A Type Discipline for Program Modules

Robert Harper Robin Milner Mads Tofte
Laboratory for Foundations of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ
UNITED KINGDOM

Abstract

The ML modules system is organized around the notions of *structure*, *signature*, and *functor*. A structure is an encapsulated declaration of data types and values, a signature is a "type" or specification of a structure, and a functor is a function taking structures to structures. We present a static semantics for a fragment of this system in the style of Plotkin's operational semantics. The treatment of structures and signatures has interesting parallels with the type assignment rules for ML given by Damas and Milner. In particular there is a notion of principal typing.

1 Introduction

The ML modules system [Mac86a] is an approach to solving some of the problems that arise in organizing and maintaining large ML programs. The approach taken in ML, as in many other languages, is to define a calculus of environments whereby the programmer may decompose a program into relatively independent components with a well–defined interface. These program units are variously called *modules*, *packages*, and *clusters*; in ML we use the term *structure* to suggest both the idea of an "environment structure" and the mathematical notion of an algebraic structure. It is also standard to ascribe some form of "type" to a module. Such specifications go by names such as *interface* or *package description*; in ML we use the word *signature*, by analogy with the signature of an algebraic structure and with established usage in algebraic data type specifications. Many languages supporting modularization have the ability to define what are often called *parameterized modules*. In ML these appear as functions from structures to structures, called *functors*, a term that suggests their functional character.

The tension between the desire to decompose a program into units that may be understood and maintained in relative isolation from the other components, and the desire to combine these components into a coherent unit is perhaps the fundamental problem in modular program construction. MacQueen's analysis of these problems reveals that, in large measure, they can be expressed as problems of *sharing*. The idea is that two program units may come to depend on one another primarily by sharing a common subunit. For example, the common subunit may allocate, say, a stack, and define a collection of operations on it. Two program units may then cooperate with one another by virtue of the fact that they share a common stack. This is represented in ML by the use of *substructures*, the inclusion of a structure as a component of another. Two structures that include the same substructure are said to share that structure.

This analysis of dependence hinges on an appropriate notion of structure equality. Suppose that the stack structure is implemented by use of a reference to the heap (ML has a primitive similar to the **new** operation in Pascal). Then if two program units are to use a stack cooperatively, it is clear that they must use the *same* stack, so the definition of structure equality must be at least as fine–grained as equality of references. Similar problems of structure equality arise in connection with ML's data types and exceptions, and so the problem is not simply one of references. In order to capture these intuitions the ML convention on structure equality is that two structures are equal if and only if they arise from the same evaluation of the same structure expression.

The problem of sharing is most acute when we consider the dynamics of program construction. As static entities, programs in ML are organized as a hierarchical collection of structures, where the

hierarchy is defined in terms of the substructure relationship. As such it would appear unproblematic to ensure that proper sharing is maintained, for all that is necessary is for the programmer to combine the structures together properly, that is, in accordance with the desired sharing between structures. But when we consider the dynamics of program development, it is clear that we must provide some assistance with sharing management. One of the principal reasons for wanting to treat structures independently is so that one structure can be modified in relative isolation from the others. For example, if we correct an error in the stack structure, then we wish to relink (rather than recompile!) the rest of the program with the new stack structure built in. Functors provide the means of performing this relinking: each structure is built as a function of its substructures, and the entire program hierarchy is constructed by applying these structures "bottom up", structures on which a given structure depends being built earlier than that structure.

For example, if structures A and B both employ the stack structure S, then they are built as functions of S by functors F_A and F_B. The structure A is constructed by applying the functor F_A to S, and similarly for B. Notice that functors give us the ability to define several "instances" or "versions" of a structure by applying the appropriate functor more than once. Now if A and B are to be used cooperatively in some structure C, then it is crucial that the functor F_C that is used to build C be applied to compatible instances of A and B, namely those that are built on the *same* stack S. In ML this is expressed by attaching a *sharing specification* to the parameter of F_C that specifies the required sharing between its arguments.

