

# Extending Record typing to type parametric modules with sharing

María Virginia Aponte  
Conservatoire National des Arts et Métiers and INRIA \*

## Abstract

We extend term unification techniques used to type extensible records in order to solve the two main typing problems for modules in Standard ML: matching and sharing. We obtain a type system for modules based only on well known unification problems, modulo some equational theories we define. Our formalization is simple and has the elegance of polymorphic type disciplines based on unification. It can be seen as a synthesis of previous work on module and record typing.

## 1 Introduction

Building programs from modules can be safely done only if modules can be connected coherently. The difficulty of *module coherence checking* depends on the complexity and power of the module system considered. The SML module system [5, 7] is, among the modular programming languages with decidable checking, the most expressive existing one. SML modules allow multiple but consistent views of the same module (sharing), and parametrization of a module (functors) by a module specification (signatures), and they can be typed statically. Moreover, the module language itself is largely independent of the SML core language, and therefore provides a starting point to build module systems for many different programming languages.

Previous work on the static semantics of SML modules had been based on special-purpose term algebras to represent modules and on specific and complex static semantic notions [3, 7, 6], or does not address sharing nor multiple views of modules [4]. We show here that, this static description can be completely expressed by classical typing techniques such as polymorphic typing and the techniques developed to type extensible

records. Moreover, our type discipline leads to efficient typechecking algorithms, thanks to a local formalization of module consistency verification. Finally, in our discipline the exclusion of defective signatures comes for free. This stands in contrast with previous type systems, where static semantic constraints are globally imposed on inferred types in order to reject defective signatures. In our system, these conditions are ensured directly by the unification process.

We adapt the algebra of extensible record types proposed by Rémy [8, 10] to encode the types of modular objects, and extend its equational theory to perform unification on the terms obtained. The two main problems in module checking — matching a structure against a signature; finding types for signatures containing sharing constraints — are then expressed as unification problems modulo certain equational theories. We show that both problems have a principal solution whenever they have a solution. We then reformulate the module checking problem as a type system using these results. For this type system, we easily obtain a result on principal signatures, the equivalent of the principal type result for the ML core language. Finally, we show that principal signatures are no more than principal solutions for the problem of typing a signature by unification.

We start by presenting an overview of the typing problems related to SML modules. In section 3 we present the module language. In section 4 we discuss the analogies between record types and modules and present Rémy's extensible records and our language of module types. In the next two sections we give our solutions for the matching and the sharing problems. In section 7 we present a type system for modules and results on principality, consistency and defective signatures.

## 2 Overview of SML modules typing problems

Many modular languages have some form of encapsulation of types and values. This kind of *basic module* is called a *structure* in SML. Some languages also have *parameterized modules*, which are called *functors* in SML. A functor takes structures as arguments and produces

---

\*Author's address: INRIA Roquencourt, Projet Formel, B.P. 105, 78153 Le Chesnay, France. E-mail: Maria.Aponte-Garcia@inria.fr

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.

ACM-20th PoPL-1/93-S.C., USA

© 1993 ACM 0-89791-561-5/93/0001/0465...\$1.50

```

structure S =
  struct
    datatype t = C of int
    fun ts (C x) = C (x+1)
    structure A = struct ... end
  end;
signature  $\Sigma$  =
  sig
    type t
    val ts : t  $\rightarrow$  t
  end
structure S' = S :  $\Sigma$ ;

```

Figure 1: Example of a signature constraint

structures as results. When parametrizing a functor by a structure variable, a *specification* of the structure components must be provided; in SML, the specification for a structure is called a *signature*.

## 2.1 Signature Matching

A structure and a signature can be *matched* when the former satisfies the type specifications of the latter. Following the requirements of independence in modular development, it must be possible to have structures *richer* than their specifications, and also several *views* of the same structure. Then, a signature can be matched against a richer structure, resulting in a *constrained* view of that structure. Another way of constraining a structure is by matching it against the signature of a functor argument. Figure 1 shows an example of a structure  $S$  and a signature  $\Sigma$  that can be matched successfully. The resulting structure  $S'$  is a view of  $S$  where the sub-structure  $A$  is absent.

## 2.2 Polymorphism and generative typing

ML is a language with *generative typing*: two different type declarations produce two incompatible types. Each type constructor in ML is associated with a unique mark, or *stamp*, which is fixed through the program life. Type equality is determined by comparing type stamps.

The notion of type stamps is naturally extended to stamps in structures. Each structure declaration has a new stamp assigned (for instance, the structure  $S$  in figure 1). Nevertheless, constrained structures keep their original stamp: different views of the same structure share the same stamp.

Thus, the type of a structure can be represented by its stamp together with the representations of its components. Figure 2 shows a picture of a possible representation of structures  $S$  and  $S'$ . Stamps  $m$ ,  $n$ , and  $p$

are drawn inside circles, while components appear on arcs. In this paper, since we do not consider the typing of objects from the core language, we omit their type representation, and draw them as filled circles. This representation is both intuitive, and very near from the encoding of structures proposed by Harper, Milner and Tofte. As we shall see later, our encoding, which carries information, can be more advantageous checking module coherence.

Signatures can be represented in a similar fashion. The only difference is that stamps are no more fixed, but can take many “stamp values” when matching different structures. Intuitively, a signature can be seen as a structure where stamps are universally quantified variables. Roughly speaking, one can consider structure and signature types in modules, respectively as the equivalent of simple types and type schemes in the core language. This elegant analogy was first proposed by Harper, Milner and Tofte [3] when describing the semantics of SML modules. Figure 2 shows a representation of the signature  $\Sigma$ . The dashed circles stand for universally quantified stamps:  $x$  and  $y$  in this example.

Harper, Milner and Tofte [3, 7] and Tofte [11] developed several static semantic descriptions of SML modules using this analogy. In particular, they described matching as a process combining instantiation of universally quantified stamps with *enrichment* of signatures when they have less information than the matched structure. In this paper, we give a new description of the matching process, using techniques from subtyping with extensible records.

## 2.3 Typing sharing constraints

Two different structures can share the same type component. For instance,  $S'.t$  and  $S.t$  are the same type. In contrast, different specifications in different signatures are assumed not to share types, since each specification can be implemented by different actual types. Sometimes, however, one may need to consider two separate specifications as standing for the same type. The way to require this behavior is by adding *sharing constraints* in signatures. Consider the functor  $F$  in figure 3 using the signature  $\Sigma$  of figure 1.  $F$  would be ill-typed without the sharing constraint in the functor heading. Sharing can also be specified on structure and type identifiers, which can be either internal or external to signatures. The signature  $P$  is an example of sharing between internal and external structures.

## 2.4 Consistency

As SML allows having different views of the same structure, solving sharing constraints cannot be done by classical unification, but instead by a process identifying stamps in components. Indeed, some kind of con-

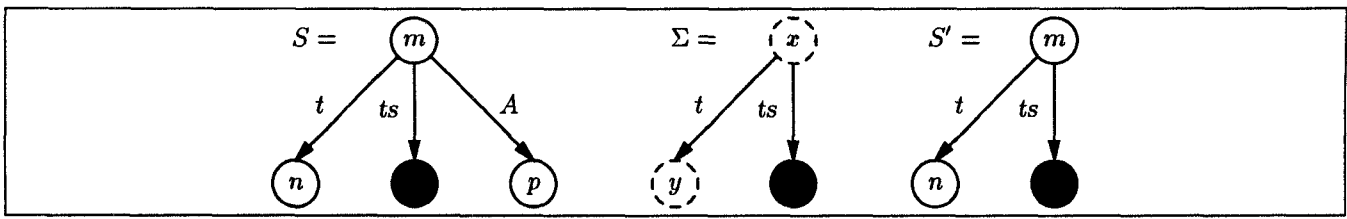


Figure 2: Representation of  $S$ ,  $S'$  and  $\Sigma$

```

functor F (X :  $\Sigma$ , Y :  $\Sigma$  sharing X.t = Y.t) =
  struct
    ... fun t.two = (X.ts) o (Y.ts) ...
  end;

signature P =
  sig
    structure M :
      sig
        structure A : sig end;
      end;
    sharing M = S'
  end

```

Figure 3: Sharing constraints

sistency between the structures sharing a given stamp must be also verified. From now on, we omit value and type components from module representations and concentrate only on sub-structure components and stamps. We justify this choice in the next section.

The weakest condition we can request on module consistency is that all the structures sharing the same stamp have the same stamp on their common components. Consistency can be easily preserved by the pruning process of signature constraint, as there is no need to create nor to destroy sharing when constraining a structure.

