and Tofte [13]. One way to prove this result is to take their static semantics for higher-order functors and show that all correct programs in this semantics are well-typed in our calculus. We have not been able to show this result due to the complexity of their formalism. Instead, we will show that the calculus with applicative functors can encode a stratified calculus with strong sums similar to MacQueen's DL calculus and Harper and Mitchell's XML calculus [12, 9]. Strong sums account for many features of the SML module system, excluding generativity but including transparent type bindings and fully transparent higher-order functors. The existence of a type-preserving encoding into the calculus with applicative functors is therefore a strong hint that the latter ensures full transparency.

5.1 The manifest sums calculus

To simplify the encoding, we start from the "manifest sums" calculus, a variant of the strong sums calculus that differs on the way type equalities are propagated (through compile-time reductions of terms in the strong sums calculus, but through enriched types in the manifest types calculus), but has the same expressive power. All terms well-typed with strong sums are also well-typed with manifest sums; we omit the proof, which is along the lines of the proof of proposition 1 in [10]. The manifest sums calculus has the following syntax:

Terms:
$m ::= x_i \mid \lambda x_i : M.\, m \mid m_1(m_2) \mid \iota_v(e) \mid \iota_t(\tau)$
$\quad \mid \langle x_i = m_1,\, m_2 \rangle \mid \pi_1(m) \mid \pi_2(m)$

Types:
$M ::= \mathtt{V}(\tau) \mid \mathtt{TYPE} \mid \mathtt{EQ}(\tau) \mid \Sigma x_i : M_1.\, M_2 \mid \Pi x_i : M_1.\, M_2$

Simple terms:
$e ::= \ldots \mid \pi_v(m)$

Simple types:
$\tau ::= \ldots \mid \pi_t(m)$

Structures are built from injections $\iota_v(e)$ of simple terms (values) and $\iota_t(\tau)$ of simple types using the dependent pair operator $\langle x_i = m_1,\, m_2 \rangle$. The corresponding signatures are $\mathtt{V}(\tau)$ for a value of type τ, $\mathtt{EQ}(\tau)$ for a type manifestly equal to τ, $\mathtt{TYPE}$ for an arbitrary type, and Σ-types (dependent pair types). Access inside structures is provided by the projections π_v for values, π_t for types, π_1 and π_2 for pair components. Functors are presented by λ-abstractions and Π-types (dependent function types). The typing rules for the calculus are shown in figure 3.

5.2 Path normalization

In preparation for the encoding into the applicative functor calculus, we first show how to rewrite terms to meet the syntactic restrictions imposed by the latter, such as the restriction of projections to paths. Paths p in the manifest sums calculus are described by the grammar

$$p ::= x_i \mid \pi_1(p) \mid \pi_2(p) \mid p_1(p_2).$$

To express the rewriting more easily, we extend the syntax of terms and types with a `let` binding (i.e. explicit substitutions):

Terms: $\quad m ::= \ldots \mid \mathtt{let}\ \sigma\ \mathtt{in}\ m$
Types: $\quad M ::= \ldots \mid \mathtt{Let}\ \sigma\ \mathtt{in}\ M$

Substitutions: $\sigma ::= \varepsilon \mid x_i = m; \sigma$

For the purposes of type-checking and evaluation, $\texttt{let}\ x_1 = m_1; \ldots; x_n = m_n\ \texttt{in}\ m$ is treated as the textual substitution $m\{x_n \leftarrow m_n\} \ldots \{x_1 \leftarrow m_1\}$.

The first group of rewrite rules introduce names for arguments to projections that are not paths. In the following, m_c ranges over terms that are not paths.

Over terms:

$$\begin{array}{rcl}
\iota_v(e[\pi_v(m_c)]) & \to & \texttt{let}\ x_i = m_c\ \texttt{in}\ \iota_v(e[\pi_v(x_i)]) \\
\iota_t(\tau[\pi_t(m_c)]) & \to & \texttt{let}\ x_i = m_c\ \texttt{in}\ \iota_t(\tau[\pi_t(x_i)]) \\
\pi_1(m_c) & \to & \texttt{let}\ x_i = m_c\ \texttt{in}\ \pi_1(x_i) \\
\pi_2(m_c) & \to & \texttt{let}\ x_i = m_c\ \texttt{in}\ \pi_2(x_i) \\
m(m_c) & \to & \texttt{let}\ x_i = m_c\ \texttt{in}\ m(x_i)
\end{array}$$