Informal examples of these problem and how they are solved in ML may be found in the papers of MacQueen [Mac86a,Mac86b] and Harper [Har86]. The purpose of this paper is to provide a formal grounding for these ideas. For the sake of perspicuity we limit our attention to a fragment of the full language, called ModL. This fragment consists only of of structures, signatures, and functors, with the entire core language of ML left out. This subset language is sufficiently rich to illustrate the issues that arise in connection with sharing without introducing any of the complexities that are irrelevant for our purposes.

In Section 2 we give some examples of the use of ModL to express modular structure, and make more precise the issues that arise in connection with structure equality and sharing. Then in Section 3 we present a formal static semantics of ModL. By "static" semantics, we mean a collection of inference rules (presented in Plotkin's operational style [Plo81]) that specifies the compile–time correctness conditions on a valid ModL program. In the full language, this encompasses type checking, but here we concentrate on those aspects that involve modules. This semantics bears a close resemblance to the type assignment rules for the core language of ML, as given by Damas and Milner [DM82].

2 Overview of ModL

The syntax of our skeletal language ModL is given in Figure 1. Each syntactic category is defined in terms of its typical elements; the set of phrases in a category is written as a capitalized version of the typical element. For example, *strexp* is a typical element of the set StrExp of structure expressions. The set of identifiers is graded into three sets, StrId for structure identifiers, SigId for signature identifiers, and FunId for functor identifiers. These sets are ranged over by *strid*, *sigid*, and *funid*, respectively. The set Path, of which *path* is a typical member, is the set of finite non–empty sequences of structure identifiers, which we write as a dot–separated list. Paths are sometimes called *qualified names*.

A program is a sequence of declarations, functor bindings, and signature bindings. The basic form of structure expression is an encapsulated declaration, bracketed by `struct` and `end`. Informally, a structure expression denotes an environment assigning meaning to identifiers. In the full language a structure may contain bindings for variables, types, exceptions, and other structures.

dec	::=	`structure` *strid* = *strexp*	structure binding
		$dec_1; \ldots ; dec_k$	sequence, $k \geq 0$
strexp	::=	`struct` *dec* `end`	basic
		path	qualified name
		funid(*strexp*)	functor application
spec	::=	`structure` *strid* : *sigexp*	basic
		$spec_1; \ldots ; spec_k$	sequence, $k \geq 0$
		spec `sharing` *path*=*path*	equation
sigexp	::=	`sig` *spec* `end`	signature expression
		sigid	signature identifier
prog	::=	*dec*	declaration
		`signature` *sigid* = *sigexp*	signature binding
		`functor` *funid*(*strid*: *sigexp*) = *strexp*	functor binding
		prog ; *prog*	sequence

Figure 1: Syntax of ModL

Example 1 (Structure Declaration)

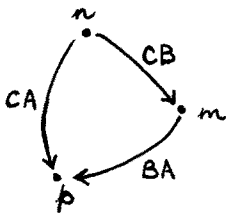
```
structure A =
  struct
    type t = int;
    fun fact(x) = if x=1 then 1 else x*fact(x-1)
  end;

structure B =
  struct
    structure BA = A;
    fun f(x) = BA.fact(x) * BA.fact(x)
  end;

structure C =
  struct
    structure CA = A;
    structure CB = B;
    fun g(x) = CB.f( CA.fact(x) )
   end
```

Structure A defines a type t and a function fact. Structure B incorporates A as a substructure, and in addition defines a function f in terms of fact. The qualified name BA.fact designates the identifier fact in structure BA. Similarly, C incorporates both A and B as substructures, and defines a function g in terms of CB.f and CA.fact. Note that both B and C incorporate the same structure A.

