Principal Signatures for Higher-order Program Modules

Mads Tofte
Department of Computer Science (DIKU)
Copenhagen University

Abstract

Under the Damas-Milner type discipline for functional languages, every expression has a principal type, if it elaborates at all. In the type discipline for ML Modules, a signature expression has a principal signature, if it elaborates at all. However, while functions can be higher-order in ML, parameterised modules in ML are first-order only. We present a type discipline for a skeletal higher-order module language which has principal signatures. Sharing and multiple wiews of structures are handled in a manner which is compatible with the semantics of the first-order ML modules.

1 Introduction

Several programming languages have a notion of program module. Typically, a module is either *basic* or *parameterised*. A basic module is called a *package* in ADA and a *structure* in Standard ML[5]; a parameterised module is called a *generic package* in ADA and a *functor* in ML. In these two languages, a parameterised module can at most take a basic module as parameter and yield a basic module as result, i.e. parameterised modules are *first-order*. Our proposal is to admit higher-order modules, i.e. parameterised modules whose arguments and results might themselves be parameterised.

To see the need for such a concept, consider the modules in Figure 1. Standard ML syntax is used. First a signature (i.e. a "structure type" or "interface") called *MONOID* is declared. Then we define the functor *Prod*, which maps monoids *N* and *M* to their product monoid, and use *Prod* in another functor *Square* which we finally apply to get a structure *Plane*.

Because ML is statically scoped, *Square* has no meaning unless *Prod* has already been declared. Consider Figure 2, however, which contains the declaration of a functor signature *PROD* and a higher-order functor *Square*. Now *Square* makes sense without reference to any particular functor; *Square* is thus easier to read and in addition it can be compiled separately regardless of whether *Prod* has been written or not.

In short, we propose to admit functors in structures and signatures. Functors are still mappings from structures to structures, but since structures can contain functors, functors now are higher-order.

```
signature MONOID = sig
  type t
  val e: t
  val plus: t * t -> t
end;

functor Prod(
  structure M: MONOID
  structure N: MONOID): MONOID=
struct
  type t = M.t * N.t
  val e = (M.e, N.e)
  fun plus((x1,x2),(y1,y2))=
    (M.plus(x1,y1),N.plus(x2,y2))
end;

functor Square(X: MONOID):MONOID=
  Prod(structure M = X
       structure N = X);

structure Plane = Square(
  struct
    type t = real
    val e = 0.0
    fun plus(i,j):real = i+j
  end);

Plane.plus(Plane.e,(7.4,5.4));
```

Figure 1: First-order functors

The main theoretical challenge posed by higher-order modules is understanding their static semantics.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 089791-453-8/92/0001/0189 $1.50

```
signature PROD =
 (structure M: MONOID
  structure N: MONOID): MONOID

functor Square(
 structure X: MONOID
 functor Prod: PROD):MONOID=
   Prod(structure M = X
        structure N = X);
```

Figure 2: A Higher-order Functor

Harper *et al.*[3] propose a very elegant type-theoretic module concept which allows higher-order modules. Unfortunately, their approach does not address sharing and multiple wiews of structures, both of which are important in ML. In the present paper we do treat sharing and multiple views of structures, and the treatment is compatible with ML.

Just as the static semantics of the Core of ML (i.e. the non-modules part) depends critically on the existence of principal types, so the static semantics of ML modules relies critically on the existence of principal signatures, as explained below. The aim of this paper is to address what we think is the most interesting theoretical problem posed by adding higher-order functors to ML: can one extend the modules type discipline to the higher-order language while maintaining that principal signatures exist? The contribution of this paper is that we can answer this question in the affirmative for a skeletal language.

The semantic theory presented in this paper is intended to be the simplest possible generalisation of the first-order theory. In one respect, the theory in its present form is restrictive: it requires that whenever a functor f with functor signature Φ_1 is matched against a functor signature Φ_2, we have $\Phi_1 = \Phi_2$. The relaxation of this restriction will be discussed below.

Although the significance of principal type schemes in the Core is widely recognised, the fact that principal signatures are as crucial to the Modules as the principal type schemes are to the Core is less well known. We shall therefore briefly explain what principal signatures are in Section 2. We then proceed to present the static semantics, leading to the statement of the principality theorem in Section 3. In Section 4 we explain *admissification*, the Modules equivalent of unification, and in Section 5 we present the signature inference algorithm. Finally, in Section 6 we explain why the restriction of our type discipline to the case of first-order modules is not completely identical to the ML discipline.

2 Elaboration of Modules

The word *elaboration* means "the static part of execution"; the term *type inference* is often used for the same thing, but the latter term is not good for Modules, since other semantic objects than types are inferred. In this section we review as much of the elaboration of Modules as is required to explain informally what principal signatures are.

ML is a statically typed language, so the definition of equality of types is important. In languages with a fixed signature of types, the issue of when two types are equal might be trivial, but ML allows user-defined types, type abbreviations and type specifications, so equality of types is not a trivial issue.

Equality of types cannot simply be identity of type constructors (i.e. program identifiers) for the `type` construction allows one to have multiple identifiers for the same type. Therefore, ML has a distinction between *type constructors*, which are program identifiers, and *type names*, which are "unique names". The latter are internal to the semantics; testing of equality of types tests equality of type names, not equality of type constructors. Throughout this paper, *name* means "unique name," not identifier.

The basic rule for introducing new type names is simple: `datatype` declarations introduce new type names, `type` declarations do not.

Now let us consider type *specifications*, for example the specification of the two types $M.t$ and $N.t$ in functor *Prod* in Figure 1. We wish to elaborate the body of the functor under some assumptions about M and N (the actual structures may not yet be available and in any case they could vary). Should we assume that $M.t$ and $N.t$ are equal or not? A moment's reflection reveals that they should not be treated as identical, for one can easily produce actual structures corresponding to M and N which have different t types. In general, we should only deem the body of a functor type correct if it would also be type correct were we to use any real structures that match the parameter specifications instead of the formal structures. In particular, M and N should only have as much equality of types as any real structures that match the parameter specifications must have. The way to force equality between specified types is by the `sharing` specification. Figure 3 shows a functor which would not be type correct without a sharing constraint.