In contrast, solving sharing does need to verify that consistency is preserved while performing stamp identification. This is not straightforward. Consider the signature  $P$  of figure 3. It has a sharing constraint between a structure  $M$  and the structure  $S'$  described in figure 1. There,  $S'$  is a constrained view (without the sub-structure  $A$ ) of  $S$ . On the other hand,  $M$  has an  $A$  component. In spite of the fact that  $S$  does not appear in the sharing constraint, one must look at it while solving the sharing, and, in general, we must look at any structure sharing a stamp with  $M$  or  $S'$ . Otherwise, a naive solution will lead to an inconsistent assembly of structures, and then to an incorrect signature, as in figure 4. The structure  $M^*$  is obtained from  $M$  by solving the sharing between  $M$  and  $S'$  in a local way, only looking at these two structures. The result is in-

correct since, without considering the sub-structure  $A$  of  $S$ , the  $y$  variable is not identified with  $p$ . The structures  $S$  and  $M^*$  are inconsistent, because they share the stamp  $m$  and have a common component  $A$  with two different stamps. Thus, the unification process must verify on the whole assembly of structures to preserve consistency and obtain correct signatures.

As we will see in the next section,  $M^*$  is not only inconsistent with the current typing environment, but it is also a defective signature.

## 2.5 Defective signatures

Not every signature one can build from stamps and sub-structure components can be considered legal. For instance, we can find signatures that are impossible to match with any possible structure for a given module typing context. In this section, we present the notion of *defective signature* used by Milner and Tofte [6] (they also call them *monsters*) to describe signature types that do not make sense with respect to a given typing context, but which could be inferred under their type rules if no care were taken. In the study of new type disciplines for SML modules, the problem is to determine whether defective signatures can be inferred, and if so, how to exclude them. In SML modules, signatures belong to the language constructions and they are actually part of the code; they are used in particular to build functors. Defective signatures are then bad code, and it is nice to have a typing discipline clever enough to reject them.

Our aim in this section is to characterize defective signatures and give some examples. In section 7.4 we show how within our type discipline it is impossible to obtain defective signatures for the same examples: our type system fails in typing them.

Milner and Tofte introduced some global conditions on inferred types in order to exclude defective signatures. We use these conditions to characterize defective them.

The first one is *well-formedness*. Recall that signatures have universally quantified variable stamps. If a free stamp occurs in a signature, this means that it is shared with at least one real structure in the type context. A signature is *well-formed* if all the components attached to that free stamp have also free stamps. Con-

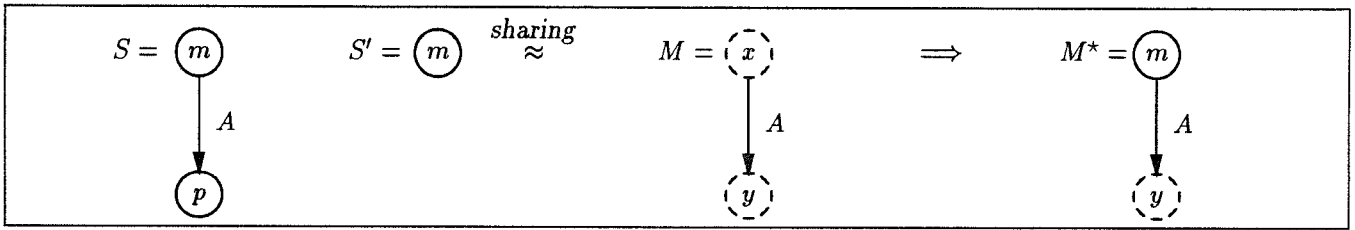


Figure 4: An inconsistent assembly of structures

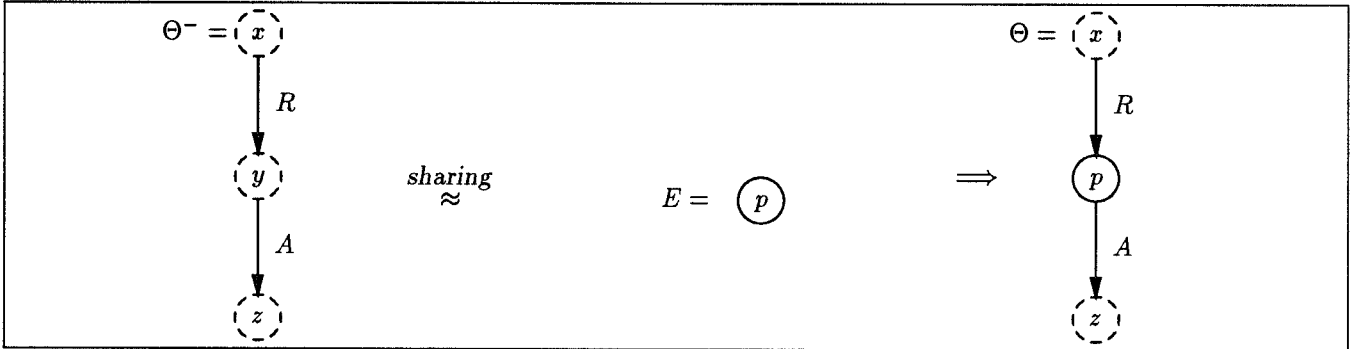


Figure 6: An ill-formed signature

```

structure E = struct end;
signature theta =
  sig
    structure R :
      sig
        structure A : sig end
      end;
    sharing R = E
  end

```

Figure 5: Creation of an ill-formed signature

```

structure C1 = struct end;
structure C2 = struct end;
signature Omega =
  sig
    structure A :
      sig structure B : sig end end;
    sharing A = C1; sharing A.B = C2
  end

```

Figure 7: Uncovering within a signature

sider the declarations of figure 5 and their corresponding "types" in figure 6. We call  $\Theta^-$  the type of  $\Theta$  before solving the sharing constraint, and  $\Theta$  its type after solving it. This latter is ill-formed since the bound stamp  $z$  is under the free stamp  $p$ . Since there is no other structure than  $E$  having a stamp  $p$  and also an  $A$  component, it is impossible to match the result signature  $\Theta$  with any existing structure, and even to create a new structure able to match it. In section 7.4 we show how our type rules fail in assigning a type to  $\Theta$ .

The second global condition is *covering*. We introduced it using the examples in figures 7 and 8. The signature  $\Omega$  in figure 8 is well-formed but will never match any real structure. The reason is that no structure having the stamp  $m$  can have a sub-structure  $B$  on it: the structure  $C_1$  where  $m$  comes from was originally created empty. Roughly speaking, a structure  $S$  having a fixed stamp  $m$  is *covered by the typing context*  $\Gamma$ , if,

for every  $Q$  component of  $S$ , there is in  $\Gamma$  a structure  $S'$  sharing  $m$ , and which has a  $Q$  component. In figure 8, the sub-structure  $A$  of  $\Omega$  is uncovered as the only structure sharing  $m$  with  $A$  is  $C_1$  and it does not contain any  $B$  component.

Harper, Milner and Tofte impose well-formedness as a global constraint on any inferred object and covering on any inferred signature, thus rejecting defective signatures. In section 7.4 we show how, with our encoding, matching, and sharing resolution, we do not need any of these side conditions to guarantee that only legal signatures are obtained: our type discipline directly fails to assign types for the two examples above.