In our skeletal language ModL we throw away all but substructure declarations within structures. One effect of this is that a structure can be represented as a directed acyclic graph with edges labeled by the substructure identifiers, acyclic because a sturcture cannot be a substructure of itself. Each node of the graph is labeled by the *name* of the structure, which we think of as a kind of absolute designator, or address, for the structure. In the static semantics of ModL, structures are evaluated to just such a DAG, with each structure having its own unique name. For example,

we might depict structure C (with all but substructures omitted) as in the following example where n is the name of the structure bound to C etc.

Example 2



Signatures are specifications of structures. In the full language a specification of a structure describes the types of variables and exceptions, the arities of type constructors, and the signatures of substructures. The basic form of signature expression is an encapsulated *specification*, bracketed by `sig` and `end`. In ModL the only form of specification is the *structure specification*, whereby a signature is attached to a structure identifier.

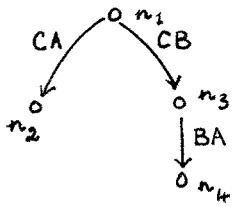
Example 3 (Signature Declaration)

```
signature SIGA =
  sig
    type t;
    fun fact : int -> int
  end;
signature SIGB =
  sig
    structure BA : SIGA;
    fun f : int -> int
  end;
signature SIGC =
  sig
    structure CA : SIGA;
    structure CB : SIGB
  end
```

The precise definition of what it means for a structure to satisfy, or *match*, a signature is given by the static semantics below. Roughly speaking, the structure must have *at least* the components appearing in the signature, and these components must satisfy the specification of that component. Structure `B` matches `SIGB` because the substructure `BA` matches `SIGA`, and `f` is indeed of type `int->int`. Note that structure `C` matches `SIGC`, even though `SIGC` makes no mention of the component `g` of `C`. Since ModL structures contain only structure bindings, signatures specify only structure signatures, and signature matching, at the current level of detail, reduces to recursively matching substructures against their signatures. (This definition makes sense because any structure matches `sig end`, the empty signature.)

Signatures may also be depicted as DAG's, but there is an important difference as compared with those of structures. Whereas for structures the substructure identifiers designate *fixed* structures, in signatures they are merely formal names that are instantiated during signature matching. The idea is that a structure matches a signature if there is an instantiation of the formal names in the DAG of the signature such that the resulting DAG can be expanded (by adding new edges and nodes) to the DAG of the structure. For example, the following is the DAG representation of signature `SIGC`.

Example 4



In DAGs representing signatures we draw circles for the nodes with formal names to distinguish them from the nodes with "actual" names used in structure diagrams. In the semantics this will be represented by the fact that the formal names are bound whereas the actual names are free. In fact this is the entire difference between the representations of structures and signatures in the static semantics.

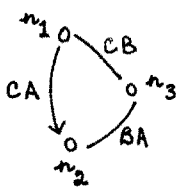
Signatures may contain *sharing specifications* that specify that a certain equation must hold between structures. For example, the following signature is matched by the structure C defined in Example 1.

Example 5 (Signature with Sharing)

```
signature SIGC' =
  sig
    structure CA : SIGA;
    structure CB : SIGB
    sharing CA = CB.BA
  end
```

SIGC' requires that the structures bound to CA and CB.BA be equal. The representation of SIGC' differs from that of SIGC in that it uses the same formal name for CA and CB.BA, as can been seen in the following diagram.

Example 6



Any instance of this DAG must have the same name for these two components, and thus this convention accurately reflects the intended meaning of the sharing specification.

Sharing specifications may involve non–local structures, thereby limiting the structures that match that specification to a single, fixed structure.