A functor with multiple structure arguments is just a derived form of a functor with a single structure argument which has substructures. (Figure 4 shows the signature of the argument of the functor from Figure 3.) It therefore suffices to consider functors of the simple form

```
functor F(X:sigexp) = body
```

```
functor F(
  structure M: MONOID
  structure N: MONOID
  sharing type M.t = N.t)=
struct
  val x = M.plus(M.e, N.e)
end;
```

Figure 3: Example of type sharing

```
SIGNATURE ARG= sig
  structure M: MONOID
  structure N: MONOID
  sharing type M.t = N.t
end;
```

Figure 4: Signature with structure specifications

where *sigexp* is a signature expression. If we can elaborate *sigexp* to a formal structure S which has precisely the components and sharing that any structure which matches *sigexp* must have then we can elaborate *body* assuming that X is bound to S without making assumptions that are not certain to hold when the functor is applied. If in addition S is uniquely determined by *sigexp* then the elaboration of the body can be done without worrying about which S to choose for X in order to get the body to elaborate.

ML is designed in such a way that whenever *sigexp* elaborates at all then it elaborates to a structure S which has precisely the components and the sharing that any structure which satisfies *sigexp* must have. Moreover, S is unique, up to a certain kind of renaming, as explained below. This crucial property of signature elaboration is expressed formally in the *principality theorem*[4, App A] which states, roughly speaking, that whenever *sigexp* can be elaborated at all, then it can be elaborated to a unique principal signature.

3 A Skeletal Language

We shall now define elaboration of higher-order functors using structural operational semantics. Most of the notation and terminology is exactly as in [5] and [4] but is repeated here for expository reasons.

The grammar of our skeletal language appears in Figure 5. We are primarily interested signature expressions (*sigexp*). These can contain specifications (*spec*). Compared with Standard ML, the one novelty

sigexp ::=	
sig *spec* **end**	generative
sigid	signature identifier
funsigexp ::=	
$(strid : sigexp_1) : sigexp_2$	functor signature
spec ::=	
structure *strid*: *sigexp*	structure
functor *funid*: *funsigexp*	functor
sharing $longstrid_1 = longstrid_2$	sharing
	empty
$spec_1 \langle ; \rangle spec_2$	sequential

Figure 5: Grammar

in Figure 5 is that we allow specification of functors in our language. Functor specifications contain *functor signature expressions*, (*funsigexp*). For simplicity, all specifications but structure and functor specifications have been omitted. Also notice that our skeletal language does not include structure expressions and declarations; in particular, there are no productions for functor declaration and functor application. Apart from the fact that we would allow the declaration of functors inside structures, there would be little of interest in spelling out the details here, especially since our main aim is the principality theorem, which concerns signature expressions only.

As in ML, we assume three disjoint classes of identifiers, namely SigId (signature identifiers), FunId (functor identifiers) and StrId (structure identifiers), ranged over by *sigid*, *funid* and *strid*, respectively. Substructures are referenced by *long identifiers*, for example `A.B` is the `B` substructure of the structure `A`. The $\langle ; \rangle$ in the last production means that the semicolon is optional.

In Standard ML, one can specify sharing between structures and this implicitly implies sharing of all type and structure components that are specified in both structures. Just as types have type names, so structures have *structure names* and we say that two structures *share* if they have the same name. The elaboration of a **struct**···**end** expression produces a structure with a fresh name. Structure names are only used for representing sharing and they play no part in the dynamic semantics.

3.1 Semantic Objects

We shall now define the semantic objects, i.e. the objects that occur in the inference rules which will be presented below. The semantic objects are defined in Figure 6.

$$
\begin{aligned}
m &\in \text{StrName}\\
N &\in \text{NameSet} = \text{Fin(StrName)}\\
G &\in \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Sig}\\
F &\in \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunSig}\\
SE &\in \text{StrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Str}\\
S \text{ or } (m, E) &\in \text{Str} = \text{StrName} \times \text{Env}\\
E \text{ or } (F, SE) &\in \text{Env} = \text{FunEnv} \times \text{StrEnv}\\
\Sigma \text{ or } (N)S &\in \text{Sig} = \text{NameSet} \times \text{Str}\\
\Phi \text{ or } &\\
(N)(S,(N')S') &\in \text{FunSig} = \text{NameSet} \times (\text{Str} \times \text{Sig})\\
A &\in \text{Asmb} = \text{Str} \cup \text{Asmb} \times \text{Asmb}\\
B \text{ or } N, G, E &\in \text{Basis} = \text{NameSet} \times \text{SigEnv} \times \text{Env}
\end{aligned}
$$

Figure 6: Semantic Objects

(m2, (	structure name
{},	functor environment
{},	structure environment
$\{t \mapsto$ real * real}	type environment
$\{e \mapsto$ real * real,	variable environment
$plus \mapsto \sigma$}))	

Figure 7: The structure *Plane*

A *structure environment* SE is a finite map from structure identifiers to structures; we write Dom SE for the domain of SE. Similarly for *signature environments* G and *functor environments* F. A *(static) structure* S is a pair (m, E), where m is the name of the structure and E is an environment, which gives the static information about the components of the structure. To term such an object *static structure* (as opposed to signature) might seem a bit odd, since one informally thinks of a signature as being a "structure type," but as we shall see shortly, signatures and static structures are different. Since the only structure components we allow in this paper are structures and functors, an environment E is a pair of just two environments (F, SE). Figure 7 shows a structure which *Plane* from Figure 1 could elaborate to, if we assume that environments also contain type and variable environments. In the figure, σ is (real * real) * (real * real) $\rightarrow$ real * real.