### 3 The language ModL

The matching and sharing problems can be studied in a simplified language where modules contain only stamps

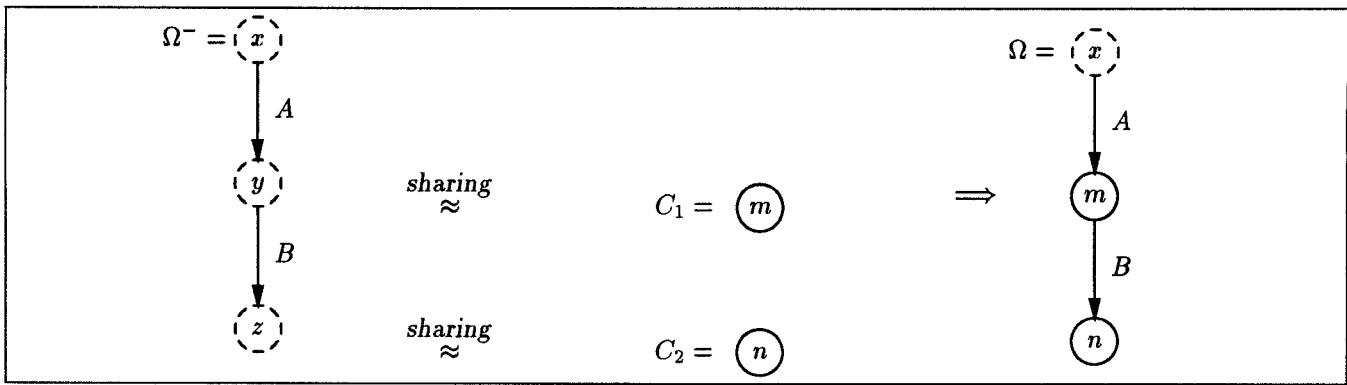


Figure 8: Creation of an uncovered signature

<i>Strexp</i>	<code>::= Strid</code>
	<code>Strexp . Strid</code>
	<code>Strexp : Sigid</code>
	<code>Funid ( Strexp )</code>
	<code>struct Dec end</code>
<i>Dec</i>	<code>::=</code>
	<code>Dec ; structure Strid = Strexp</code>
<i>Sigexp</i>	<code>::= Sigid</code>
	<code>sig Spec end</code>
<i>Spec</i>	<code>::=</code>
	<code>Spec ; structure Strid : Sigexp</code>
	<code>Spec ; sharing Longid = Longid</code>
<i>Funexp</i>	<code>::= func ( Strid : Sigid ) Strexp</code>
<i>Longid</i>	<code>::= Strid</code>
	<code>Longid . Strid</code>
<i>Program</i>	<code>::= structure Strid = Strexp</code>
	<code>signature Sigid = Sigexp</code>
	<code>functor Funid = Funexp</code>
	<code>Program in Program</code>

Figure 9: The language ModL

and information about their sub-structure components. This is justified as the matching between value types, exception types, and type declarations can be easily defined using the classical subsumption ordering between polymorphic types. Also, following the SML Definition [7], we do not consider sharing between value types. Finally, most of the problems of sharing between type declarations can be solved in the same fashion as sharing between sub-structures. What is left to study, then, is the matching and sharing problems with respect to identification of stamps, and consistency and enrichment with respect to sub-structures.

ModL was first introduced by Harper, Milner and Tofte [7] and Tofte [11] to study SML module typing. The language we present is a slight modification of their calculus, where the functor construction has been

split in two syntactic constructions to make typing rules shorter. ModL is a simplification of SML modules with no values, type declarations, or exceptions, but only structures, signatures, and functors. We assume three disjoint identifier classes: *Strid* for structures, *Sigid* for signatures and *Funid* for functors. The ModL syntax is presented in figure 9.

## 4 Extensible records and modules

Modules and records are both built from labeled components. Moreover, module matching allows signatures to be matched with richer structures, much as greater records can be accepted in place of records with less information. For records, this problem is strongly connected to subtyping.

Our proposal exploits the similarity between the record and module type representations and between their typing problems.

### 4.1 Extensible record terms

The basic idea behind Rémy's discipline is to have record types defining different sets of labels and then allow unification on types of different sizes. Rémy's records are defined by a term algebra together with an equational theory controlling their unification. They use *row variables*, which were first introduced by Wand [13], to allow the extension of a record by new fields in a polymorphic way. They also use *flags* on fields indicating their *presence* or *absence*. For instance, the record type  $t$  asking for the presence of the field  $a$  (if for instance, a function extracts it from its argument), with an arbitrary type  $\alpha$  is shown in figure 10.

*Extension* is performed by substitution on row variables during unification against a larger record. In figure 10 the type  $t$  has a row variable  $\rho'$ , so it can be extended. But  $a$  is the only component that  $t$  must

$$r = \begin{array}{|c|c|c|} \hline a : \varepsilon_1. \sigma & b : \varepsilon_2. \tau & abs. \rho \\ \hline \end{array} \quad t = \begin{array}{|c|c|} \hline a : pre. \alpha & abs. \rho' \\ \hline \end{array} \quad r^* = \begin{array}{|c|c|c|} \hline a : pre. \sigma & b : abs. \tau & abs. \rho'' \\ \hline \end{array}$$

Figure 10: Record extension by unification

ask for. Therefore,  $\rho'$  is tagged by *abs*, and thus any subsequent extension will lead to absent fields.

*Variable flags* are used when a field can be taken either as present or absent, giving the possibility to render some type information invisible. Subtyping is achieved by allowing unification on types with different sizes: variable flags get instantiated to present or absent, row variables are instantiated to make types richer.

Consider the example of figure 10. The record type  $r$  has fields  $a$  and  $b$  with types  $\sigma$  and  $\tau$ . Both fields are tagged by variable flags, so they can be taken as present or invisible. In a similar way,  $r$  can be extended only by absent fields. The unification of  $r$  and  $t$  results in a term  $r^*$  where the flag variable of  $a$  in  $r$  is instantiated to present, the row variable of  $t$  is extended by an absent  $b$  field, and the  $b$  flag is instantiated to absent.

## 4.2 The module terms

In figure 10, if we take  $r$  as the type of a structure, and  $t$  as the type of a signature specifying an  $a$  component, then, the result from the unification above is exactly the signature constraint of  $r$  by  $t$ ; that is, in the resulting term, the  $b$  component is no longer visible.

We exploit this similarity and present the grammar corresponding to module terms. Unfortunately, Rémy's theory is not powerful enough to express all matching cases one wants to type in modules. The problem comes from the encoding of subtyping using variable flags. Once a flag variable has been instantiated, say, to present, it can no longer be compared with the same field component having an incompatible absent flag, even if this flag naturally represents less information. Thus, some cases of signature matching cannot be typed. Consequently, we do not use flag variables in our encoding of terms, but only absent and present flags. The problem with this is how to achieve subtyping during matching. We shall do it by introducing in the next section an order between terms tagged by *abs* and *pre*.

Record terms are very general and admit many concrete signature of symbols. In our version above, we simply add stamps and write some symbols differently to be closer to the module terminology. The structure constructor *Str* takes pairs of stamps and structure environments. Rows are either variables, sequences of fields, or the empty row made of a non-labeled field. Rémy's records are sorted to restrict the terms that can meet by unification, and also to forbid label rep-

etition. To simplify the notation, we do not give the details of sorts, but keep these restrictions in mind when considering the type rules and the record equations. The symbol  $\emptyset$  is the empty structure, which will be used to prevent extension on structure types. We write  $\mathcal{T}$  the for term language obtained. The assertion list  $a_1 : \sigma_1 :: \dots :: a_n : \sigma_n :: []$  is also written  $[a_1 : \sigma_1; \dots; a_n : \sigma_n]$ . The symbols  $\alpha, \beta$  are structure variables,  $\chi$  is a row variable,  $\theta$  is a field variable, and  $x$  is a stamp.

$$\begin{array}{ll} \sigma ::= \alpha \mid Str(x, \rho) \mid \emptyset & \text{structures} \\ \varsigma ::= [] \mid (a : \sigma) :: \varsigma & \text{assertion lists} \\ \phi ::= \varphi \mid \sigma \rightarrow \sigma & \text{functors} \\ \rho ::= \chi \mid a : \varepsilon; \rho & \text{rows} \\ & \mid \varepsilon \mid \pi \cdot a : \sigma; \sigma \\ \varepsilon ::= \theta \mid \pi \cdot \sigma & \text{fields} \\ \pi ::= abs \mid pre & \text{flags} \end{array}$$

The equational theory of record terms allows commutativity of fields and the extension of row variables. We present the laws below and write  $E$  for this theory and  $\stackrel{E}{=}$  for its equality.

$$a : \varepsilon; b : \varepsilon'; \rho \stackrel{E}{=} b : \varepsilon'; a : \varepsilon; \rho$$

$$\pi \cdot (a : \sigma; \tau) \stackrel{E}{=} a : \pi \cdot \sigma; \pi \cdot \tau$$

The first axiom states  $E$ -equality modulo the reordering of fields. The second, states the distributivity of flags over the  $a : \_;$  row constructor. This axiom allows the extension of a row by new components. For instance, in our encoding, a row variable in a signature has always the form  $abs(\alpha)$ . We can extend it by an  $a$  component plus a non-extensible row by substituting  $a : \sigma; \emptyset$  for  $\alpha$ . Using the second axiom we obtain the term (in the right hand side):

$$abs \cdot (a : \sigma; \emptyset) \stackrel{E}{=} a : abs \cdot \sigma; abs \cdot \emptyset$$

Notice that in our grammar we admit terms like  $abs \cdot \sigma$  and  $abs \cdot (a : \sigma; \rho)$ , but we do not admit terms like  $abs \cdot (a : abs \cdot \sigma; abs \cdot \rho)$ . Notice also that, assertion lists are different from rows. Assertion lists are built with the  $::$  constructor and rows with the  $a : \_;$  constructor. In particular, the flag distributivity axiom works only on rows and never on assertion lists.