Example 7 (Non–local Sharing)

```
signature SIG =
  sig
    structure X : SIGA
    sharing X = A
  end
```

This signature can be matched only by structures that have the structure A bound to X. Diagrammatically, this shows up as an uncircled node, since the sharing specification forces the name for X to be the name of A.

m, n	$\in$	Names
M, N	$\in$	Fin(Names) i.e., finite subsets of Names
E	$\in$	Env = StrId $\xrightarrow{\text{fin}}$ Str
S *or* (n, E)	$\in$	Str = Names $\times$ Env
Σ *or* $(N)S$	$\in$	Sig = Fin(Names) $\times$ Str
$[strid, \Sigma, strexp, \mathcal{E}]$	$\in$	Fun = StrId $\times$ Sig $\times$ StrExp $\times$ ProgEnv
F	$\in$	FunEnv = FunId $\xrightarrow{\text{fin}}$ Fun
G	$\in$	SigEnv = SigId $\xrightarrow{\text{fin}}$ Sig
$\mathcal{E}$ *or* (M, F, G, E)	$\in$	ProgEnv = Fin(Names) $\times$ FunEnv $\times$ SigEnv $\times$ Env

Table 1: Semantic Objects

Functors are functions taking structures to structures. In ModL each functor has a single parameter with a specified signature (there is no "signature inference" for functors).

Example 8 (Functor Declaration)

```
functor F( P: SIGC' ) =
  struct
    structure X = P.CA;
    fun f(y) = X.fact(y)+1
  end
```

Each evaluation of a `struct...end` expression yields a new structure (*i.e.*, a structure with a new name). For instance each application of `F` yields a new structure.

Example 9 (Functor Application)

```
structure D1 = F( C );
structure D2 = F( C )
```

These two declarations yield two isomorphic but unequal structures.

3 Static Semantics of ModL

In this section we present a formal static semantics for ModL. The semantics will be given in terms of a set of inference rules. Some of the objects occurring in the rules are syntactic and they were defined in the last section. In Section 3.1 we define the remaining objects, called the *semantic objects*. Then we present the rules.

3.1 Semantic Objects

Table 1 contains the recursive definition of the sets of semantic objects together with the variables we shall use to range over them.

In the present section, we shall use the word *structure* to mean a member of the set Str. A structure, S, has the form (n, E), where $n \in$ Names and $E \in$ Env; n is the *name* of the structure, and E is its *environment* binding structures to structure identifiers, one binding for each substructure of S — the set Names of names can be any infinite set. In a richer language, environments would contain other bindings as well (for instance bindings to type and value identifiers).

In terms of the DAG representation introduced in the last section, each node corresponds to a structure, the label of the node is the name of the structure, and the edges emanating from a node are determined by the environment component.

Two structures are equal if and only if their names are equal *and* their environment components are equal as finite functions. This definition reflects the fact that names are used to define a finer–grained equality than that induced by the equality on environments. Notice that the equality is also finer than simply name equality.

A *signature*, Σ, is a pair $(N)S$ consisting of a finite set of names together with a structure. (In the present section, the word *signature* will always have this technical meaning.) The set N binds names within S. Not all names appearing in S need be bound by N; free names in a signature are used to represent non–local sharing. The circled nodes in the diagrammatic representation of a signature are those that are bound by N here.

Functors are represented as *(functor) closures* consisting of the parameter identifier, its signature, the text of the functor body, and the environment in which it was defined. The need for closures is as usual: the body of a functor may contain free identifiers, and so must be packaged with its environment of definition if capture is to be avoided.

A program environment $\mathcal{E}$ is a quadruple (M, F, G, E) consisting of a finite set of names (intuitively, the set of "fixed" names), a functor environment mapping functor identifiers to functor closures, a signature environment mapping signature identifiers to signatures, and an environment. Environments are extended to paths as follows:

$$E(path) = \begin{cases} E(strid) & path = strid \\ E'(path') & path = strid.path',\ E(strid) = (n, E') \end{cases}$$

Structure environments E are extended similarly. We often write $E.path$ for $E(path)$, and $S.path$ for $E.path$, when $S = (n, E)$.

Environments are combined using the "+" operator, defined as follows:

$$(E + E')(i) = \begin{cases} E'(i) & i \in \mathrm{Dom}(E') \\ E(i) & i \notin \mathrm{Dom}(E') \end{cases}$$

Similar definitions apply to signature and functor environments.