We shall often need to select parts of semantic objects — for example the name of a structure. In such cases we rely on variable names to indicate which part is selected. For instance "m of S" means "the structure name of S."

Moreover, when a semantic object contains a finite map we shall "apply" the object to an argument, relying on the syntactic class of the argument to determine the relevant function. For instance $S(strid)$ means $(SE \text{ of } (E \text{ of } S))(strid)$.

Names that are specified in a signature expression and are not shared with already declared types or structures stand for unknown types and structures. In other words, such names are *generic* or *flexible*, as opposed to names of declared types and structures, which we call *rigid*. A *signature* is an object of the form $(N)S$, where S is a structure and N is a finite set of names. The prefix (N) is a binding operator which binds the members of N throughout S. The names in N are the flexible names of the signature whereas names that are free in $(N)S$ are rigid. Semantic objects that can be obtained from each other by renaming of bound names are considered equal. For all semantic objects A, we write names(A) to mean the set of names that occur free in A.

Notice the resemblance between a signature $\Sigma = (N)S$ and a type scheme $\sigma = \forall\alpha^{(k)}.\tau$ in the Damas-Milner type discipline. Instantiation of bound names in Σ takes place when a structure is matched against Σ.

A *functor signature* Φ is an object of the form $(N)(S, (N')S')$. Here $(N)S$ is the (principal) signature for the parameter signature expression of the functor and S' is the body structure of the functor; the names bound by (N') are the names in S' which have to be generated afresh upon each functor application, typically because of **struct**···**end** expressions in the body. As an example, the functor signature of the functor *Square* in Figure 1 is $(N)(S, (N')S')$, where

$$
N = \{\texttt{m1}, \texttt{t1}\} \quad S = (\texttt{m1}, \{t \mapsto \texttt{t1}\})
$$
$$
N' = \{\texttt{m2}\} \quad S' = (\texttt{m2}, \{t \mapsto \texttt{t1} * \texttt{t1}\})
$$

(Here we have temporarily admitted types into the language, but ignored variable environments and empty environments.) Notice that the meaning of the t type of the result depends on the t type specified in the argument via the name t1 which is bound outermost in the functor signature.

A *basis* B is a triple N, G, E, where N is the set of rigid names generated so far, G is a signature environment and E is an environment. The G and E components are used for looking up identifiers during elaboration.

An *assembly* A records the structures that have been generated so far. Assemblies and bases appear together in pairs (A, B). Not all structures in A need occur in B, for B contains only those semantic objects which are currently in scope. On the other hand, every structure S which occurs somewhere in B will also occur somewhere in A, or at least there will be some structure occurring somewhere in A of which S is a cut-down version, as defined below. Throughout this paper, occurrence is hereditary; for example, S occurs in $(N)(S, (N')S')$ and E occurs in (m, E).

As one sees from the definition in Figure 6, an assembly can be thought of as a binary tree of struc-

```
signature PLUS = sig
  type t
  val plus: t * t -> t
end;

structure Plane1 : PLUS= Plane;
```

Figure 8: A signature constraint

tures; this is not strictly necessary, although practical in proofs. The assembly is just a store of structures, and other representations would work well too.

3.2 Consistency

Once a structure has been created, it is possible to have restricted views of it. This can happen in two ways. The first is by a *signature constraint*. (Figure 8 shows a signature constraint which has the effect of creating a view of the *Plane* structure which hides the e component.) The other way is to apply a functor with signature $(N)(S,(N')S')$ to an actual structure S^+ which has more components than required by S. In this case a cut-down view of S^+ is created.

Both of these ways may hide components, but they never hide sharing. Thus one can easily have two structures $S_1 = (m_1, E_1)$ and $S_2 = (m_2, E_2)$ where $m_1 = m_2$ (i.e. the two structures share) and $E_1 \neq E_2$. In this case, however, names in E_1 and E_2 should at least be used consistently in those components that are present in both views. This leads to the definition of *consistency*:

A semantic object A or assembly A of objects is said to be *consistent* if (after changing bound names to make all nameset prefixes in A disjoint) for all S_1 and S_2 occurring in A and for every *strid* and every *funid*, if m of $S_1 = m$ of S_2, then

1. If $S_1(strid)$ and $S_2(strid)$ exist, then m of $S_1(strid) = m$ of $S_2(strid)$
2. If $S_1(funid)$ and $S_2(funid)$ exist, then $S_1(funid) = S_2(funid)$

Note that consistency of functor components is equality; for structures, only name equality is required. It is essentially item 2 which has the undesirable effect of demanding complete agreement between specified and declared functors. Perhaps it is possible to drop this item altogether without upsetting the rest of the theory, but the consequences of this change are not yet fully understood.

3.3 Cover

As hinted in Section 3.1 once a structure $S = (m, E)$ has been created, S must have at least the components of any subsequent view S^- of S. In particular, S^- should be (m, E^-), where E^- is an environment satisfying $\mathrm{Dom}\, E^- \subseteq \mathrm{Dom}\, E$. Of course we would also expect some relationship between the objects in the range of E^- and the objects in the range of E, but that is part of consistency (see Section 3.2). The definition of cover given below is a slight generalisation of the "has at least the components of" relation. Moreover, during elaboration one sometimes wants to extend the current assembly with new structures without adding components to structures in the current assembly. This leads to the notion of conservative cover, a strengthening of ordinary cover:

Let A_1 and A_2 be semantic objects or assemblies of objects, let N be a name set and let *id* range over structure and functor identifiers. We say that A_2 *covers* A_1 *on* N if whenever (m, E_1) occurs in A_1 with m free and $m \in N$, then $m \in \mathrm{names}(A_2)$ and for all $id \in \mathrm{Dom}\, E_1$ there exists an E_2 such that (m, E_2) occurs in A_2 with m free and $id \in \mathrm{Dom}\, E_2$. We say that A_2 *covers* A_1 if A_2 covers A_1 on names(A_1). We say that A_2 is a *conservative cover* of A_1 if A_2 covers A_1 and A_1 covers A_2 on names(A_1).