## 5 Solving the matching problem

We extend the theory  $E$  above to compare structure and assertion lists of  $\mathcal{T}$  having incompatible *abs* and

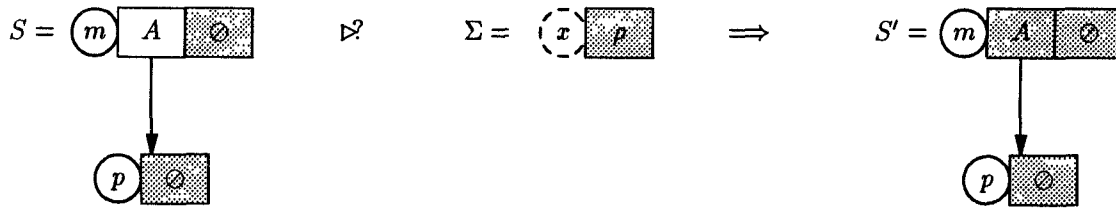


Figure 11: Matching modules

pre flags (under  $E$ ). The relation  $\triangleright$  of *enrichment* is defined as the smallest transitive and  $E$ -reflexive relation (i.e., containing  $\stackrel{E}{=}$ ) satisfying the rules below. We write  $C$  for any constructor of the module terms except  $\rightarrow$ . Notice that this relation does not consider functors and that it is co-variant. The  $\triangleright$  relation is a structural subtyping relation simpler than the subtyping problems studied in [2] where contravariance is considered<sup>1</sup>.

$$\frac{}{\text{pre} \triangleright \text{abs}} \quad \frac{\sigma_1 \triangleright \tau_1 \dots \sigma_n \triangleright \tau_n}{C(\sigma_1, \dots, \sigma_n) \triangleright C(\tau_1, \dots, \tau_n)}$$

In [1] we prove that  $\triangleright$  is stable under substitutions so we can perform unification modulo it. We write *inequation of enrichment* for the unification problem given by the expressions  $\sigma \triangleright \tau$  between the module terms  $\sigma$  and  $\tau$ , and we say that  $\mu$  is a solution to this problem if  $\mu\sigma \triangleright \mu\tau$  holds. If  $\sigma$  is the encoding of a structure and  $\Sigma$  is the encoding of a signature, the matching problem between  $\sigma$  and  $\Sigma$  is given by the inequation  $\sigma \triangleright \Sigma$  and solved by unification<sup>2</sup>. When it has a minimal solution  $\mu$ , the structure resulting from the constraint of  $\sigma$  by  $\Sigma$  is given by the term  $\mu\sigma$ .

Consider the example in figure 11. The structures  $S$  and the signature  $\Sigma$  of figure 1 are now represented without using variable flags. Absent components are drawn in grey boxes, while present ones are drawn in white boxes. To solve the matching problem  $S \triangleright \Sigma$ , the variable  $\rho$  must be extended by the  $A$  component of  $S$ . Since  $\rho$  is grey, the extension in the resulting term  $S'$  is also grey. The term obtained is a restricted vision of  $S$  where  $A$  is absent.

The inequation above has a principal solution under  $E$ . This is not always the case. Consider the inequation  $\text{Str}(m, A : \text{pre} \cdot \sigma; \rho) \triangleright \text{Str}(m, \chi)$ . It has two minimal  $E$ -solutions:  $\mu = \{\chi \mapsto A : \text{pre} \cdot \sigma; \rho\}$  and  $\mu' = \{\chi \mapsto A : \text{abs} \cdot \sigma; \rho\}$ . We choose the second one, which is minimal under the order  $\triangleright$ . This choice corresponds to an interpretation of  $\triangleright$  as a constraining relation: in the right hand of an inequation, we choose the solution revealing the least of information.

**Lemma 1** *Any matching problem having a solution has also a principal solution under  $\triangleright$ .*

<sup>1</sup>In [1] we studied a contravariant version of  $\triangleright$ .

<sup>2</sup>Actually, unification is a more general technique than what we need to solve this problem, but this is just a more general framework.

This lemma is proved by showing the existence of an algorithm which finds  $\triangleright$ -principal solutions for inequations. In appendix A, we present an algorithm  $\mathcal{S}i$  simplifying an inequation into a system of *elementary* equations and inequations. An inequation is *elementary*, if it contains a variable on one side and either a variable, or a constant on the other side. An equation is *elementary* if it has the form  $\alpha \stackrel{E}{=} \sigma$ . A simplified system is *equivalent* to the initial inequation when they both have the same principal solutions.

**Lemma 2** *The algorithm  $\mathcal{S}i$  simplifies an inequation into a system of elementary equations and inequations, when it admits a solution and fails otherwise. The simplified system is equivalent to the initial inequation.*

In the output of  $\mathcal{S}i$  there are not two equations or inequations on the same variable, and moreover, the inequations are only of the forms:  $\alpha \triangleright \text{abs}$ ,  $\text{pre} \triangleright \alpha$ , and  $\alpha \triangleright \beta$ , which admit several solutions. The complete algorithm of inequation resolution involves three steps. First, decomposing an inequation into a simplified system; second, checking for the existence of cycles (occur check); third, choosing a  $\triangleright$ -minimal solution for the simplified inequations having several solutions. Occur check is well known and  $\triangleright$ -minimal solutions for the above inequations are trivial. The proof of the  $\triangleright$ -principality the algorithm solutions can be found in [1].

## 6 Solving sharing constraints

Putting absent flags hides some fields without destroying their type information. Consider the restricted view  $S'$  of  $S$  in figure 11. Both structures have exactly the same stamps and type information, the only difference being in the flags carried (box color) by the components. This has an important effect on the consistency verification for modules. The unification process of sharing resolution becomes purely local: it does not need to examine structures other than those in the sharing constraint to verify that consistency is preserved.

In this section we develop a second extension of Rémy's equations, this time solving the sharing problem. It states the equivalence of terms modulo some differences in the flags of components. We call the new

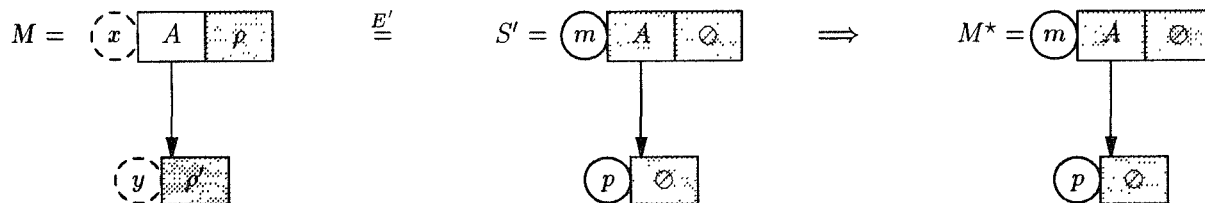


Figure 12: Solving sharing constraints

theory  $E'$ . It adds the axiom  $abs \stackrel{E'}{=} pre$  to the previous theory  $E$ .

A *sharing equation* has the form  $\sigma \stackrel{E'}{=} \tau$  for the module terms  $\sigma$  and  $\tau$ . Now, if  $\sigma$  and  $\tau$  are two structures appearing in a sharing constraint, solving this constraint is equivalent to solving the sharing equation  $\sigma \stackrel{E'}{=} \tau$ . But the  $E'$  theory identifies *abs* and *pre* flags in order to compare different views of structures. To identify the solutions in the same way would be incorrect, so we must consider them modulo the equations  $E$ . As for inequations, sharing equations can have several  $E$ -minimal solutions, but when they have one, they also have a  $\triangleright$ -minimal solution. Again, this is the solution we are interested in, as it expresses exactly the amount of visible type information necessary to satisfy sharing constraints within a module semantics allowing different views of structures. The unification algorithm for sharing resolution is an easy extension of Rémy's algorithm that can be found in [9]. The proof of the lemma below can be found in [1].

