Module Programming using Overloading and Late Binding

María-Virginia Aponte* Giuseppe Castagna†

February 8, 1999

Abstract

One of the criticisms moved to the SML module system is that it does not allow code reusing and, more generally, and incremental style of programming. In this work we propose to extend this module system by adding overloading and late binding, and we show how by this extension it is possible to program modules in an incremental style, similar to the one of object-oriented languages.

1 Introduction

SML module systemsL [Mac85, MTH90] are very general and powerful. The allow modular decomposition of pragrams and the definition of transformations (called *functors*) for the modules (called *structures*). However they offer a very limited code reusing and incremental programming. More precisely, the SML modules do not possess the characteristic of code inheritance and reusing of object-oriented languages. This lack is sensible in program evolution.

If at a given moment of the program's life we decide to add it new functionalities by declaring new structures that specialize the existing ones, the use of old functors for these new structures is allowed. On the contrary it is not possible to refine the behavior of these old functors by adding to them new functionalities. Therefore either the new structures use the functors defined for the old one or they use new functors of their own that will be not correlated with the old ones. In such a case the possibility of evolution of the existing programs is seriously limited the

,,,

An important distinction has been extensively used in language theory for the last two decades, between *parametric* (or universal) polymorphism and *ad hoc* polymorphism: [Str67]. Parametric polymorphism allows one to write a function whose code can work on different types, while by using *ad hoc* polymorphism, also known as *overloading*, it is possible to write a function which executes a different code for each type.

Traditional languages offer a very limited form of overloading: the actual meaning of an overloaded operator is always decided at compile time, according its definition and the type of its arguments.

In languages that use subtyping the dynamic type of an expression does not generally coincide with its static type. In that case the selection of the code to associate to an overloaded function may be different for different calls. Thus, it is possible to differentiate at least two disciplines to select code for overloaded functions:

*CEDRIC Conservatoire National des Arts et Métiers - 292, rue St. Martin - 75003 Paris France. e-mail:aponte@cnam.cnam.fr

†CNRS. DMI-LIENS. Ecole Normale Supérieure - 45, Rue d'Ulm - 75005 Paris France. e-mail:castagna@dmi.ens.fr

- The selection is based on the static type of the arguments: the least type information is used. We call this discipline *early binding*.
- The selection is based on the dynamic type of the arguments: the maximal type information is used. We call this discipline *late binding*.

The use of overloading with early binding does not significantly affect the underlying language. On the contrary, the ability to define new overloaded functions when combined with subtyping and late binding can add expressive power in programming, namely, by increasing code reusability and giving raise to an incremental style of programming typical of object-oriented languages. We illustrate how to achieve this style of programming with an example.

Suppose that our programming language has overloading and subtyping, and consider two programs M_A and M_B. Each program has a parameter x whose type is A for M_A and B for M_B, where B is a subtype of A (denoted $B \leq A$). The two programs are identical except for a variant part that handle the argument x in a different way. Call the variant part P_A for M_A and P_B for M_B. This situation is illustrated in figure 1.

Using overloading it is possible to rewrite these two programs into a unique program M which reuses the common part of M_A and M_B and calls an overloaded subprogram P, which executes either P_A or P_B according to the type of its argument. Without loss of generality, we suppose that P depends on the parameter of M. Note that, as $B \leq A$, then $M(x : A)$ accepts an argument of both types A and B.

In order to state the precise relationship between programs in figure 1 and 2 we use contexts of λ-calculus. The program M of figure 2 is then equivalent to:

$$M = \lambda x\colon A.\mathcal{C}[P(x)] \tag{1}$$

where the context $\mathcal{C}[\ ]$ corresponds to the code shared between M_A and M_B, and P is an overloaded function with two *branches*[1] P_A and P_B. Now, M_A and M_B are equivalent to:

$$\begin{array}{rcl} M_A & = & \lambda x\colon A.\mathcal{C}[P_A(x)] \\ M_B & = & \lambda x\colon B.\mathcal{C}[P_B(x)] \end{array} \tag{2}$$

The rewriting of M_A and M_B into M ensures reusability of their common parts and properly reflects the original program structure. Nevertheless, this transformation does achieve the correct behavior only if code selection is performed with the late binding discipline.