3.4 Admissibility

Besides demanding that semantic objects be consistent, the definition of ML requires *cycle-freedom* and *well-formedness*. The former prevents a structure from being specified as a proper substructure of itself and the latter ensures that for every name prefix (N) and every structure $S = (m, E)$ in the scope of (N), if m is not bound by (N) then no name in E is bound by (N) either. We refer to [5, Sec 5.3] and [5, Sec 5.4] for details. We now collect these constraints as follows:

An object or assembly A is *admissible* if it is cycle-free, consistent and well-formed. We say that A_2 is an *admissible cover* of A_1, written $A_1 \sqsubseteq A_2$, if (A_1, A_2) is admissible and A_2 covers A_1. We say that A_2 is a *conservative admissible cover* of A_1, written $A_1 \trianglelefteq A_2$, if (A_1, A_2) is admissible and A_2 is a conservative cover of A_1. Both $\sqsubseteq$ and $\trianglelefteq$ are preorders.

3.5 Realisation

A *realisation* is a function φ : StrName $\rightarrow$ StrName. The *support* $\mathrm{Supp}\,\varphi$ of a realisation φ is the set of names n for which $\varphi(n) \neq n$.

Realisations φ are extended to apply to all semantic objects; their effect is to replace each name n by $\varphi(n)$. In applying φ to an object with bound names, such as a signature $(N)S$, first bound names must be changed to avoid name capture.

3.6 Signature Instantiation

When matching a structure S' against a signature $(N)S$, S' may have more components than present in S. However, for the components that do appear in S, it must be possible to obtain consistent naming in S and S' by instantiation of the members of N:

A structure S_2 *is an instance of* a signature $(N_1)S_1$, written $(N_1)S_1 \geq S_2$, if there exists a realisation φ such that $\varphi(S_1) = S_2$ and $\text{Supp}\,\varphi \subseteq N_1$. Note that there is at most one such φ.

3.7 Principal Signatures

Let B be a basis, A an assembly and let *sigexp* be a signature expression. In Section 3.8 we shall define what it is for *sigexp* to elaborate to a structure S in (A, B), written $A, B \vdash sigexp \Rightarrow S$. This is used in the definition of principal signature:

We say that a signature $(N)S$ is *principal* for *sigexp* in (A, B) if $A \sqsupseteq B$ and, choosing N such that $N \cap \text{names}(A, B) = \emptyset$

1. There exists an A' such that $A \trianglelefteq A'$ and $A', B \vdash sigexp \Rightarrow S$
2. For all A' and S', if $A \sqsubseteq A'$ and $A', B \vdash sigexp \Rightarrow S'$ then $(N)S \geq S'$

The reason for introducing A' in the above definition will be given in Section 3.8. For now, simply note that a signature $\Sigma = (N)S$ is principal for *sigexp* in (A, B) only if all structures that *sigexp* elaborates to can be obtained from Σ by instantiation of the flexible names, i.e. the names in N. This is similar to the notion of principal type scheme in the Damas-Milner type discipline [2], where the flexible names play the rôle of generic type variables.

3.8 Inference Rules

The inference rules for signature expressions, specifications and functor signature expressions appear below. All the conclusions of the rules are of the form

$$A, B \vdash phrase \Rightarrow P \tag{1}$$

which is read: in the basis B and assembly A, the phrase *phrase* elaborates to the object P. Here *phrase* is one of the phrase forms defined in Figure 5 and P is either a structure, a signature, a functor signature or an environment.

At top level, A is the assembly of all structures declared so far. The phrase classes we consider do not declare new structures (recall that we are concerned with signature expressions only), so elaboration does not produce an assembly. Thus we have A on the left-hand side of the $\vdash$, but not on the right-hand side of the $\Rightarrow$. If we were to write down rules for structure expressions, functor declaration and functor application, elaboration would gradually increase the top-level assembly, so we would have assemblies on the right-hand side of the $\Rightarrow$ in those rules.

An important invariant to keep in mind when reading the rules is that one can infer (1) only if A is admissible and covers both B and P. Applying this invariant to the case where *phrase* is a top-level signature expression and P is a signature Σ, we see that Σ cannot add components to top-level structures.

The rules appear in the Appendix. There is one rule for each production in the grammar, plus a rule (4) for principal signatures. We shall now discuss the most important rules.

The assembly is only expanded in two rules. The first is rule 4. Let us first read the rule bottom-up. Assume the binding operation is non-trivial (i.e. that $N \neq \emptyset$). Then there are structures that occur free in S but are not free in $(N)S$. Therefore we need to expand the assembly to cover the elaboration of *sigexp* to S. Conversely, reading the rule top-down, it is natural that when we bind structures that occur in S by (N) then we simultaneously discharge the corresponding structures from the assembly. That way renaming of bound names and expansion of the assembly go together. The expansion of A to A' is not supposed to add components to structures that are already in A, for the expansion is solely there to introduce new structures that are used locally in the elaboration of *sigexp*, so we require $A \trianglelefteq A'$.

The second rule which expands the assembly is rule 5. We first elaborate $sigexp_1$ to a signature $(N)S$. This must happen via rule 4, so $(N)S$ is principal for *sigexp* in (A, B). We then choose N in a way that avoids accidental sharing ($\text{names}(A, B) \cap N = \emptyset$). We now form the assembly $A' = (A, S)$ which is used for the elaboration of $sigexp_2$ to Σ, the functor result signature. By the invariant, every structure (m, E) that occurs free in Σ and satisfies $m \in N$ must be covered by A', i.e. by S.