**Lemma 3** *Any sharing equation having a solution has also a principal solution under  $\triangleright$ . There exists an algorithm that finds this solution when it exists and fails otherwise.*

Now, given a  $\triangleright$ -principal solution  $\mu$  for the equation  $\sigma \stackrel{E'}{=} \tau$ , the structures satisfying a sharing constraint between  $\sigma$  and  $\tau$  are given by the terms  $\mu\sigma$  and  $\mu\tau$ .

As we saw above, the advantage of this encoding is that the inconsistency we considered in section 2.4 cannot arise any more when solving sharing constraints by local unification, that is, when unifying only the structures specified in the sharing constraint. Fig 12 shows our encoding and solution to the sharing problem in figure 4 of section 2.4. Now,  $S'$  has all the stamps and components of  $S$ , so when unifying  $S'$  and  $M$  — the only structures in the constraint —  $S'$  has all the necessary information to guarantee that consistency is preserved in the result  $M^*$ . If we call  $\mu$  the  $\triangleright$ -principal solution of  $S' \stackrel{E'}{=} M$ , the structure  $M^*$  resulting from  $M$  is given by  $\mu M$ . The structure  $S'$  remains unchanged as it has only fixed stamps (in solid circles).

## 7 Typing ModL

In appendix B, we present a type system for checking modules using our module terms, and the sharing and matching results described above. We extend module types to module type schemes in order to capture polymorphism of stamps in signatures, and to allow row extensions during unification. A *module type scheme* is a module type with some variables universally quantified. We write  $\forall W \cdot \sigma$ . the module type  $\sigma$  when the variables in the set  $W$  are universally quantified.

A *module context*  $\Gamma$  is a triple  $(\Gamma_s, \Gamma_t, \Gamma_f)$  of partial functions of finite domain, mapping respectively structure, signature, and functor identifiers into structure types, structure schemes, and functor schemes. We define  $\mathcal{V}(\sigma)$  as the set of variables appearing in  $\sigma$ . We extend  $\mathcal{V}$  to module contexts  $\Gamma$  and note it  $\mathcal{V}(\Gamma)$ . Substitutions are defined in the classical way: on free variables of terms. We extend them to module contexts in such a way that the assertion  $x : \sigma$  in  $\Gamma$  becomes  $x : \mu\sigma$  in  $\mu\Gamma$ .

### 7.1 Functor schemes and generativity

We have seen that structure stamps are *fixed* while signature stamps are bound variables. What happens with functor stamps? A functor scheme is constructed with  $\rightarrow$ . By the SML module semantics, each new structure obtained by functor application has new stamps associated; that is, structure creation is generative under functor application. The new stamps are those appearing only on the right hand side of the arrow, i.e., those that are not instantiated by the actual functor argument. We call them *generative variables*. They will be transformed into *new fixed stamps* by the functor application, that is, they will never take the value of a stamp appearing already in  $\mathcal{V}(\Gamma)$ .

Suppose now we have *instantiation substitutions* on functor schemes, that is, for a type  $\forall W \cdot \tau$ , a substitution  $\mu : W \rightarrow \mathcal{V}$ . We must forbid them from substituting stamps appearing in the typing context for generative variables. In the rules (see appendix B) we use a predicate  $Gen(\mu, \Gamma, \forall W \cdot \phi)$  over instantiation substitutions, type contexts, and functor schemes. It holds exactly when the image of the substitution  $\mu$  applied to the generative variables of the functor scheme is disjoint from

free stamps in the typing context. The exact definition of this predicate is given in the appendix B.

## 7.2 Principal Signatures

Building functors and constraining structures in SML requires the specification of signatures. In both cases, one must ensure that the inferred signature corresponds exactly — in components and sharing — to the user’s specification. This property is known as *principality of signatures*. Checking for principality is crucial in order to give a correct semantics to module typing: building functors or constraining structures with non principal signatures would lead to objects different from those explicitly specified by the programmer.

We adapt the principality definition of Harper, Milner and Tofte [7] to our formalism. Let  $\sigma$  be a signature type such that  $\Gamma \vdash \text{sigexp} \Rightarrow \sigma$ . We say that  $\sigma$  is *principal for sigexp in the context  $\Gamma$* , if for any other signature type  $\sigma'$  such that  $\Gamma \vdash \text{sigexp} \Rightarrow \sigma'$ , there exists an instantiation substitution  $\mu : \mathcal{V}(\sigma) \setminus \mathcal{V}(\Gamma) \rightarrow \mathcal{T}$  such that  $\sigma' \triangleright \mu\sigma$ . That is, if any other signature type inferred for *sigexp* is richer under  $\triangleright$  than an instance of  $\sigma$ .

Our type rules for signature declaration and functor abstraction have side conditions on principality of signatures. This situation is to contrast with the ML core typing: since no type constraints can be possibly stated by the user, no special side conditions are necessary to ensure that the user’s specifications are not violated. This is one of the major points where the analogy between the core language and the module typing breaks.

Nevertheless, by our method to type signatures we still obtain a nice analogy between principal types in the core and in the module languages. We type a signature *sigexp* by solving a unification problem: the principal solution to this problem give us the *principal signature* (in the sense introduced above) for the signature expression *sigexp*. Thus, we express signature principality typing as the principal solution of an equational unification problem; much as principal typing in ML is expressed using principal solutions for a classical unification problem.

We now introduce the unification problem we use to type signatures, noted  $\Gamma \vdash \text{sigexp} \stackrel{?}{\Rightarrow} \alpha$ , between a context  $\Gamma$ , a signature specification *sigexp*, and a variable  $\alpha$ . We say that this problem has a solution  $\mu$  if  $\Gamma \vdash \text{sigexp} \Rightarrow \mu\alpha$  holds<sup>3</sup>. This problem is solved essentially using unification for sharing resolution. Then, from our previous result (lemma 3) on the existence of  $\triangleright$ -minimal solutions for sharing equations, we can easily show the following lemma. The algorithm  $\mathcal{W}$  is actually the signature typing algorithm for our type discipline. We present it in appendix C.

<sup>3</sup>This unification problem can be set this way because the type rules have been proved stable under substitutions in [1].

**Lemma 4** *If the unification problem  $\Gamma \vdash \text{sigexp} \stackrel{?}{\Rightarrow} \alpha$  has a solution, then it has a  $\triangleright$ -principal solution  $\mu$  such that  $\Gamma \vdash \text{sigexp} \Rightarrow \mu\alpha$  holds. There exists an algorithm  $\mathcal{W}$  finding this solution when it exists and failing otherwise.*

With this result, it is easy to show the following theorem. It states that principal signatures are precisely the principal solutions for the unification problem of typing signatures. That solution is found by the signature typing algorithm  $\mathcal{W}$  given in appendix C.

**Theorem 1** *If  $\Gamma \vdash \text{sigexp} \Rightarrow \sigma'$  holds, then the typing signature algorithm  $\mathcal{W}$  finds a signature  $\sigma$  which is principal for sigexp in  $\Gamma$  or fails otherwise.*

Proof: If  $\Gamma \vdash \text{sigexp} \Rightarrow \sigma'$  holds, then, there exists a solution  $\nu$  for the unification problem  $\Gamma \vdash \text{sigexp} \stackrel{?}{\Rightarrow} \alpha$ , such that  $\sigma' = \nu\alpha$  and there exists also a  $\triangleright$ -principal solution  $\mu$ , for the same problem, such that  $\mu\alpha = \sigma$ . But  $\mu$  is  $\triangleright$ -principal, and then there exists a substitution  $\eta$  such that  $\sigma' \triangleright \eta\sigma$ . We conclude that  $\sigma$  is principal for *sigexp*. ■

An immediate corollary of this theorem is the existence of principal signatures in our type system.

## 7.3 Consistency preservation

We saw that two structures are consistent if, when they share a stamp, their common components also share their stamps. This condition can be formalized in our type discipline as follows. Two structure terms  $\sigma$  et  $\tau$  sharing the same stamp are *consistent* if the sharing equation  $\sigma \stackrel{E'}{=} \tau$  holds. This notion is naturally extended to sets of module terms and to the set of module terms in a context  $\Gamma$ . By abuse of notation, we will talk about the consistency of the set of terms in  $\Gamma$  together with the term  $\sigma$ , as the consistency of the set  $\{\Gamma, \sigma\}$ .

We say that *consistency is preserved* by a module type judgment  $\Gamma \vdash e \Rightarrow \sigma$ , if whenever  $\Gamma$  is consistent, the set  $\{\Gamma, \sigma\}$  is also consistent.