Consider the definition for M given by (1). Since $x\colon A$, a call to M always execute the branch P_A of P if early binding is used. In other words, with early binding the definition given in (1) for M becomes equivalent to

$$\lambda x\colon A.\mathcal{C}[P_A(x)],$$

(i.e, to M_A), even when the actual parameter of M is a subtype of B. Thus, with early binding, the overloading schema of figure 2 does not reflect at all the behavior of the original program.

[1]We consider that an overloaded function is formed by all the different codes associated to it, and we call *branch* every distinct piece of code composing it.

Figure 1: Two programs with a shared part

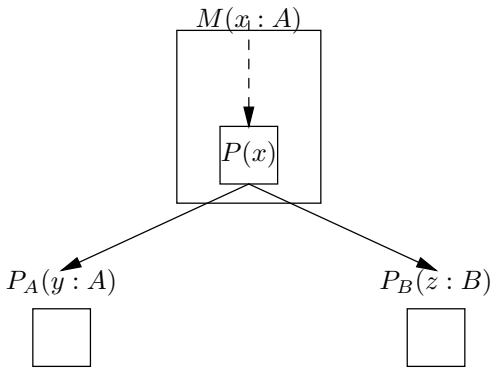

Figure 2: Overloading and reusability

Indeed, with early binding, the only way to execute different branches of P within M is to define M itself as an overloaded function of two branches M_A and M_B (as defined by (2)), thus going back to the original program scheme, and losing then reusability.

On the contrary, if late binding is used, the code to execute for P is chosen only when the formal parameter x has been substituted by the actual parameter. Thus, with late binding the definition of M in (1) is equivalent to M_A when M is applied to arguments of type A and to M_B when it is applied to arguments that are subtype of B. Actually, with late binding the function M in (1) is *implicitly* an overloaded function with two branches; thanks to late binding these virtual branches share the code $\mathcal{C}[\,]$.

In [CGL95] one of the authors has studied the integration of overloading and late binding in functional languages in order to obtain a typed model of object-oriented languages. In this paper we show how to integrate these mechanisms to a module language near to SML modules [Mac85, MTH90]. The module language we use has been proposed by X. Leroy in [Ler94] and it is very close in syntax and power to SML modules. These modules, that we call *manifest modules*, have a simpler semantics than SML modules, specially for the crucial constructions to define higher-order functors and to express sharing. Thus, they better fit the study of semantic extensions as the one we describe in this work. Also, we present in this paper a new style of module programming, based on incremental definition and code reusing but still enjoying of the generality of SML modules and, under some suitable conditions, of static verification of signature matching.

2 The module language

Building programs in a modular way is crucial in order to the develop large programs. A well-known modular discipline is used in SML, where modules are handled by a small typed language inside ML. In this sub-language, the base construction are modules (*structures*), which are composed from collections of declarations (of types, values and other modules). The type of modules are the module specifications (*signatures*), and modules can be built from other modules via functions from modules to modules (*functors*). Modules are then combined using functor application.

2.1 SML modules

We show below some examples of structures, signatures and functors in SML modules. The following signatures describe trees and dictionaries structures.

```
signature Item =
 sig
  type item
  val isequal: item *  item -> bool
 end
signature Tree =
 sig
  type 'a tree
  val empty: 'a tree
  val isnull: 'a tree -> bool
  val cons: 'a * 'a tree * 'a tree -> 'a tree
  val root: 'a tree -> 'a
  val left: 'a tree -> 'a tree
  val right: 'a tree -> 'a tree
 end
signature Dict =
 sig
  type key
  type 'a dict
  val empty : 'a dict
  val isnull: 'a dict -> bool
  val find: key * 'a dict -> 'a
  val insert : key * 'a * 'a dict -> 'a dict
 end
```

The following structure has signature `Item`:

```
structure IntItem : Item =
 struct
  type item = int
  fun isequal (a,b) = (a = b)
 end
```