The $+$ operation on environments is defined as follows: $E + E'$ is the environment defined by $(E + E')(id) = E'(id)$, if $id \in \text{Dom}\,E'$, and $(E + E')(id) = E(id)$ otherwise; The $+$ operation is extended to semantic objects containing environments in the obvious way. The "$+ N$" in rule 5 is explained in Section 4. We can now express the invariant as a formal constraint on elaboration:

A sentence of the form $A, B \vdash phrase \Rightarrow P$ is *admissible* if (A, B, P) is admissible and A covers (B, P). A tree ∇ is called an *inference tree* if it is formed by applying the inference rules and satisfies the side-conditions on the rules; ∇ is said to be a *proof* if every sentence occurring in it is admissible; inference trees that are not proofs are banned from elaboration.

```
structure P =
struct
 structure Q = struct end
end;

signature SIG = sig(3)
 structure P' : sig end
 functor F: (X: sig(4)
  structure P'': sig
    structure Q: sig end
  end
  sharing P'' = P'
  end(4)): sig end
 sharing P' = P
end(3);
```

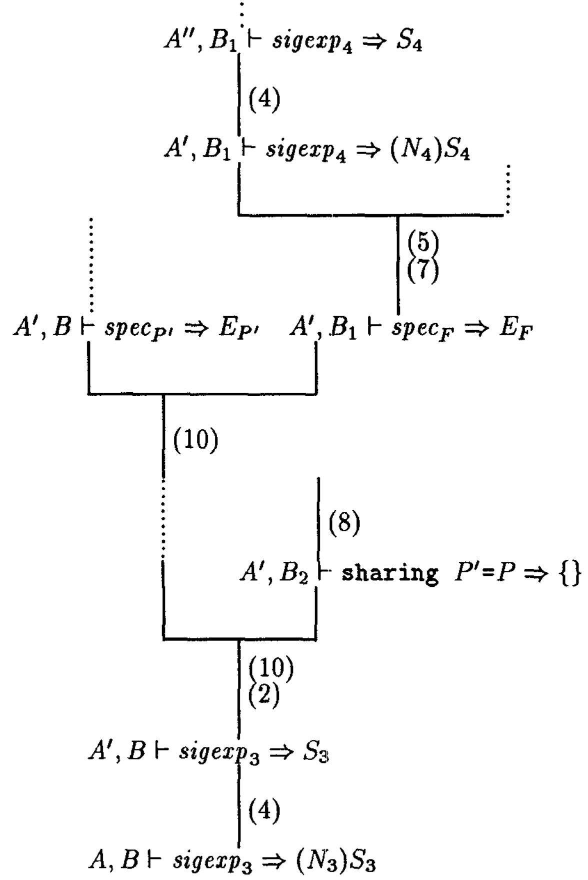
Figure 9: Embedded functor signature

We can now state the principality theorem.

Theorem 3.1 (Principal Signatures) *Let B be a basis, A an assembly and let $A, B \vdash sigexp \Rightarrow S$, for some S. Then there exists a principal signature for $sigexp$ in (A, B).*

This theorem follows from a stronger theorem, which we state in Section 5.

The elaboration of the program in Figure 9 is summarised in Figure 10. It illustrates most features of the inference system. Some of the occurrences of **sig** and **end** have been decorated with occurrence numbers. Let $sigexp_i$ be the signature expression from $\texttt{sig}^{(i)}$ to $\texttt{end}^{(i)}$, $i = 3, 4$. The specifications of P' and F are referred to as $spec_{P'}$ and $spec_F$, respectively. We assume that the assembly and the basis are empty at the outset. After the elaboration of the structure declaration we have $B = \{P \mapsto S_P\}$ and $A = S_P$, where $S_P = (\texttt{m1}, \{Q \mapsto (\texttt{m2}, \{\})\})$. The elaboration then proceeds as outlined in Figure 10.



$S_4 = (\texttt{m4}, \{P'' \mapsto S_P\})$

$N_4 = \{\texttt{m4}\}$

$E_{P'} = \{P' \mapsto (\texttt{m1}, \{\})\}$

$B_1 = B + E_{P'}$

$\Phi = (N_4)(S_4, (\{\texttt{m5}\})(\texttt{m5}, \{\}))$

$E_F = \{F \mapsto \Phi\}$

$B_2 = B_1 + E_F$

$S_3 = (\texttt{m3}, E_{P'} + E_F)$

$N_3 = \{\texttt{m3}\}$

$A = S_P$, $A' = (A, S_3)$ and $A'' = (A', S_4)$

$B = \{P \mapsto S_P\}$

$S_P = (\texttt{m1}, \{Q \mapsto (\texttt{m2}, \{\})\})$

Figure 10: An elaboration tree

4 Admissification

Principal signatures can be found by an algorithm which we will present in Section 5. The algorithm crucially relies on the existence of *most general* (or *principal*) *admissifiers*, just as the algorithm W[2] relies on the existence of most general unifiers in first-order term unification[7]. Admissification is invoked during elaboration when a sharing specification is encountered. If the two structures that are specified to share do not have the same name, admissification seeks to identify the names. Because the resulting assembly has to be admissible, this might lead to further identification of names of structures that occur below the two structures.

For example, consider the program in Figure 9. The algorithm traverses the program from left to right, corresponding to a depth-first, left-to-right traversal of the elaboration tree. The algorithm makes as few identifications of names as possible, in order that the result be principal. The first sharing constraint has the effect of identifying the names of P' and P''. The second sharing constraint identifies this name with `m1`, the name of P, and this forces identification of the name of $P''.Q$ with the name of $P.Q$, i.e. `m2`, even though P'' is not mentioned in the sharing equation. The reason for this is that we have to stay within a consistent assembly.

Not any pair of names can be identified. Names of declared structures (as opposed to specified structures) are always supposed to be unique, so two such names cannot be identified. Also, if $(N)S$ is the result of elaborating a parameter signature expression of a functor specification, then the names of N are to be treated as distinct within the result signature.