Inconsistency can arise when introducing sharing between a set of terms which are not  $E'$ -equal. The only rule that introduces sharing is (*Nsig*) which creates new signatures. In this rule, there is no restriction on the stamp chosen to be on top of the new signature. In particular, it can be a stamp already appearing in the type context on top of a term  $E'$ -different from the new signature. It is easy to show that consistency is preserved under all the module type judgments except for those in the *Sigexp* and *Spec* syntactic classes of signatures. Indeed, these rules either introduce new stamps (they are generative), or they inherit whole terms (without introducing any sharing) appearing in the type context. The proof of the following lemma can be found in [1].

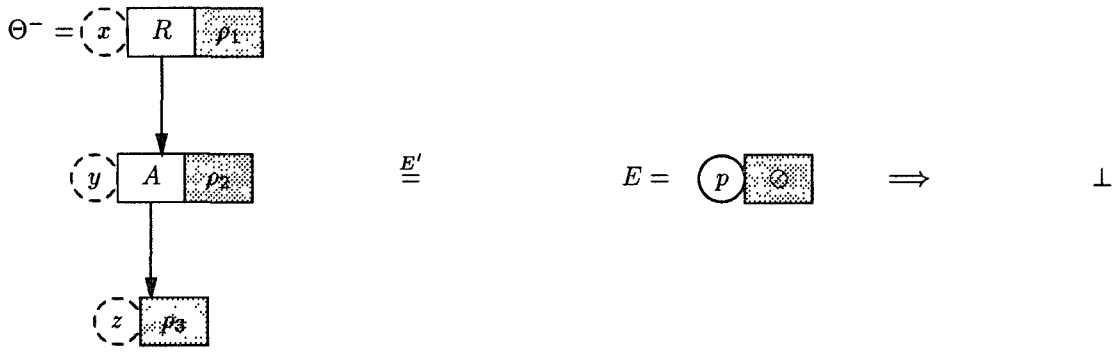


Figure 13: Unification failing on an ill-formed signature

**Lemma 5** *For any ModL expression  $e$  not in Sigexp, nor in Spec, the judgment  $\Gamma \vdash e \Rightarrow \sigma$  preserves consistency.*

On the other hand, our unification notions have a good behaviour with respect to consistency. The proof of the following lemma is trivial.

**Lemma 6** *Let  $\{\Gamma, \sigma\}$  be a consistent set. If  $\sigma \stackrel{E}{\equiv} \tau$  or  $\sigma \stackrel{E'}{\equiv} \tau$  hold, then the set  $\{\Gamma, \sigma, \tau\}$  is also consistent.*

The signature type algorithm in appendix C uses unification which is well behaved by this lemma. The algorithm takes several precautions with respect to stamp variables. First, it forbids unification on fixed stamps (appearing in real structures) of the type context. Second, it always chooses new stamp variables to build types for new signature specifications. Also, already existing signatures are instantiated by completely new stamp variables. In both cases, sharing is finally achieved by  $E'$ -unification. The following theorem states the existence of a signature type algorithm preserving consistency. The proof can be found in [1].

**Theorem 2** *Let  $\Gamma$  be a consistent module context and  $\mu$  the solution found by the algorithm  $\mathcal{W}$  for the  $\Gamma \vdash \text{sigexp} \stackrel{?}{\Rightarrow} \alpha$  typing problem; then the set  $\{\Gamma, \mu\alpha\}$  is also consistent.*

## 7.4 Rejection of defective signatures

In this section we consider the examples of defective signatures inferred under the Harper, Milner and Tofte type system that we presented in section 2.5. Our type rules fail in assigning a type to these two examples thanks to our encoding of terms, and to our definitions of matching and sharing resolution.

Figure 13 presents our solution for the ill-formedness example of figure 6. In contrast with the unification definition of Harper, Milner and Tofte [7, 6], with our encoding and definition of unification, it is impossible to unify the structure  $E$  with the term attached to  $R$ .

To unify them, one would need to extend  $E$  by an  $A$  component, which is impossible by the encoding of  $E$ .

Let us now consider the covering example of figures 7 and 8. In our type system, the typing of  $\Omega$  fails because it is impossible to unify the type of the structure  $C_1$  — empty and non extensible — with the type of the substructure  $A$  containing a  $B$  component. Figure 14 shows our type discipline failing on the typing of  $\Omega$  because of this sharing resolution failure.

Under our type rules, the two signature specifications above are rejected. Unfortunately, it is actually possible to build badly formed and uncovered signatures using our types rules. More precisely, we cannot state a general result on preservation of well-formedness and covering for our type rules. As for consistency, the rule creating problems is ( $N\text{sig}$ ): using it we can build arbitrary pathological signatures, either inconsistent, badly-formed, or uncovered.

But actually, things are not too bad. In the real world of typing, one does not try to guess arbitrary types but just to build very general types (directed by the syntax), which are finally specialized by unification. We saw above that unification behaves correctly: there will not be any inconsistency, bad formation, or uncovering introduced via unification on a set of consistent, well-formed, and covered objects. The following lemma, which is easy to show, formalizes this good behaviour. It extends lemma 6 of the previous section. Covering and well-formedness are extended to module type contexts.

**Lemma 7** *Let the set  $\{\Gamma, \sigma\}$  be consistent, well-formed, and covered. If  $\sigma \stackrel{E}{\equiv} \tau$  or  $\sigma \stackrel{E'}{\equiv} \tau$  hold, then the set  $\{\Gamma, \sigma, \tau\}$  is consistent, well-formed, and covered.*

Our notion of unification and particular encoding of terms intervene in the type rules. This is why the two defective examples above are rejected. On the other hand, we actually find types for signatures by solving a unification problem. We just have to put together the consistency results of the previous section with the lemma below to extend the good behaviour of unifica-

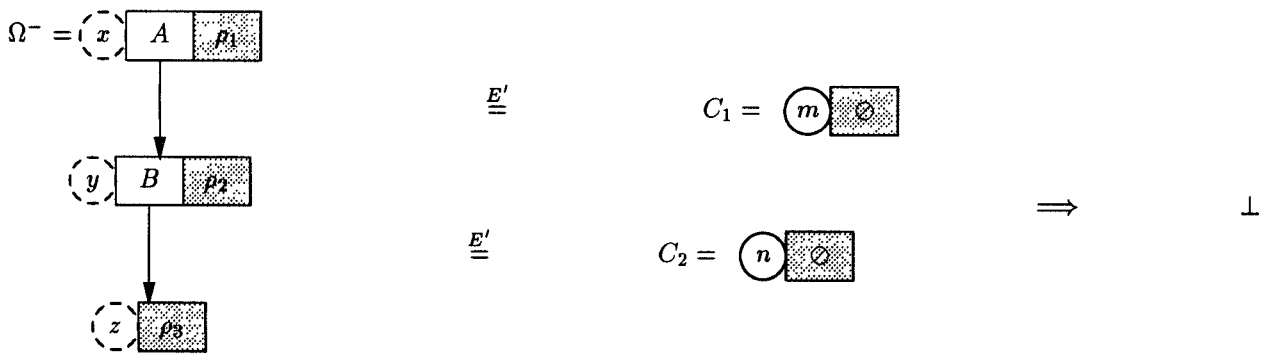


Figure 14: Unification failing on an uncovered signature

tion to the process of signature typing. In other words, if we know how to obtain consistent signatures, we can be sure they will be covered and well-formed.

**Lemma 8** *If the judgement  $\Gamma \vdash e \Rightarrow \sigma$  preserves consistency, then it also preserves well-formedness and covering.*

Proof. We sketch the proof. Let us suppose that  $\Gamma$  is consistent, well-formed and covered. Let us suppose also that  $\Gamma \vdash e \Rightarrow \sigma$  preserves consistency but that  $\sigma$  is badly-formed and uncovered. Then, there exist a structure  $\tau'$  in  $\Gamma$ , having on top a fixed stamp  $m$ , which is shared with a sub-structure  $\tau$  of  $\sigma$ . As  $\Gamma$  is well-formed and covered, it must be that  $\tau \not\stackrel{E'}{=} \tau'$ , but this contradicts the hypothesis on the consistency of  $\{\Gamma, \sigma\}$ . ■

From the previous sections there exists a signature typing algorithm finding principal signatures and preserving consistency. By the lemma above, this algorithm also preserves well-formedness and covering. Moreover, we achieve this result for free: without the introduction of any special condition checking for defective signatures. The following theorem arises trivially. It synthesizes the typing results presented in this paper.