Finally, we define "+" on program environments as follows:

$$(M, F, G, E) + (M', F', G', E') = (M \cup M', F + F', G + G', E + E')$$

In cases where some components of an operand are empty, we shall often omit them. For instance, $\mathcal{E} + M'$ means $\mathcal{E} + (M', \emptyset, \emptyset, \emptyset)$.

3.2 Realization Maps and Matching

We shall now define what it is for a structure to match a signature. The relation $\Sigma \geq S$ holds if there is a so–called *realization map* taking $\Sigma = (N)S'$ to S that affects only those substructures of S' whose name appears in N. In this way we obtain an interesting parallel with the polymorphic type discipline of ML [Mil78,DM82] (Table 2). Using the terminology of polymorphism, a structure matches a signature precisely if it is a generic instance of that signature, where the generic instantiation relation is defined in terms of realization maps.

Definition 10 (Realization Map) *A* realization map *is a total function* ϕ: Str $\rightarrow$ Str *such that for all* $S \in$ Str *and all* $strid \in$ StrId, *if* $S.strid$ *is defined, then* $\phi(S).strid$ *is defined, and* $\phi(S.strid) = (\phi(S)).strid$.

ModL	*Polymorphism*
structure	type
signature	type scheme
realization map	substitution
matching	generic instance

Table 2: ModL *vs.* Polymorphism

There are two important consequences of this definition. Firstly, a realization map may introduce more components (by enlarging the environment contained within a structure), but it may not destroy existing paths: if $S.path$ is defined, then $(\phi(S)).path$ is also defined. Secondly, a realization map may introduce more sharing (by identifying names within a structure), but it must preserve existing sharing: if $S.path = S.path'$, then $(\phi(S)).path = (\phi(S)).path'$, because both of the latter are equal to $\phi(S.path)$.

Example 11 *There is a realization map mapping S_1 to S_2 but none mapping S_2 to S_1, where*

The identity function on Str is a realization map, and if ϕ_1 and ϕ_2 are realization maps, then so is $\phi_2 \circ \phi_1$, where "$\circ$" is ordinary function composition.

The application of a realization map ϕ to an environment E is defined pointwise i.e., $(\phi(E))(strid) = \phi(E(strid))$. Thus the domain of $\phi(E)$ is the same as the domain of E.

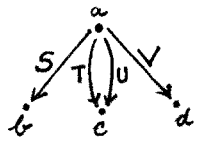
We say that ϕ *glides* on a structure $S = (n, E)$ if $\phi(S) = (n, \phi(E))$. Note that this does *not* imply that $\phi(S) = S$, but rather only that ϕ "descends smoothly" into S without changing its name or the domain of its environment. The *support* of a realization map ϕ, written $\mathrm{Supp}(\phi)$, is defined to be the set of structures on which ϕ does not glide.

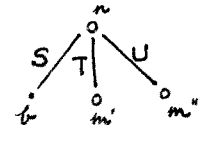
For every structure, S, we write $\mathrm{Names}(S)$ for the set of names occurring in S. Similarly for environments. For every signature, $\Sigma = (N)S$, the *bound names* are the names in N while the *free names*, $\mathrm{FN}(\Sigma)$, are those in $\mathrm{Names}(S) \setminus N$. Similarly, a structure S' in S is said to be *bound* in Σ if its name is a member of N, and is said to be free otherwise; $\mathrm{BS}(\Sigma)$ means the set of structures bound in Σ.

Definition 12 (Signature Matching) *A structure S* matches *a signature $\Sigma = (N)S'$, written $\Sigma \geq S$, if and only if there is a realization map ϕ such that $\phi(S') = S$ and* $\mathrm{Supp}(\phi) \subseteq \mathrm{BS}(\Sigma)$.

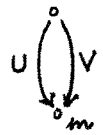
In this definition, the restriction of ϕ to S' and its substructures is uniquely determined by S' and S, so S can essentially only match Σ in one way.