Operationally, what it means for a name to be rigid is that it cannot be replaced by another name by admissification. Formally, the set of rigid names is the names in the N-component of the basis. This is why we add (i.e. union) N with the N-component of the basis in rule 5.

Finally, the side-condition $N \cap \mathrm{names}(A, B) = \emptyset$ in rule 5 constrains possible identifications of names. The way we check whether this side-condition is being violated is by assuming that all names have a *rank*, which is a natural number. At top level, names have rank 0. Rank is increased by one locally when one starts elaboration of the result signature of a functor signature expression. The details are as follows:

Let A be an assembly, φ a realisation and N a name set. The *rank of* A, written $\mathrm{rank}(A)$, is the maximum rank of any name that occurs free in A. A is *stratified* if whenever (m, E) occurs in A with m free then $\mathrm{rank}(E) \leq \mathrm{rank}\, m$. (Intuitively, (m, E) contains free names that will later be bound further out by name prefixes nested within each other; by insisting that structures be stratified we can keep these name prefixes disjoint and avoid capture of free names throughout elaboration.) We say that φ is *decreasing* if $\mathrm{rank}(\varphi n) \leq \mathrm{rank}(n)$, for all names n; φ is said to be *fixed on* N if $\varphi(n) = n$, for all $n \in N$. (Intuitively, realisation is supposed to be fixed on rigid names, i.e. the names in N of B; on other names, realisation should not increase rank — the binding of a name may be moved further out, but never further in, as this could lead to capture of free names when forming signatures.) Moreover, φ is said to be an *admissifier for* A *under* N if φ is decreasing and fixed on N, A covers φA on N and φA is admissible and stratified. An admissifier φ^* for A under N is said to be *most general* or *principal* if, for every admissifier φ for A under N, there exists a realisation φ' such that φ' is decreasing, fixed on N and $\varphi'(\varphi^* A) = \varphi A$. The following theorem corresponds to Theorem 10.3 in [4] and the proof is similar.

Theorem 4.1 (Admissification) *For any assembly A and name set N, if there exists some admissifier for A under N then there exists a principal admissifier for A under N.*

The proof of this theorem is similar to the proof of Theorem 10.3 in [4]; Essentially, the proof gives an algorithm *Admissify*, such that for all assemblies A and name sets N, either $\mathit{Admissify}(A, N)$ returns a principal admissifier φ^* for A under N, or it fails, in case no admissifier exists.

Admissification is reminiscent of Remy's record unification[6] and Aït-Kaci's type unification[1]. The main extra subtlety we have to deal with is unification under nested name prefixes.

5 An Inference Algorithm

In this section we state a stronger version of Theorem 3.1 which is suitable for inductive proof. Also, we present the constructive part of the proof as an inference algorithm.

Let K be the category defined as follows. An object O of K is a pair (A, B), where A is an assembly and $B = (N, G, E)$ is a basis satisfying that A is admissible and stratified and $A \sqsupseteq B$ (by which we mean $A \sqsupseteq (G, E)$ and $\mathrm{names}(A) \supseteq N$). The set of objects of K is denoted Obj. For all objects $O_1 = (A_1, B_1) = (A_1, (N_1, G_1, E_1))$ and $O_2 = (A_2, B_2)$, and for every φ and rank k, there is a morphism $O_1 \xrightarrow{\varphi,k} O_2$ if φ is decreasing and fixed on N_1, $\varphi B_1 = B_2$, $\varphi A_1 \sqsubseteq A_2$, A_1 covers A_2 on N_1 and $\mathrm{rank}(O_1, O_2) \leq k$. The condition "$A_1$ covers A_2 on N_1" prevents realisation from adding components to rigid structures. Notice that $O_1 \xrightarrow{\varphi,k} O_2$ implies $\varphi(B_1) = \varphi(N_1, G_1, E_1) = (N_1, \varphi G_1, \varphi E_1) = B_2$.

Morphisms $O_1 \xrightarrow{\varphi,k} O_2$ and $O_1 \xrightarrow{\varphi',k'} O_2$ between O_1 and O_2 are equal if $k = k'$ and $\varphi(n) = \varphi'(n)$, for all $n \in \text{names}\, O_1$. Composition in K is the natural extension of composition of realisations.

For all $O = (A, B)$ we write $O \vdash phrase \Rightarrow Q$ as a shorthand for $O \in \text{Obj}$ and $A, B \vdash phrase \Rightarrow Q$.

Theorem 5.1 (Realisation and Principality) *Let phrase be one of sigexp, spec or funsigexp. Then* **(1)** *and* **(2)**.

(1) *For all O, O', Q, φ and k, if $O \vdash phrase \Rightarrow Q$ and $O \xrightarrow{\varphi,k} O'$ then $O' \vdash phrase \Rightarrow \varphi(Q)$*

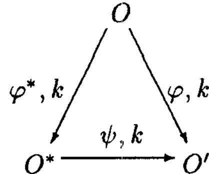
(2) *For all O, O', Q', φ and k, if*

$$O \xrightarrow{\varphi,k} O' \quad \text{and} \quad O' \vdash phrase \Rightarrow Q'$$

then there exist O^*, φ^* and Q^* depending only on O, phrase and k, such that

$$O \xrightarrow{\varphi^*,k} O^* \quad \text{and} \quad O^* \vdash phrase \Rightarrow Q^*$$

Moreover, there exists a ψ such that the diagram



commutes and $\psi Q^* = Q'$.

Theorem 3.1 follows from part **(2)** of the above theorem if we let A be the assembly of the structures that occur in B, let $O = O' = (A, B)$, $Q' = S'$, $\varphi = \text{Id}$, $k = 0$, $phrase = sigexp$ and $Q^* = S^*$.