Over types:

$$\begin{array}{rcl}
\texttt{V}(\tau[\pi_t(m_c)]) & \to & \texttt{Let}\ x_i = m\ \texttt{in}\ \texttt{V}(\tau[\pi_t(x_i)]) \\
\texttt{EQ}(\tau[\pi_t(m_c)]) & \to & \texttt{Let}\ x_i = m\ \texttt{in}\ \texttt{EQ}(\tau[\pi_t(x_i)])
\end{array}$$

The `let` bindings have no equivalent in the applicative functor calculus, unless they occur immediately below a pair construction, in which case they can be translated as extra bindings in a `struct...end`. The second group of rules lift `let` bindings upwards until they hit a pair operator.

Over terms:

$$\begin{array}{rcl}
\texttt{let}\ \sigma\ \texttt{in}\ \texttt{let}\ \sigma'\ \texttt{in}\ m & \to & \texttt{let}\ \sigma; \sigma'\ \texttt{in}\ m \\
(\texttt{let}\ \sigma\ \texttt{in}\ m_1)(m_2) & \to & \texttt{let}\ \sigma\ \texttt{in}\ m_1(m_2) \\
\lambda x_i : M.\ \texttt{let}\ \sigma\ \texttt{in}\ m & \to & \texttt{let}\ \lambda x_i : M. \sigma\ \texttt{in}\ \sigma(m, x_i) \\
\lambda x_i : (\texttt{Let}\ \sigma\ \texttt{in}\ M). m & \to & \texttt{let}\ \sigma\ \texttt{in}\ \lambda x_i : M. m
\end{array}$$

Over types:

$$\begin{array}{rcl}
\texttt{Let}\ \sigma\ \texttt{in}\ \texttt{Let}\ \sigma'\ \texttt{in}\ M & \to & \texttt{Let}\ \sigma; \sigma'\ \texttt{in}\ M \\
\Sigma x_i : (\texttt{Let}\ \sigma\ \texttt{in}\ M_1). M_2 & \to & \texttt{Let}\ \sigma\ \texttt{in}\ \Sigma x_i : M_1. M_2 \\
\Sigma x_i : M_1. (\texttt{Let}\ \sigma\ \texttt{in}\ M_2) & \to & \texttt{Let}\ \lambda x_i : M_1. \sigma\ \texttt{in} \\
 & & \Sigma x_i : M_1. \sigma(M_2, x_i) \\
\Pi x_i : (\texttt{Let}\ \sigma\ \texttt{in}\ M_1). M_2 & \to & \texttt{Let}\ \sigma\ \texttt{in}\ \Pi x_i : M_1. M_2 \\
\Pi x_i : M_1. (\texttt{Let}\ \sigma\ \texttt{in}\ M_2) & \to & \texttt{Let}\ \lambda x_i : M_1. \sigma\ \texttt{in} \\
 & & \Pi x_i : M_1. \sigma(M_2, x_i)
\end{array}$$

Over substitutions:

$$x_i = (\texttt{let}\ \sigma\ \texttt{in}\ m);\ \sigma' \quad \to \quad \sigma;\ x_i = m;\ \sigma'$$

To express the introduction of abstractions and applications when a `let`-binding crosses a binder (λ, Σ or Π), we have used the following notations: if σ is the substitution $y_1 = m_1; \ldots; y_n = m_n$, we write

$$\begin{array}{rcl}
\lambda x_i : M. \sigma & \text{for} & y_1 = \lambda x_i : M. m_1; \ldots; y_n = \lambda x_i : M. m_n \\
\sigma(m, x_i) & \text{for} & m\{y_1 \leftarrow y_1(x_i) \ldots y_n \leftarrow y_n(x_i)\}
\end{array}$$

It can be shown that the rewrite rules above preserve typing: if $E \vdash m : M$ and $m \to m'$, then $E \vdash m' : M$. Moreover, a term in normal form with respect to the rules above is such that: all projections are applied to paths; all functions arguments in applications are paths; no `Let` bindings remain; and `let` bindings occur only in toplevel position or as arguments to a pair construction. Assuming without loss of generality that complete programs have a pair in toplevel position, and identifying m with $\texttt{let}\ \varepsilon\ \texttt{in}\ m$ as needed, we can therefore describe normalized programs by the following grammar:

Paths:
$p ::= x_i \mid \pi_1(p) \mid \pi_2(p) \mid p_1(p_2)$

Normalized terms:
$m ::= p \mid \lambda x_i : M. m \mid m(p) \mid \iota_v(e) \mid \iota_t(\tau)$
$\quad \mid \langle x_i = (\texttt{let}\ \sigma_1\ \texttt{in}\ m_1), \texttt{let}\ \sigma_2\ \texttt{in}\ m_2\rangle$

Simple terms:
$e ::= \ldots \mid \pi_v(p)$

Simple types:
$\tau ::= \ldots \mid \pi_t(p)$

5.3 Encoding

Normalized terms are then encoded as module expressions from the applicative functor calculus by turning injections and pairs into structures with conventional field names (`v` for values, `t` for types, `fst` and `snd` for pair components). The encoding, written $\lfloor \cdot \rfloor$, is defined as:

$$\begin{array}{rcl}
\lfloor x_i \rfloor & = & x_i \\
\lfloor \pi_1(p) \rfloor & = & \lfloor p \rfloor.\texttt{fst} \\
\lfloor \pi_2(p) \rfloor & = & \lfloor p \rfloor.\texttt{snd} \\
\lfloor \lambda x_i : M. m \rfloor & = & \texttt{functor}(x_i : \lfloor M \rfloor)\lfloor m \rfloor \\
\lfloor m(p) \rfloor & = & \lfloor m \rfloor(\lfloor p \rfloor) \\
\lfloor \iota_v(e) \rfloor & = & \texttt{struct val v}_i = \lfloor e \rfloor\ \texttt{end} \\
\lfloor \iota_t(\tau) \rfloor & = & \texttt{struct type t}_i = \lfloor \tau \rfloor\ \texttt{end}
\end{array}$$

$$\begin{array}{l}
\lfloor \langle x_i = (\texttt{let}\ \sigma_1\ \texttt{in}\ m_1), \texttt{let}\ \sigma_2\ \texttt{in}\ m_2\rangle \rfloor = \\
\quad \texttt{struct}\ \lfloor \sigma_1 \rfloor; \\
\qquad \texttt{module fst}_j = \lfloor m_1 \rfloor; \\
\qquad \lfloor \sigma_2 \rfloor\{x_i \leftarrow \texttt{fst}_j\}; \\
\qquad \texttt{module snd}_k = \lfloor m_2 \rfloor\{x_i \leftarrow \texttt{fst}_j\} \\
\quad \texttt{end}
\end{array}$$

The encoding of a substitution is a sequence of module bindings: $\lfloor x_1 = m_1;\ \ldots;\ x_n = m_n \rfloor$ is ($\texttt{module}\ x_1 = \lfloor m_1 \rfloor;\ \ldots;\ \texttt{module}\ x_n = \lfloor m_n \rfloor$). Types are translated to module types and signatures as follows:

$$\begin{array}{rcl}
\lfloor \texttt{V}(\tau) \rfloor & = & \texttt{sig val v}_i : \lfloor \tau \rfloor\ \texttt{end} \\
\lfloor \texttt{TYPE} \rfloor & = & \texttt{sig type t}_i\ \texttt{end} \\
\lfloor \texttt{EQ}(\tau) \rfloor & = & \texttt{sig type t}_i = \lfloor \tau \rfloor\ \texttt{end} \\
\lfloor \Sigma x_i : M_1. M_2 \rfloor & = & \texttt{sig module fst}_j : \lfloor M_1 \rfloor; \\
 & & \quad \texttt{module snd}_k : \lfloor M_2 \rfloor\{x_i \leftarrow \texttt{fst}_j\} \\
 & & \texttt{end} \\
\lfloor \Pi x_i : M_1. M_2 \rfloor & = & \texttt{functor}(x_i : \lfloor M_1 \rfloor)\lfloor M_2 \rfloor
\end{array}$$

Finally, for projections inside values and simple types, we take

$$\lfloor \pi_v(p) \rfloor = \lfloor p \rfloor.\texttt{v} \qquad \lfloor \pi_t(p) \rfloor = \lfloor p \rfloor.\texttt{t}$$

We then show that this encoding is type-preserving: if $E \vdash m : M$ in the manifest sums calculus and E, m, M are normalized, then $\lfloor E \rfloor \vdash \lfloor m \rfloor : \lfloor M \rfloor$ in the applicative functor calculus. This completes the proof that our calculus with applicative functors can express strong sums.