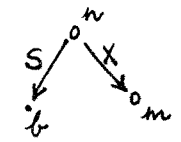
Example 13 *The structure S matches the signature Σ, where*

$S =$ 

$\Sigma =$ 

but S does not match either Σ' or Σ'':

$\Sigma' =$ 

$\Sigma'' =$ 

Definition 14 *A signature Σ is said to be* more general than *a signature Σ', written $\Sigma \geq \Sigma'$ if whenever a structure S matches Σ', it also matches Σ. Two signatures are equivalent, $\Sigma \equiv \Sigma'$ if each is more general than the other.*

One can prove that $(N)S \geq (N')S'$ if and only if $(N)S \geq S'$ and no $n' \in N'$ is the name of a free structure in $(N)S$. It can also be shown that two signatures are equivalent if and only if they can be obtained from each other by renaming of bound variables together with the introduction and elimination of spurious bound variables. Note the correspondence with the notion of generic instance defined by Damas and Milner [DM82]. The relation $\geq$ on signatures is a partial ordering on the equivalence classes.

3.3 Inference Rules

We shall now present the inference rules. The rules recursively define relations. For instance $\boxed{\mathcal{E} \vdash dec \Rightarrow E}$ in Figure 2 indicates that we define a ternary relation $(\_ \vdash \_ \Rightarrow \_) \subseteq \text{ProgEnv} \times \text{Dec} \times \text{Env}$. Let us say that a phrase is *legitimate* in a given program environment if it can be evaluated to a semantic object using the rules.

It is helpful to think of the rules as falling into two categories, "strict" and "liberal". The idea is that the strict rules, given a phrase and a program environment, leave very little choice as to what can follow after the arrow. By contrast, the liberal rules may be used to infer a host of different semantic objects for a given program environment and phrase.

The rules for declarations and structure expressions are strict and appear in Figure 2. Intuitively, the first component of a program environment records the names of structures that should be considered "fixed". Hence, in (2) names are accumulated in the program environment. In (3) n is required to be outside M so that it is not the name of a priviously fixed structure, and outside E_1 to avoid circularity in (n, E_1).

Functor applications (5) are evaluated in the usual way, by binding the (value of the) argument, S, to the formal parameter identifier, *strid*, and elaborating the body, *strexp*, provided that the argument structure S matches the parameter signature Σ. Notice that M is added to $\mathcal{E}'$ so that we are sure that the names that are chosen during the evaluation of the body are new with respect to the *actual* program environment. This is the all–important reason for having name sets in program environments.

Notice that the only freedom in these rules is the choice of n in Rule 3. By contrast, the rules for specifications and signature expressions are liberal (Figure 3). The key source of freedom is the instantiation rule (12) which says that whenever we can infer a signature for a signature expression, then we can also infer any less general signature — as defined in Section 3.2. This is crucial to get the simple rule for sharing, because the instantiation rule gives us the ability to introduce extra sharing before testing the equality in (7).

$\boxed{\mathcal{E} \vdash dec \Rightarrow E}$

$$\frac{\mathcal{E} \vdash strexp \Rightarrow S}{\mathcal{E} \vdash \mathtt{structure}\ strid = strexp \Rightarrow \{ strid \mapsto S \}} \quad (1)$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash dec_1 \Rightarrow E_1 \\ \cdots \\ \mathcal{E} + \mathrm{Names}(E_1) + \cdots + \mathrm{Names}(E_{k-1}) + E_1 + \cdots + E_{k-1} \vdash dec_k \Rightarrow E_k\end{array}}{\mathcal{E} \vdash dec_1\ ; \ldots ;\ dec_k \Rightarrow E_1 + \cdots + E_k} \quad (2)$$