The proof is by induction on the depth of inference. It is too long to be included here, but the construction of O^*, φ^* and Q^* in part **(2)** happens according to the algorithm W in Figure 11. The algorithm is named after the Damas-Milner algorithm with the same name [2] because of the great similarity. Given an expression e and a type assignment A, a call

$$(S, \tau) = W(A, e)$$

of their algorithm either returns (S, τ), where S is a substitution and τ (after suitable binding of type variables) is the principal type for e in $S(A)$, or fails, in case the expression does not elaborate at all. Similarly, a call

$$(O', \varphi, S) = W_{sigexp}(O, sigexp)$$

of our algorithm either returns (O', φ, S), where φ is a realisation and S (after suitable binding of names) is the principal signature for *sigexp* in O', or fails, in case the signature expression does not elaborate at all. The analogy is strengthened by the fact that O' is roughly $\varphi(O)$, except that O' in fact might be an non-trivial admissible cover of $\varphi(O)$, for reasons explained in the conclusion.

The realisation φ returned by the algorithm is the composition of realisations obtained for the subexpressions of *sigexp*. The only place non-trivial realisations arise is when a sharing specification is processed. In this case, the admissification algorithm mentioned in Section 4 is invoked. Apart from failing because of identifiers that are not found in the basis, the only way our algorithm W can fail is indirectly because *Admissify* fails. Again, this is similar to the behaviour or the Damas-Milner algorithm.

Our W algorithm actually consists of four mutually recursive functions, one for each group of inference rules in Section 3.8. In the last of these, W_{spec}, we use the following notation. For all $O = (A, B) = (A, (N, G, E))$, we write $Admissify(O, longstrid_1, longstrid_2)$ to mean $Admissify(A', N)$, where

$$A' = \begin{matrix} (A, & (m, \{X \mapsto B(longstrid_1)\}), \\ & (m, \{X \mapsto B(longstrid_2)\})) \end{matrix}$$

where m is new and X is an arbitrary structure identifier.

Notation used in Figure 11:

$\text{Clos}_A S$ means $(N)S$, where $N = \text{names}\, S \setminus \text{names}\, A$

$\text{Below}(A, A')$ is a minimal (with respect to $\sqsubseteq$) assembly A'' consistent with A satisfying that whenever $S = (m, E)$ occurs in A with m free and $m \in \text{names}(A', A'')$, then $S \sqsubseteq A''$

$O \,\text{as}\, (A, B)$ introduces a variable O and simultaneously introduces variables for the components of O

$B \,\text{of}\, O$ means the B component of object O; similarly for N of B and E of B

$O \uplus P$ is the object $((A, P), B)$, where $(A, B) = O$ and P is any semantic object. ($O \uplus P$ concerns the assembly component of O, whereas $O + P$ always concerns the basis component of O.)

6 Conclusion

Our type discipline is almost, but not completely, a conservative extension of the Standard ML modules discipline. The difference, as far as first-order modules is concerned, has to do with the notion of cover and the rôle of assemblies. Assemblies do not appear in the inference rules for Standard ML, although they play a prominent part in the proof of the principality theorem[4, Theorem A.2]. The reason why we have them in our inference rules has to do with cover. In ML, cover is only required at one point: ML allows

$W_{sigexp}(O \text{ as } (A, B), sigexp, k) : \text{Obj} \times \text{Rea} \times \text{Str} =$
case *sigexp* of
  longsigid =>
    let $(N^*)S^* = B(longsigid)$, where all names in
      N^* are chosen to be fresh and have rank k
    in $(O \uplus S^*, Id, S^*)$
  | **sig** *spec* **end** =>
    let $(O^*, \varphi^*, E^*) = W_{spec}(O, spec, k)$
      $S^* = (m^*, E^*)$, where m^* is new
    in $(O^* \uplus S^*, \varphi^*, S^*)$

$W_{prinsigexp}(O \text{ as } (A, B), sigexp, k) : \text{Obj} \times \text{Rea} \times \text{Sig} =$
let $(O^* \text{ as } (A^*, B^*), \varphi^*, S^*) = W_{sigexp}(O, sigexp, k)$
  $A_1^* = \text{Below}(A^*, \varphi^* A)$
  $\Sigma^* = \text{Clos}_{A_1^*} S^*$
in $((A_1^*, B^*), \varphi^*, \Sigma^*)$

$W_{funsigexp}(O, funsigexp, k) : \text{Obj} \times \text{Rea} \times \text{FunSig} =$
let
  $O = (A, B)$
  $(strid : sigexp_1) : sigexp_2 = funsigexp$
  $(O_1^*, \varphi_1^*, \Sigma_1^*) = W_{prinsigexp}(O, sigexp_1, k)$
  $(N_1^*)S_1^* = \Sigma_1^*$, where all names in N_1^* are
    chosen to he fresh and have rank $k + 1$
  $O_2 = O_1^* \uplus S_1^* + N_1^* + \{strid \mapsto S_1^*\}$
  $(O_2^*, \varphi_2^*, \Sigma_2^*) = W_{prinsigexp}(O_2, sigexp_2, k + 1)$
  $A_2^* = A \text{ of } O_2^*$
  $A^* = \text{Below}(A_2^*, \text{names}(A_2^*) \setminus N_1^*)$
  $\varphi^* = \varphi_2^* \circ \varphi_1^*$
  $\Phi^* = (N_1^*)(\varphi_2^* S_1^*, \Sigma_2^*)$
in $((A^*, \varphi_2^*(B \text{ of } O_1^*)), \varphi^*, \Phi^*)$