Trees can be used to build dictionaries. The functor `MkDict` below builds a dictionary from a tree structure.

```
exception Notfound
functor MkDict(t: Tree): Dict =
 struct
  type key = t.i.item
  type 'a dict = (key * 'a) t.tree
  val empty = t.empty
  val isnull = t.isnull
  fun find (k,d) = if isnull(d) then raise Notfound
     else let (k',a) = t.root(d) in
          if t.i.isequal(k,k') then a
                 else ... (* Recursive breadth-first search *)
  fun insert (k,a,d) = if isnull(d) then t.cons((k,a),empty, empty)
                       else (* Recursive search of a free position *)
 end
```

2.1.1 Subtyping

The type discipline of SML allows a form of structural subtyping via the notion of *signature matching*. A structure s *matches* a signature S, if s contains at least all the components specified

by S; also, the common components to s and S must have compatible types, but types on s can be more polymorphic. Following, we declare a signature `OrdItem` for items with an order and a signature `OrdTree` for ordered trees. The structure `IntOrder` below matches not only signature `OrdItem` but also the signature `Item` of items without any order.

```
signature OrdItem =
 sig
  type item
  val isequal: item * item -> bool
  val isless: item * item -> bool
 end
signature OrdTree =
 sig
  structure i: OrdItem
  type 'a tree
  val empty: 'a tree
  val isnull: 'a tree -> bool
  val insert: 'a * 'a tree -> 'a tree
  val remove: 'a * 'a tree -> 'a tree
  val root: 'a tree -> i.item
  val left: 'a tree -> 'a tree
  val rigth: 'a tree -> 'a tree
  val max: 'a tree -> 'a
  val min: 'a tree -> 'a
 end
structure IntOrd =
 struct
  type item = int
  val isequal = op =
  val isless = op <
 end
```

Signature matching applies recursively on structures. Thus, any structure implementing the signature `OrdTree` matches also the signature `Tree`. On the other hand, the type rules for functor application requires matching between the actual parameter and the formal parameter signature. Thus, subtyping can be used to apply functor `MkDict` on a structure matching `OrdTree`, which is a subtype of the formal parameter signature of `MkDict`.

2.2 Manifest modules

The semantics of SML modules is very complex, particularly in the the case of higher-order functors and sharing. In this work, we choose to use *manifest modules*, a variant from SML modules proposed in [Ler94] which have simpler semantics and the same expressive power of SML modules.

Manifest modules have the same constructions as SML modules, but type constrains in signatures, such as abstraction, transparency and sharing, are specified via a unique sort of type specifications: the manifest type specifications. Manifest type specifications appear in signatures as annotations on type constructors. An explicit type annotations σ on type constructor `t` (of the form `type t =` σ) requires `t` to be implemented as type σ, and therefore makes `t` compatible with σ. Absence of annotations on a specified type constructor (of the form `type t`) make type constructor `t` abstract, and annotations on two type constructors are used to state sharing constraints. The structure

```
structure intOrder =
```

```
 struct
  type t = int
  fun cmp i1 i2 = i1 < i2
 end
```

has signature

```
intOrder : sig
             type t = int
             fun cmp : t -> t -> bool
           end
```

where the manifest type annotation for `t` states that `intOrder.t` is compatible with `int`. On the other hand, type constructor in signatures can be specified as abstract. The specification of type `t` in the following signature is abstract: if this signature is used during signature matching within a structure `s`, the corresponding type `s.t` will be consider abstractly during the typing.

```
signature Intlist =
 sig
  type t
  val nil : t
  val cons : int -> t -> t
 end
```

Furthermore, the type system of manifest is totally syntactical, that is, there is no need to introduce a special syntax for types: types are given by the module language syntax.

2.3 Manifest modules with overloading and late binding

In this section we consider having manifest modules together with an overloading mechanism on functors, for which code selection is performed using late binding. We illustrate the use of such a language by some examples.