6 Related work

Several semantics for fully transparent higher-order functors have been given, first in type-theoretic frameworks based on strong sums [12, 9], then as extensions of SML's stamp-based static semantics [13, 2]. Only MacQueen and Tofte's formalism [13] handles the full SML module language; both the strong sums-based models [12, 9] and Biswas's static semantics based on higher-order variables [2] fail to account for generative type definitions. Unfortunately, MacQueen and Tofte's description is technically involving, in part because it is oriented towards an efficient implementation (functor re-elaboration is minimized). The present paper provides a simpler description of fully transparent higher-order functors with both generative and non-generative type bindings; the main missing SML feature is structure generativity and sharing.

Among the formalisms mentioned above, ours is the only one that provides complete syntactic signatures for higher-order functors (syntactic signatures that captures exactly their transparent behavior), which are required to support Modula-2 style separate compilation. Crégut [6] also attacks the problem of complete syntactic signatures for higher-order functors. His proposal relies on enriching signatures with equalities between structures (ours uses only equalities between types), which has the advantages of accounting for structure sharing and remaining compatible with the standard generative semantics of functor application, and the disadvantage of requiring a rather complex stamp-based formalism.

A notion of functors with applicative semantics appears in Rouaix's Alcool language [18], which combines Haskell-style dynamic overloading with a type abstraction mechanism. In the presence of dynamic overloading, applicative functors arise naturally as a generalization of ML's parameterized type constructors such as `list`: when types are equipped with dictionaries of functions implementing overloaded symbols at that type, parameterized types become functors from types plus dictionaries to types plus dictionaries. Since type constructors are applicative by definition (the types τ_1 `list` and τ_2 `list` are equal as soon as $\tau_1 = \tau_2$), these functors naturally have applicative semantics. The applicative semantics of functors might therefore prove useful to account for type classes or Alcool-style abstract types by translation to a language with structures and functors.

7 Conclusions

The applicative semantics of functor applications seems useful to increase the expressive power of the SML module system, and has very little impact both on the semantic properties of the language and on the complexity of its type system. It supports precise syntactic signatures for fully transparent higher-order functors, which facilitates separate compilation and provides a simple formalization of full transparency. On the negative side, the applicative semantics precludes modules as first-class values; also, existing stamp-based type-checkers cannot easily be modified to implement it. More practical experience with the applicative semantics is needed to assess its impact on the modular programming style.

References

[1] A. W. Appel and D. B. MacQueen. Separate compilation for Standard ML. In *Programming Language Design and Implementation 1994*, pages 13–23. ACM Press, 1994.

[2] S. K. Biswas. Higher-order functors with transparent signatures. In *22nd symp. Principles of Progr. Lang.* ACM Press, 1995.

[3] L. Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal description of programming concepts*, pages 431–507. Springer-Verlag, 1989.

[4] L. Cardelli and X. Leroy. Abstract types and the dot notation. In M. Broy and C. B. Jones, editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, 1990.

[5] L. Cardelli and D. B. MacQueen. Persistence and type abstraction. In M. P. Atkinson, P. Buneman, and R. Morrison, editors, *Data types and persistence.* Springer-Verlag, 1988.

[6] P. Crégut. Compilation séparée pour un langage de modules avec types génératifs, Sept. 1994. Presentation given at the 1994 meeting of the G.D.R. "Programmation", C.N.R.S.

[7] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symp. Principles of Progr. Lang.*, pages 123–137. ACM Press, 1994.

[8] R. Harper, R. Milner, and M. Tofte. A type discipline for program modules. In *TAPSOFT 87*, volume 250 of *LNCS*, pages 308–319. Springer-Verlag, 1987.

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

[10] X. Leroy. Manifest types, modules, and separate compilation. In *21st symp. Principles of Progr. Lang.*, pages 109–122. ACM Press, 1994.

[11] X. Leroy. A syntactic approach to type generativity and sharing (extended abstract). In *Proc. 1994 Workshop on ML and its applications*, pages 1–12. Research report 2265, INRIA, 1994.

[12] D. B. MacQueen. Using dependent types to express modular structure. In *13th symp. Principles of Progr. Lang.*, pages 277–286. ACM Press, 1986.

[13] D. B. MacQueen and M. Tofte. A semantics for higher-order functors. In D. Sannella, editor, *Programming languages and systems – ESOP '94*, volume 788 of *LNCS*, pages 409–423. Springer-Verlag, 1994.

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

[15] 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.

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

[17] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983.

[18] F. Rouaix. *The ALCOOL 90 report.* INRIA, 1992. Included in the distribution available on `ftp.inria.fr`.