**Theorem 3** *There exists a signature typing algorithm finding principal signatures and preserving consistency, well-formedness, and covering.*

## 8 Related work

In his Ph.D. thesis [11], Tofte studied a different semantics for consistency, matching, and sharing for SML modules. According to this semantics, consistency (which he called coherence) is stronger: two structures sharing the same stamp must be completely equal. Sharing constraints become equality constraints and matching is not coercive: components in a structure cannot be hidden as this would destroy consistency of multiple occurrences of the same structure. As in the language studied in this paper, sharing constraints can be specified within differently shaped structures. In

the same way, matching is allowed between structures richer than signatures. The matching definition and the unification algorithm in [11] are reminiscent of Rémy's algorithm in that both methods extend sequences of components by new components.

More recently, Milner and Tofte studied [6] the semantics we consider in this paper, and which happens to be the current semantics of SML modules. According to them, sharing is solved by a process called "admissification" that identifies stamps while checking consistency, cycle-freeness, and well-formedness on the whole context of current structures.

Our work uses several ideas already present in Tofte's thesis and in Milner and Tofte semantics: fixed stamps in the types of structures, bound stamps in the signature types, and structure extension during unification, but our presentation is strongly based on the Rémy's unification framework. To this approach we incorporate the encoding and the equational theories of unification necessary to deal with the current semantics of SML. All of that result in several important advantages: nice notions and proofs on principality, local consistency checking, consistency preservation and well-formedness and covering preservation coming directly from our signature typing techniques.

## 9 Conclusions

We have shown a simple and elegant formalization of the static semantics of SML modules. We obtain a type system relying only on well-known unification techniques and which does not need new concepts nor external constraints to guarantee that only legal types are inferred. The unification necessary to ensure consistency remains local to specified structures in sharing constraints, and this is naturally efficient. Last but not least, the proof on principality, relying on the principality result of sharing unification, is extremely simple.

Our type system is an interesting application of record type disciplines. Its extension to higher-order functors (see [12]) is still under study. There are two

main problems with this extension.

The first one concerns the extension of the  $\triangleright$  relation to be contravariant in order to compare functor types in structures during matching. The principal problem with contravariance is ensuring the existence of principal solutions.

The second problem concerns the semantics of functor sharing. In the type discipline of Tofte [12], two functor signatures can be shared if they are identical. In our framework, two functors can be shared if they differ only in some flags of components in the argument or in the result. It is not clear yet for us which one of these two solutions corresponds to the more “natural” functor sharing semantics.

## 10 Acknowledgements

Many thanks to Xavier Leroy, Mads Tofte, Didier Rémy, Michel Mauny, Benjamin Pierce and Ian Jacobs for their comments.

## References

- [1] María Virginia Aponte. *Typage d'un système de modules paramétriques avec partage: une application de l'unification dans les théories équationnelles*. Thèse de doctorat, Université de Paris 7, 1992.
- [2] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT'89*, 1989.
- [3] Robert Harper, Robin Milner, and Mads Tofte. A type discipline for program modules. In *Theory and Practice of Programming Languages*, volume 250 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.
- [4] Robert Harper and John C. Mitchell. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.
- [5] David MacQueen. Modules for standard ML. *Polymorphism Newsletter*, II, 1985.
- [6] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.
- [7] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [8] Didier Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.

- [9] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objects Enregistrements dans les Langages Fonctionnels*. Thèse de doctorat, Université de Paris 7, 1990.
- [10] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, Inria, Rocquencourt, May 1991. Also in [Rem90], chapter 4.
- [11] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1987.
- [12] Mads Tofte. Principal signatures for higher-order program modules. In *19th ACM Symposium on Principles of Programming Languages*, 1992.
- [13] Mitchell Wand. Complete type inference for simple objects. In *Second Symposium on Logic In Computer Science*, 1987.

## A Appendix: The inequation decomposition algorithm

The inequation decomposition algorithm  $\mathcal{S}i$  in figure 15 outputs a set of elementary equations and inequations. The Remy's equation decomposition algorithm  $\mathcal{S}e$ , simplifies an equation into a set of elementary equations. This algorithm can be found in [9]. Inequations in the output of the  $\mathcal{S}i$  algorithm are only of the forms:  $\alpha \triangleright abs$ ,  $pre \triangleright \alpha$  and  $\alpha \triangleright \beta$ , and there are not two equations or inequations on the same variable. The algorithm **elem**, given an elementary inequation where the variable  $\alpha$  appears, checks for the occurrence of an inequation or an equation on  $\alpha$  in the already simplified system  $\mathcal{M}$ . This algorithm tries to simplify all the inequations and equations on  $\alpha$  or to find a contradiction between them. For more details on this kind of simplification and on the whole inequation resolution process, the reader can consult [1].

In order to find the solutions of an inequation after the obtention of a simplified system, two more steps must be performed. First, one must check that there are no cycles in the system. Second, one must choose a solution for the elementary inequations having several solutions. In our type system we always choose  $\triangleright$ -minimal solutions. Thus, the  $\triangleright$ -minimal solution of  $\alpha \triangleright abs$  and of  $pre \triangleright \alpha$  is  $\alpha \stackrel{E}{=} abs$ .

## B Appendix: Inference rules

The typing relation is of the form  $(\Gamma \vdash e \Rightarrow \tau)$ , where  $\Gamma$  is a module context,  $e$  is a ModL expression, and  $\tau$  is

```

let rec Si(M, e) =
match e
with abs ▷ pre → ⊥
| pre ▷ abs → M
| α ▷ α → M
| C(σi) ▷ C(τi) →
  Si(M, σi ▷ τi)
| σ ▷ C(σi) →
  let αi = newvars() in
  let M' = Se(M, σ  $\stackrel{E}{\equiv}$  C(αi)) in
  Si(M', αi ▷ σi)
| (* symmetric *)
| c1 ▷ c2 →
  if c1 ≠ c2 then fail else M
| i → elem(M, i)

where rec elem(M, i) =
match i
with α ▷ β → M ∪ {α ▷ β}
| α ▷ c →
  if α  $\stackrel{E}{\equiv}$  c' ∈ M
  and c' ≠ c then fail
  else M ∪ {α  $\stackrel{E}{\equiv}$  c}
| abs ▷ α →
  if α  $\stackrel{E}{\equiv}$  pre ∈ M
  or α ▷ pre ∈ M then fail
  else
  (M \ {α ▷ abs, pre ▷ α}) ∪ {α  $\stackrel{E}{\equiv}$  abs}
| α ▷ pre →
  if α  $\stackrel{E}{\equiv}$  abs ∈ M
  or abs ▷ α ∈ M then fail
  else
  (M \ {α ▷ abs, pre ▷ α}) ∪ {α  $\stackrel{E}{\equiv}$  pre}
| α ▷ abs →
  if α  $\stackrel{E}{\equiv}$  abs ∈ M
  or α  $\stackrel{E}{\equiv}$  pre ∈ M then M
  else M ∪ {α ▷ abs}
| pre ▷ α →
  if α  $\stackrel{E}{\equiv}$  abs ∈ M
  or α  $\stackrel{E}{\equiv}$  pre ∈ M then M
  else M ∪ {pre ▷ α}
;;

```

Figure 15: The inequation decomposition algorithm

either a module type or a module context. It is defined as the smallest relation satisfying the rules below.

When  $A$  is a context component, we write  $A[x : t]$  for the function that is equal to  $A$  everywhere except on  $x$ , and assigns  $t$  to  $x$ . When the syntactic class of  $x$  is clear, we write  $\Gamma[x : t]$  instead of recomposing the whole context  $(\dots A[x : t] \dots)$  from the components of  $\Gamma$ . Let  $A$  and  $B$  be two context components of the same kind. We write  $A+B$  the context component defined by  $A(x)$  if  $B(x)$  is not defined and  $B(x)$  otherwise. Two module contexts can be added by adding their components. We assume for any type  $\forall W \cdot \tau$  and any context  $\Gamma$ , that  $W$  is disjoint from the variables in  $\Gamma$ . We also assume that  $W$  is contained in  $\mathcal{V}(\tau)$ .

Let  $\phi$  be  $\sigma \rightarrow \sigma'$  and call  $R$  the set  $W \setminus \mathcal{V}(\sigma)$ . The predicate  $Gen(\mu, \Gamma, \forall W \cdot \phi)$  holds if and only if  $I(\mu \upharpoonright R) \cap \mathcal{V}(\Gamma) = \emptyset$  where  $I(\nu)$  is the image of the substitution  $\nu$  and  $\upharpoonright$  the restriction of substitutions.