2.3.1 Overloading and modular programming

Consider the functor `Use` which builds and uses a dictionary. The dictionary is built from a tree structure and its elements are taken from a priority queue. The functor takes as arguments the tree and priority queue structures[2].

```
functor Use (p: PQueue, t: Tree with t.i.item =  p.item) =
 struct
   (* Manipulations on q *)
   structure dict = MkDict(t)
       ....
   fun solve(q,d) =
     val fkey = p.front q
     val el = dict.find(fkey,d) in
     ....
 end
```

The first version of this functor uses dictionaries built on top of simple trees which make the search operation quite inefficient. In a further version one can use ordered tree to improve searching. The following functor builds dictionaries on top of ordered trees:

[2]We do not define the signature `PQueue`.

```
functor mkODict(t:OrdTree): Dict =
 struct
  type key = t.i.item
  type 'a dict = (key * 'a) t.tree
  val empty = t.empty
  val isnull = t.isnull
  fun find (k,d) = if isnull(d) then raise Notfound
     else let (k',a) = root(d) in if t.i.isequal(k,k') then a
     else if t.i.isless(k,k') then find (k,left(d))
     else find (k,right(d))
  fun insert (k,a,d) = if t.isnull(d) then t.cons((k,a),empty, empty)
                       else (* Ordered search of a free position *)
 end
```

Now, we want when and ordered tree is passed to `Use`, the corresponding dictionary is built using the functor `mkODict`. In the actual definition of SML modules, the only way to achieve this behavior is to define a new functor[3], say, `OUse` replacing `Use` for arguments matching `OrdTree` and from which it will differ only in the line `structure dict = MkODIct(t)`. Of course this has two drawbacks: it causes code duplication and it leaves to the programmer the task of choosing when to use `Use` or `OUse`.

An easy solution which avoids rewriting the program and delegate the choice of the code to the system is to overload the functor `MkDict` used within the body of `Use` by adding to its definition the functor `mkODict` given above. The syntax we propose for this overloading operation is:

```
functor mkDict = mkDict addfunctor mkODict
```

which adds to the definition of `mkDict` given in the previous section, the definition of `mkODict`, both definitions forming by now an overloaded functor with two branches, called `mkDict`. We use late binding to select the branch to execute while applying an overloaded functor. Then, if we apply `Use` to a tree without order, the first version of `MkDict` will be used as before. But, if `Use` is applied to a tree matching `OrdTree`, thanks to late binding, the branch of `Use` corresponding to `MkODict` will be executed and the dictionary build will be more efficient on searching.

This example faithfully reflects the situation given in the introduction: `Use` is M, `Tree` is A, `OrdTree` is B, `MkDict` in the previous section is P_A, `MkODict` is P_B and `MkDict` in the definition of `Use` is P.

2.3.2 Overloading and module's sharing

In manifest modules, sharing between types is specified by a manifest type equation. For instance, in a functor of the form

```
functor F(structure s1: sig type t;..end s
          structure s2: sig type t=s1.t;.. end)
```

the signature of structure `s2` contains a manifest type specification of type `t`, i.e, the specification `type t=s1.t` which can be only satisfied if the actual parameters `a` et `b` of `F` have equal types `a.t` and `b.t`. In manifest modules, the manifest type specification is the base construction to specify types in signatures. In particular, manifest types can be compared by subtyping with other type specifications, as showed by the subtyping rules in section (??). Thus, verifying sharing constraints while applying a functor is obtained for free while verifying signature

[3]Even if we consider SML with higher-order functors, a functor `Use` properly typed cannot be written, because of contravariance of signature matching.

matching by subtyping on signatures. As selection on overloaded functors uses also subtyping on signatures, the expression of sharing within overloaded functors does not need any particular extension of module typing, besides those showed in section (??) used to express overloading within functions.

3 A calculus with modules and overloaded functors

We define here the base calculus on modules with overloaded functors.

3.1 Syntax

As in [Ler94], we suppose given a base language which is left unspecified. In the following grammar τ stands for type expressions in the base language, and p stands for paths to access structure components. We use t for type identifiers and x for structure identifiers.

Structure expressions (ranged over by s) can be path expressions, structure definitions, functor applications or signature restrictions. Structures are collections of type declarations and structure declarations; we do not consider value declarations since they are not relevant in this context.