$\boxed{\mathcal{E} \vdash strexp \Rightarrow S}$

$$\frac{M, F, G, E \vdash dec \Rightarrow E_1 \qquad n \notin M \cup \mathrm{Names}(E_1)}{M, F, G, E \vdash \mathtt{struct}\ dec\ \mathtt{end} \Rightarrow (n, E_1)} \quad (3)$$

$$\frac{E(path) = S}{M, F, G, E \vdash path \Rightarrow S} \quad (4)$$

$$\frac{\begin{array}{c}F(funid) = [strid, \Sigma, strexp', \mathcal{E}'] \\ M, F, G, E \vdash strexp \Rightarrow S \qquad \Sigma \geq S \\ \mathcal{E}' + M + \mathrm{Names}(S) + \{ strid \mapsto S \} \vdash strexp' \Rightarrow S'\end{array}}{M, F, G, E \vdash funid(strexp) \Rightarrow S'} \quad (5)$$

Figure 2: Rules for declarations and structure expressions

Rule (6) requires that the signature expression determine a signature with no bound names. The reason for this is that the binding of names in signatures is "outermost": no nesting of bound names is allowed within the environment part of a signature.

Rule 9 packages environments into signatures choosing an arbitrary name for it. The generalization rule (11) allows for binding of names in the signature, provided that they are not in M, i.e., not "fixed", and not free in the environment.

Turning to the rules for programs (Figure 3), let us consider functor binding (15). (The notion of principal signature will be explained in Section 4). First, the parameter signature is evaluated to $(N)S$, where N is chosen to be suitably new; this can always be achieved by applications of (12). Then the body of the functor is elaborated in the environment extended with *strid* bound to S, making sure that the names of S are considered fixed. If this succeeds, then we build a closure for the functor and bind it to *funid* in the functor environment. Notice that S' does not occur in the result. In particular, names used during the evaluation of the body may be reused; this is obtained by letting the name set in the result be empty.

4 Principal Signatures

Definition 15 *We say that a signature* Σ *is* principal *for sigexp in* $\mathcal{E}$ *if and only if (a)* $\mathcal{E} \vdash sigexp \Rightarrow \Sigma$, *and (b) whenever* $\mathcal{E} \vdash sigexp \Rightarrow \Sigma'$ *then* $\Sigma \geq \Sigma'$.

Intuitively, a signature for a signature expression is principal if it has the components and the sharing specified by the signature expression, *and no more.*

Returning to (15), let us see why we require the inferred signature to be principal. Suppose a functor declaration has been found legitimate. Then we would expect *any* application of the functor to a structure which matches the formal parameter signature *expression* to be legitimate. However, this requires that the inferred formal parameter signature be principal. So among all the similarities with polymorphic type checking, here is an important difference; in the latter,

$\boxed{\mathcal{E} \vdash spec \Rightarrow E}$

$$\frac{\mathcal{E} \vdash sigexp \Rightarrow (\emptyset)S}{\mathcal{E} \vdash \texttt{structure}\ strid : sigexp \Rightarrow \{ strid \mapsto S \}} \tag{6}$$

$$\frac{M, F, G, E \vdash spec \Rightarrow E' \qquad (E + E')(path_1) = (E + E')(path_2)}{M, F, G, E \vdash spec\ \texttt{sharing}\ path_1 = path_2 \Rightarrow E'} \tag{7}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash spec_1 \Rightarrow E_1 \\ \cdots \\ \mathcal{E} + E_1 + \cdots + E_{k-1} \vdash spec_k \Rightarrow E_k\end{array}}{\mathcal{E} \vdash spec_1 \,;\ldots;\, spec_k \Rightarrow E_1 + \cdots + E_k} \tag{8}$$

$\boxed{\mathcal{E} \vdash sigexp \Rightarrow \Sigma}$

$$\frac{\mathcal{E} \vdash spec \Rightarrow E_1}{\mathcal{E} \vdash \texttt{sig}\ spec\ \texttt{end} \Rightarrow (\emptyset)(n, E_1)} \tag{9}$$

$$\frac{G(sigid) = \Sigma}{M, F, G, E \vdash sigid \Rightarrow \Sigma} \tag{10}$$

$$\frac{M, F, G, E \vdash sigexp \Rightarrow (N)S \qquad n \notin M}{M, F, G, E \vdash sigexp \Rightarrow (N \cup \{n\})S} \tag{11}$$

$$\frac{\mathcal{E} \vdash sigexp \Rightarrow \Sigma \qquad \Sigma \geq \Sigma'}{\mathcal{E} \vdash sigexp \Rightarrow \Sigma'} \tag{12}$$