Given an assertion list  $\varsigma$  equal to  $[a_1 : \sigma_1; \dots; a_n : \sigma_n]$ , and a row term  $\rho$ , the notation  $\varsigma; \rho$  refers to the term  $a_1 : pre \cdot \sigma_1; \dots; a_n : pre \cdot \sigma_n; \rho$ .

$$\begin{array}{c}
(Vstr) \frac{\Gamma(strid) = \sigma}{\Gamma \vdash strid \Rightarrow \sigma} \\
\\
(Vsig) \frac{\Gamma(sigid) = \forall W \cdot \sigma}{\mu : W \rightarrow T}{\Gamma \vdash sigid \Rightarrow \mu\sigma} \\
\\
(Vfun) \frac{\Gamma(funid) = \forall W \cdot \phi}{\mu : W \rightarrow T \quad Gen(\mu, \Gamma, \phi)}{\Gamma \vdash funid \Rightarrow \mu\phi} \\
\\
(Res) \frac{\Gamma \vdash strexp \Rightarrow \sigma \quad \Gamma \vdash sigid \Rightarrow \sigma' \quad \sigma \triangleright \sigma'}{\Gamma \vdash strexp : sigid \Rightarrow \sigma'} \\
\\
(Extr) \frac{\Gamma \vdash strexp \Rightarrow Str(x, strid : pre \cdot \sigma; \rho)}{\Gamma \vdash strexp \cdot strid \Rightarrow \sigma} \\
\\
(Nstr) \frac{\Gamma \vdash dec \Rightarrow \varsigma \quad x \notin \mathcal{V}(\Gamma)}{\Gamma \vdash str \ dec \ end \Rightarrow Str(x, \varsigma; abs \cdot \circlearrowleft)} \\
\\
(Nsig) \frac{\Gamma \vdash spec \Rightarrow \sigma}{\Gamma \vdash sig \ spec \ end \Rightarrow Str(x, \varsigma; abs \cdot \alpha)} \\
\\
(App) \frac{\Gamma \vdash strexp \Rightarrow \sigma \quad \sigma \triangleright \sigma'}{\Gamma \vdash funid \Rightarrow \sigma' \rightarrow \tau}{\Gamma \vdash funid(strexp) \Rightarrow \tau}
\end{array}$$

$$(Abs) \frac{\Gamma \vdash sigid \Rightarrow \sigma \quad \sigma \text{ is principal for } sigid \text{ in } \Gamma \quad \Gamma[stid : \sigma] \vdash strexp \Rightarrow \tau \quad W \cap \mathcal{V}(\Gamma) = \emptyset}{\Gamma \vdash func (stid : sigid)strex \Rightarrow \forall W \cdot \sigma \rightarrow \tau}$$

$$(Spec) \frac{\Gamma \vdash spec \Rightarrow \varsigma \quad \Gamma[\varsigma] \vdash sigexp \Rightarrow \sigma \quad stid \notin dom(\varsigma)}{\Gamma \vdash spec; structure stid : sigexp \Rightarrow (stid : \sigma) :: \varsigma}$$

$$(Edec, Espec) \frac{}{\Gamma \vdash \{\} \Rightarrow \square}$$

$$(Sharing) \frac{\Gamma \vdash spec \Rightarrow \varsigma \quad \Gamma[\varsigma] \vdash longid_1 \Rightarrow \tau \quad \Gamma[\varsigma] \vdash longid_2 \Rightarrow \tau' \quad \tau \stackrel{E'}{=} \tau'}{\Gamma \vdash spec; sharing longid_1 = longid_2 \Rightarrow \varsigma}$$

$$(Equal) \frac{\Gamma \vdash e \Rightarrow \sigma \quad \sigma \stackrel{E}{=} \tau}{\Gamma \vdash e \Rightarrow \tau}$$

$$(Dec) \frac{\Gamma \vdash dec \Rightarrow \varsigma \quad \Gamma[\varsigma] \vdash strexp \Rightarrow \sigma \quad stid \notin dom(\varsigma)}{\Gamma \vdash dec; structure stid = strexp \Rightarrow (stid : \sigma) :: \varsigma}$$

$$(DecStr) \frac{\Gamma \vdash strexp \Rightarrow \sigma}{\Gamma \vdash structure stid = strexp \Rightarrow ([stid : \sigma], \square, \square)}$$

$$(DecSig) \frac{\Gamma \vdash sigexp \Rightarrow \sigma \quad W \cap \mathcal{V}(\Gamma) = \emptyset \quad \sigma \text{ is principal for } sigexp \text{ in } \Gamma}{\Gamma \vdash signature sigid = sigexp \Rightarrow (\square, [sigid : \forall W \cdot \sigma], \square)}$$

$$(DecFun) \frac{\Gamma \vdash funexp \Rightarrow \phi}{\Gamma \vdash functor funid = funexp \Rightarrow (\square, \square, [funid : \phi])}$$

$$(SeqProg) \frac{\Gamma \vdash program_1 \Rightarrow \Gamma_1 \quad \Gamma + \Gamma_1 \vdash program_2 \Rightarrow \Gamma_2}{\Gamma \vdash program_1 \text{ in } program_2 \Rightarrow \Gamma_1 + \Gamma_2}$$

## C Appendix: The signature typing algorithm

The following algorithm uses a set  $M$  of fixed stamps which cannot be modified (i.e., they cannot be in the domain of any substitution). It also uses Rémy's unification algorithm modulo the equations  $E$ . We call  $\mathcal{G}(M, \sigma \stackrel{E}{=} \tau)$  the algorithm finding the principal solution of the unification problem  $\sigma \stackrel{E}{=} \tau$  without changing the stamps in  $M$ . The algorithm  $\mathcal{G}^\pm(M, \sigma \stackrel{E'}{=} \tau)$  finds the principal solution modulo  $\triangleright$  of the sharing equation  $\sigma \stackrel{E'}{=} \tau$  without changing the stamps in  $M$ .

```

let  $\mathcal{W}(M, \Gamma \vdash sigexp \stackrel{?}{\Rightarrow} \alpha^*) =$ 
match sigexp
with sig spec end  $\rightarrow$ 
  let  $x = newvarstamp()$ 
  and  $\alpha, \beta = newvars()$ 
  and  $\mu = \mathcal{W}(M, \Gamma \vdash spec \stackrel{?}{\Rightarrow} \alpha)$ 
  in  $\mathcal{G}(M, \alpha^* \stackrel{E}{=} Str(x, \mu \alpha; abs \cdot \beta))$ 

|  $\{\}$   $\rightarrow \mathcal{G}(M, \alpha^* \stackrel{E}{=} \square)$ 

| sigid  $\rightarrow$ 
  let  $\forall W \cdot \tau = \Gamma(sigid)$  in
  let  $\mu : W \rightarrow T$  such that
     $I(\mu) \cap \mathcal{V}(\Gamma) = \emptyset$ 
  in  $\mathcal{G}(M, \alpha^* \stackrel{E}{=} \mu \tau)$ 

| spec; structure a : sigexp  $\rightarrow$ 
  let  $\alpha, \beta = newvars()$  in
  let  $\mu = \mathcal{W}(\Gamma \vdash spec \stackrel{?}{\Rightarrow} \alpha)$ 
  and  $\nu =$ 
     $\mathcal{W}(M, \mu \Gamma[\mu \alpha] \vdash sigexp \stackrel{?}{\Rightarrow} \beta)$ 
  in let  $\eta = \nu \circ \mu$ 
  in  $\mathcal{G}(M, \alpha^* \stackrel{E}{=} (a : \nu \beta) :: \eta \alpha)$ 

| spec; sharing longid1 = longid2  $\rightarrow$ 
  let  $\alpha, \beta, \gamma = newvars()$  in
  let  $\mu = \mathcal{W}(M, \Gamma \vdash spec \stackrel{?}{\Rightarrow} \gamma)$ 
  and  $\nu =$ 
     $\mathcal{W}(M, \mu \Gamma[\mu \gamma] \vdash longid_1 \stackrel{?}{\Rightarrow} \alpha)$ 
  and  $\nu' =$ 
     $\mathcal{W}(M, \mu \Gamma[\mu \gamma] \vdash longid_2 \stackrel{?}{\Rightarrow} \beta)$ 
  and  $\mu' = \mathcal{G}^\pm(M, \nu \alpha \stackrel{E'}{=} \nu' \beta)$ 
  in  $\mathcal{G}(M, \alpha^* \stackrel{E}{=} (\mu' \circ \mu) \gamma)$ 
;;

```

Figure 16: The signature typing algorithm