We limit our study to first order unary functors: the extension to multi argument functors is straightforward[4], while we leave higher-order functors to future work. Signatures (or *module types*) are either *structure signatures* (ranged over by S) or *functor signatures* (ranged over by F). Structure signatures contain abstract type specifications (of the form `type t`) or transparent type specifications (of the form `type t` $= \tau$), and other structure specifications. Functor signatures are dependent types (i.e, the parameter variable of a functor can appear in its result type) or unions of dependent types (signatures of overloaded functors).

Structures can be restricted in view by signature matching (with $s : S$) by hiding some declarations or by transforming other declarations (e.g. type bindings of the form `type t` $= \tau$, become abstract in the result signature if the restriction signature contains an abstract specification of the form `type t`)

Overloaded functors are built from a regular functor by appending new branches using the `and` construction[5].

Access paths:

p	$::= x$	structure identifier
	$\mid p.x$	access to a structure component

Structure expressions:

s	$::= p$	identifier and access to a field
	$\mid$ `struct` d `end`	structure construction
	$\mid f\ (s)$	functor application
	$\mid s : S$	restriction by a signature

Structure body:

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

Structure components

b	$::=$ `type` $t = \tau$	type definition
	$\mid$ `structure` $x = s$	structure definition

[4]Multi argument functors can be obtained by adding cartesian products of signatures

[5]The construction `addfunctor` we used in the examples before can be considered as syntactic sugar to deal with functor identifiers.

Functor expressions:

f	::= `functor` $(x : S)s$	functor definition
	$\mid f$ `and` $(x : S)s$	functor overloading

Type expressions:

τ	::= t	type identifier
	$\mid p.t$	access to a type component
	$\mid \ldots$	base language-dependent

Structure signatures:

S ::= `sig` D `end`

Signature body:

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

Signature components:

B	::= `type` t	abstract type specification
	$\mid$ `type` $t = \tau$	manifest type specification
	$\mid$ `structure` $x : S$	structure specification

Functor signatures

F	::= `functor` $(x : S)S'$	dependent functor signature
	$\mid F \cup$ `functor` $(x : S)S'$	overloaded functor signature

Note that overloaded functor signatures are of the form

$$\texttt{functor}(x_1 : S_1)S_1' \cup \texttt{functor}(x_2 : S_2)S_2' \cup \ldots \cup \texttt{functor}(x_n : S_n)S_n'$$

Intuitively this is the signature of an overloaded functor formed by n branches, the i-th branch being a regular functor of signature $\texttt{functor}(x_i : S_i)S_i'$. When an overloaded functor of this signature is applied to a structure that exactly matches S_j then the structure is passed to the $j-$th branch of the functor. When the signature of the argument does not exactly match the parameter signature of one of the branches the system selects among the branches whose parameter signature is matched by the argument, the one with the most specialized (i.e. least) parameter signature (this is formally stated by rules (7) and (8) of the dynamic semantics in Section 3.3). Thus, for example, if we apply the overloaded functor `mkDict` of Section 2.3 to an argument that matches both `Tree` and `OrdTree`, the branch defined for `OrdTree` is selected.

In the following we may abridge the signature above by $\bigcup_{i=1}^{n} \texttt{functor}(x_i : S_i)S_i'$.

Note also that signature restrictions (denoted by $s : S$) are first class values. The expression $s : S$ coerces the structure s to have exactly signature S, provided that s matches S; thus the expression $s : S$, which forces s to have the signature S, plays the same rôle as an *explicit coercion* [BL90] of functional languages with subtyping. This is a very important feature in a system like ours where the the computation is driven by signatures. Indeed, signature restrictions can be used to force the selection of a particular code for an overloaded application. In the example of Section 2.3, if `OT` is a structure that matches `OrdTree` then we know that `mkDict(OT)` will select the code defined for ordered trees. However, we can force the selection of the code for generic trees by using the following expression: `mkDict(OT:Tree)`. Similarly the application `Use(P , OT:Tree)` makes the functor `Use` to work with a generic dictionary even if we applied it an `OrdTree`.