$W_{spec}(O, spec, k) : \text{Obj} \times \text{Rea} \times \text{Env} =$
case *spec* of
    empty => $(O, \text{Id}, \{\})$, where Id is the identity
  | **sharing** $longstrid_1$=$longstrid_2$ =>
    let $\varphi^* = Admissify(O, longstrid_1, longstrid_2)$
    in $(\varphi^* O, \varphi^*, \{\})$
  | $spec_1 \langle ; \rangle spec_2$ =>
    let $(O_1^* \text{ as } (A_1^*, B_1^*), \varphi_1^*, E_1^*) = W_{spec}(O, spec_1, k)$
      $(O_2^*, \varphi_2^*, E_2^*) = W_{spec}((A_1^*, B_1^* + E_1^*), spec_2, k)$
    in $((A \text{ of } O_2^*, \varphi_2^* B_1^*), \varphi_2^* \circ \varphi_1^*, \varphi_2^* E_1^* + E_2^*)$
  | **structure** *strid*: *sigexp* =>
    let $(O^*, \varphi^*, S^*) = W_{sigexp}(O, sigexp, k)$
    in $(O^*, \varphi^*, \{strid \mapsto S^*\})$
  | **functor** *funid*: *funsigexp* =>
    let $(O^*, \varphi^*, \Phi^*) = W_{funsigexp}(O, funsigexp, k)$
    in $(O^*, \varphi^*, \{funid \mapsto \Phi^*\})$

Figure 11: The Inference Algorithm

the conclusion $B \vdash sigexp \Rightarrow (N)S$, only when B covers Below(S, B)[5, Sec 5.13]. Were we to copy this restriction to the higher-order language, principality would be lost. To see this, consider Figure 9 again and imagine that the last sharing constraint is deleted and that B is a basis which contains two structures that have different names `m1` and `m2` but both have a Q substructure. One now has the following unpleasant options when elaborating *SIG* in B. Either one identifies the name of P' with `m1` or `m2` in order to satisfy the restriction when inferring a principal signature for $sigexp_4$, in which case one does not get a principal signature for $sigexp_3$, since the choice between `m1` and `m2` is arbitrary. Or one gives P' a fresh name, in which case the principal signature for $sigexp_4$ violates the requirement, because P' has no Q component.

For the higher-order language, we therefore need a way to allow components in S that are not covered by B when we infer a principal signature $(N)S$ in B. This is achieved by introducing assemblies into the elaboration trees and by modifying the definition of principality so that principality becomes relative to both B and A.

```
structure Empty = struct end;

signature SIG1 = sig
 structure E: sig
  structure Dangle: sig end
 end
 sharing E= Empty
end

signature SIG2 = sig
 local
  structure E: sig
   structure Dangle: sig end
  end
  sharing E= Empty
 in
 end
end;
```

Figure 12: Two pathological signatures

The consequences of these modifications are illustrated by the example in Figure 12. In ML, *SIG1* is illegal, while *SIG2* is legal. (To see this, notice that *SIG1* elaborates to a structure containing a component (*Dangle*) which is not covered by the basis; in *SIG2*, the dangling structure is not in the resulting signature because of the `local` specification, so *SIG2* is legal.) In our scheme they would both be illegal.

In all other respects, our discipline is consistent with

the ML discipline. In the view of the author, signatures like the ones in Figure 12 are pathological and differences in the treatment of them are unlikely to have any practical significance whatsoever.

Acknowledgements

This work is heavily based on the work on the semantics of ML and hence uses ideas due to Robin Milner and Robert Harper. I am also grateful to Robin Milner, David MacQueen, David Turner, Don Sannella, Maria-Virginia Aponte and John Hannan for interesting discussions and very valuable feedback. Also, I wish to thank the SERC Standard ML project and ESPRIT Semantique project for their financial support.

References

[1] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45(3):293–351, 1986.

[2] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[3] R. Harper, J. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 341–354, Jan. 1990.

[4] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[5] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[6] D. Remy. Typechecking records and variants in a natural extension of ml. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 77–88, ACM, Jan. 1989.

[7] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

Appendix: Inference Rules

Signature Expressions $\boxed{A, B \vdash sigexp \Rightarrow S}$

$$\frac{A, B \vdash spec \Rightarrow E}{A, B \vdash \mathtt{sig}\ spec\ \mathtt{end} \Rightarrow (m, E)} \tag{2}$$

$$\frac{B(sigid) \geq S}{A, B \vdash sigid \Rightarrow S} \tag{3}$$

$\boxed{A, B \vdash sigexp \Rightarrow \Sigma}$

$$\frac{A \trianglelefteq A' \qquad A', B \vdash sigexp \Rightarrow S \qquad (N)S \text{ principal for } sigexp \text{ in } A, B}{A, B \vdash sigexp \Rightarrow (N)S} \tag{4}$$

Functor Signature Expressions

$\boxed{A, B \vdash funsigexp \Rightarrow \Phi}$

$$\frac{A, B \vdash sigexp_1 \Rightarrow (N)S \quad N \cap \text{names}(A, B) = \emptyset \quad (A, S), B + N + \{strid \mapsto S\} \vdash sigexp_2 \Rightarrow \Sigma}{A, B \vdash (strid{:}sigexp_1){:}sigexp_2 \Rightarrow (N)(S, \Sigma)} \tag{5}$$

Specifications $\boxed{A, B \vdash spec \Rightarrow E}$

$$\frac{A, B \vdash sigexp \Rightarrow S}{A, B \vdash \mathtt{structure}\ strid{:}sigexp \Rightarrow \{strid \mapsto S\}} \tag{6}$$

$$\frac{A, B \vdash funsigexp \Rightarrow \Phi}{A, B \vdash \mathtt{functor}\ funid{:}funsigexp \Rightarrow \{funid \mapsto \Phi\}} \tag{7}$$

$$\frac{m \text{ of } B(longstrid_1) = m \text{ of } B(longstrid_2)}{A, B \vdash \mathtt{sharing}\ longstrid_1 = longstrid_2 \Rightarrow \{\}} \tag{8}$$

$$\frac{}{A, B \vdash \quad \Rightarrow \{\}} \tag{9}$$

$$\frac{A, B \vdash spec_1 \Rightarrow E_1 \qquad A, B + E_1 \vdash spec_2 \Rightarrow E_2}{A, B \vdash spec_1\ \langle;\rangle\ spec_2 \Rightarrow E_1 + E_2} \tag{10}$$