$\boxed{\mathcal{E} \vdash prog \Rightarrow \mathcal{E}'}$

$$\frac{\mathcal{E} \vdash dec \Rightarrow E}{\mathcal{E} \vdash dec \Rightarrow (\mathrm{Names}(E), \emptyset, \emptyset, E)} \tag{13}$$

$$\frac{\mathcal{E} \vdash sigexp \Rightarrow \Sigma \qquad \Sigma \text{ principal for } sigexp \text{ in } \mathcal{E}}{\mathcal{E} \vdash \texttt{signature}\ sigid = sigexp \Rightarrow (\emptyset, \emptyset, \{ sigid \mapsto \Sigma \}, \emptyset)} \tag{14}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash sigexp \Rightarrow (N)S \qquad \mathcal{E} = (M, F, G, E) \qquad N \cap (M \cup \mathrm{FN}(E)) = \emptyset \\ (N)S \text{ principal for } sigexp \text{ in } \mathcal{E} \\ \mathcal{E} + \mathrm{Names}(S) + \{ strid \mapsto S \} \vdash strexp \Rightarrow S'\end{array}}{\begin{array}{l}\mathcal{E} \vdash \texttt{functor}\ funid(strid{:}sigexp){=}strexp' \Rightarrow \\ (\emptyset, \{ funid \mapsto [strid, (N)S, strexp', \mathcal{E}] \}, \emptyset, \emptyset)\end{array}} \tag{15}$$

$$\frac{\mathcal{E} \vdash prog_1 \Rightarrow \mathcal{E}_1 \qquad \mathcal{E} + \mathcal{E}_1 \vdash prog_2 \Rightarrow \mathcal{E}_2}{\mathcal{E} \vdash prog_1\ ;\ prog_2 \Rightarrow \mathcal{E}_1 + \mathcal{E}_2} \tag{16}$$

Figure 3: Rules for specifications, signatures, and programs

formal function parameters have types, not type schemes, while in ModL we have signatures as the "types" of formal parameters.

A similar constraint is appropriate on the rule for signature declaration (14). For example it will ensure that in

```
signature SIG =
sig structure A: sig end;
    structure B: sig end;
end;

functor F(S:SIG) = body
```

`S.A` and `S.B` will be treated as being different within the functor body.

But *do* signature expressions have principal signatures? The answer is yes, but under certain conditions. We shall not go into details but the most interesting condition is that structure names uniquely determine structures: if $S_1 = (n, E_1)$ and $S_2 = (n, E_2)$ then $E_1 = E_2$. This constraint creates a one–one correspondence between the DAGs and the elements of Str.

The existence of principal signatures is proved constructively by giving an algorithm which either fails or succeeds and succeeds with a principal signature just in case the signature expression is legitimate. The algorithm uses a unification algorithm which generalizes ordinary term unification to structure unification. These algorithms and detailed proofs of their correctness exist and will appear in the last author's forthcoming Ph. D. thesis.

References

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on the Principles of Programming Languages*, pages 207–212, 1982.

[Har86] Robert Harper. *Introduction to Standard ML.* Technical Report, University of Edinburgh, September 1986.

[Mac86a] David MacQueen. Modules for Standard ML. In *Standard ML*, by Robert Harper and David MacQueen and Robin Milner, Technical Report ECS–LFCS–86–2; Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[Mac86b] David MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on the Principles of Programming Languages*, 1986.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Plo81] Gordon Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI–FN–19, Computer Science Department, Aarhus University, Denmark, 1981.