3.2 Static semantics

3.2.1 Subtyping and type equivalence

We use E to denote typing environments, $BV(S)$ to denote the set of variables bound by a declaration, and $S\{x \leftarrow s\}$ (respectively $s'\{x \leftarrow s\}$) to denote the type (respectively, the

Modules subtyping:

$$\frac{\text{for } i \in \{1,\ldots,m\}, \qquad E; D_1; \ldots; D_n \vdash D_{\sigma(i)} <: D'_i}{E \vdash \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} <: \texttt{sig}\ D'_1; \ldots; D'_m\ \texttt{end}} \quad (*)$$

$$\frac{E \vdash S <: S'}{E \vdash (\texttt{structure}\ x : S) <: (\texttt{structure}\ x : S')}$$

$$E \vdash (\texttt{type}\ t = \tau) <: (\texttt{type}\ t)$$

$$E \vdash (\texttt{type}\ t) <: (\texttt{type}\ t)$$

$$\frac{E \vdash \tau \approx \tau'}{E \vdash (\texttt{type}\ t = \tau) <: (\texttt{type}\ t = \tau')}$$

$$\frac{E \vdash t \approx \tau}{E \vdash (\texttt{type}\ t) <: (\texttt{type}\ t = \tau)}$$

Type equivalence:

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

$$\frac{E \vdash p : \texttt{sig}\ D_1; \texttt{type}\ t = \tau; D_2\ \texttt{end}}{E \vdash p.t \approx \tau\{n \leftarrow p.n \mid n \in BV(D_1)\}}$$

(*) σ is a mapping from $\{1,\ldots,m\}$ to $\{1,\ldots,n\}$

Figure 3: Subtyping and type equivalence

expression) obtained by substituting s for x in S (respectively, s').

The judgment $E \vdash S_1 <: S_2$ states that signature S_1 is a *subtype* (more properly *subsignature*) of signature S_2 in environment E. Judgment $E \vdash \tau \approx \tau'$ states that the type τ is *equivalent* to type τ' in environment E.

The subtyping relation is obtained from the deduction rules of Figure 3 plus the standard congruence, transitivity and symmetry rules for $\approx$ and the transitivity and reflexivity rules for $<:$.

These rules do not deserve any special comment since they coincide with the ones of [Ler94]. Just notice that we have not included subtyping rules neither for dependent nor for overloaded functor signatures, since we do not consider higher order functors.

3.2.2 Well-formed signatures

Once we have defined the subtyping relation on signatures we can select among them the signatures that are well formed. The definition of well-formedness is necessary to constrain overloaded functors to a given shape. Indeed, the use of late binding does not allow to form overloaded functors in a completely free way, otherwise the type system would not be sound.

A necessary condition for the safety of the system is that if a functor contains two branches defined for parameters of signature S and S', then whenever $S <: S'$ the result signatures of the branches must be in a covariant relation. More formally let S denote the overloaded functor signature $\bigcup_{i=1}^{n} \texttt{functor}(x_i{:}\, S_i)S'_i$. The signature S is *well formed* in the environment E (denoted by $WF_E(S)$) if and only if

$$\forall i, j \in \{1, \ldots, n\} \qquad (E \vdash S_i <: S_j \quad \Longrightarrow \quad E; \texttt{structure}\, x : S_i \vdash S'_i <: S'_j) \tag{3}$$

Intuitively the condition above (called the *covariance condition*) is necessary to assure that the signature of the structure obtained after execution of an expression is a subtype of the one statically inferred for the expression. For example suppose that f is an overloaded functor of signature $\texttt{functor}\ (x{:}\, S_1)S'_1 \cup \texttt{functor}\ (x{:}\, S_2)S'_2$ and that $S_2 <: S_1$. Let s be a structure expression that exactly matches S_1 and apply f to it. Then, statically, we deduce that the first branch of f will be selected and we infer for $f(s)$ the signature S'_1. [6] But, since $S_2 <: S_1$, it is possible that at execution time s reduces to a structure s' of signature S_2. This happens for example if s is the following expression: $(\texttt{functor}\ (x{:}\, S_1)x)(s')$. Therefore the branch selected for $f(s)$ at run-time will be the second one, and the resulting signature S'_2. Nevertheless, condition (3) guarantees that $S'_2 <: S'_1$, i.e. that the signature obtained after the execution is a subtype of the one deduced statically.

A broader discussion and more formal justification of the use of covariance for overloading and late binding can be found in [CGL95, Cas95]

3.2.3 Typing

Typing rules for structure expressions are showed in Figure 4. They use judgments of the form $E \vdash s : S$, stating that structure s has type S in the context E.

Worth to be noted are the fifth and sixth rules, that deal with overloaded functor construction and application. When a new branch is appended to an overloaded functor f, the system checks the signature of the new branch, appends this signature to the signature of f and checks that the resulting new signature is well formed. When an overloaded functor f is applied to an argument of signature S, the system selects among all the branches whose parameter signatures are matched by the argument (i.e. all those such that $E \vdash S <: S_i$), the branch whose parameter type S_j "best matches" the argument (i.e. such that $S_j = \min_{i \in I}\{S_i \mid E \vdash S <: S_i\}$).

Note also that, contrary to [Ler94], we do not use a *subsumption rule* to type expressions, but we rather use the algorithmic version of the system (in which subtyping is used directly inside the rules of elimination: functor and oveloaded functor application). This has the advantage that every typable expression has a unique signature, that is the one it exactly matches. This allows us to use this system also for the dynamic selection of the branches, while with subsumption, having each argument many signatures, the selection would not be deterministic.

The type rules use a strengthening operation noted S/p which enriches the structure type S with information about the complete path p identifying the abstract components of S. This operation is defined as follows:

$$\begin{aligned}
(\texttt{sig}\ D\ \texttt{end}) &= \texttt{sig}\ D/p\ \texttt{end} \\
(\texttt{type}\ t; D)/p &= \texttt{type}\ t = p.t; D/p \\
(\texttt{structure}\ x : S; D)/p &= \texttt{structure}\ x : S/p; D/p \\
S/p &= S \qquad \text{otherwise}
\end{aligned}$$

[6] More precisely $S'_1\{x \leftarrow s\}$, but for the sake of the example suppose that the result types are closed

3.3 Dynamic semantics

In this section we define how the structure expressions are "calculated" at link-time. In particular we describe how the late binding selection for overloaded functor is performed. To this end we define a call-by-value operational semantics for the closed (i.e. without free structure identifiers) structure expressions. Let first define what a *structure value* (or simply a *value*) is:

Structure values:

$$v ::= \texttt{struct}\ u\ \texttt{end} \mid (v : S)$$

Value components:

$$u ::= \varepsilon \mid \texttt{type}\ t = \tau; u \mid \texttt{structure}\ x = v; u$$

Thus a value is either a structure whose structure components are (structure) values, or its restriction. The operational semantics is then defined by the following rewriting rules

[Beta]

$$(\texttt{functor}(x : S)s)(v) \implies s\{x \leftarrow v\} \tag{4}$$

[Restriction]

$$(v : S).x \implies v.x \tag{5}$$

[Access]

$$(\texttt{struct}\ u_1; \texttt{structure}\ x = v; u_2\ \texttt{end}).x \implies s \tag{6}$$

[Overload]

Let $(f\ \texttt{and}\ (x : S_n)s) : \bigcup_{i=1}^{n} \texttt{functor}(x : S_i)S'_i$ and let $v : S$.

If $S_n = \min_{i=1..n}\{S_i \mid S <: S_i\}$ then

$$(f\ \texttt{and}\ (x : S_n)s)(v) \implies s\{x \leftarrow v\} \tag{7}$$

If $S_n \neq \min_{i=1..n}\{S_i \mid S <: S_i\}$ then

$$(f\ \texttt{and}\ (x : S_n)s)(v) \implies f(v) \tag{8}$$

[Context]

If $s \implies s'$ then

$$s.x \implies s'.x \tag{9}$$

$$f(s) \implies f(s') \tag{10}$$

$$(s : S) \implies (s' : S) \tag{11}$$

$$(\texttt{struct}\ u; \texttt{structure}\ x = s; d\ \texttt{end}) \implies (\texttt{struct}\ u; \texttt{structure}\ x = s'; d\ \texttt{end}) \tag{12}$$

Let us comment the rules above in detail. First of all note that they describe a deterministic operational semantics. Rule (4) implements the classical call-by-value discipline for functor application. Rule (5) erases the restriction from a structure when a component of this structure

is accessed. We could have used instead of (5) more sophisticated rules, say, to propagate the restriction to the components [7]. However, rule (5) is easier to understand and and does the least work needed. It just means that restrictions are useful only to hide some information of a structure and to sway the selection over a particular branch when the structure at issue is passed to an overloaded functor. But when a component of the structure is accessed then the restriction loose its utility and thus it can be erased. In other terms, this rule bounds the use of restrictions to alter the selection of a branch, only to the outer level of structures. Rule (6) performs the selection of a structure component. Rules (9), (10), (11) and (12) describe the rection inside a context.

The important rules are rules (7) and (8) which perform the late binding selection for overloaded functor application. More in detail. When an overloaded functor is applied to an argument the reduction take place only if the argument is a value (this implements the late binding). The first thing to do is to consider the type of the overloaded functor and of the argument. Note that the deductions of the types do not need any typing context since both the overloaded functor and its argument are closed expressions (see proposition 3.1). From the set of the parameter signatures of the overloaded functor we pick up those that are matched by the argument, i.e. all S_i such that $S <: S_i$. Again, for the same reason as before, typing environments are not needed. If the set of matched signatures has a least element, then we can perform the reduction: when the least signature coincide with the parameter signature of the rightmost branch, this branch is selected [rule (7)]; otherwise the search is continued over the remaining branches [rule (8)].

An interesting point of the rule (7) is that it allows the redefinition of a specific branch of an overloaded functor. Indeed nothing prevents from having several branches of an overloaded functor defined for the same signature. However, the peculiar definition of (7) makes that the rightmost of them is always selected. Practically speaking, this means that if an overloaded functor f has defined a branch for the signature S and we want to redefine this branch, we do not need to rewrite the whole f; it suffices to append to it the new definition of the branch: f `and` $(x : S)s$. This is a very interesting feature since the definition of the new branch f may be in a module different from the one containing the definition of f, that thus ought not be recompiled. Using the syntactic sugar of Section 2.3, we have for example that if `mk2ODict` is a functor with signature `functor(t:OrdTree):Dict` then the result of

```
functor mkDict = mkDict addfunctor mk2ODict
```

is the replacement of `mk2ODict` for the old branch defined by `mkODict`. More generally if `G` denotes a functor with signature `functor`$(x : S)S'$ then the construction

```
functor F = F addfunctor G
```

adds to the functor denoted by `F` a new branch for arguments of signature S, or if such a branch already existed in `F`, it redefines it. Note that in case of redefinition the type system (more precisely, the covariance condition) forces the type of the result of the new branch to be the same as the one of the old branch.

Proposition 3.1 *If* $s \Longrightarrow s'$ *then for any context* $\mathcal{C}[\ ]$ *such that* $\mathcal{C}[s] \Longrightarrow \mathcal{C}[s']$, *if* $C[s]$ *is a closed expression then also* s *is closed*

Proof. A straightforward induction on $\Longrightarrow$. Just notice that the body of functors are not reduced. □

[7] So that for example the restriction of the expression `struct structure` $x = s$; `type` t `end` to the signature `sig structure` $x : S$ `end` would rewrite to `struct structure` $x = (s : S)$ `end`.

Proposition 3.2 *If s is a closed structure and $s \Longrightarrow s'$ then also s' is closed.*

Proof. A trivial induction on the definition of $\Longrightarrow$. For the case of context reduction use Proposition 3.1. □

4 Open issues

Of course this system inherits the drawbacks of manifest modules. In particular it is not possible to establish the soundness of the type system in [Ler94] (and, thus, of our system) by proving preservation of typing under a (call-by-value) operational semantics. Indeed, the classical technique used to prove subject reduction does not work there because of the circularity of the definition of the type system (to prove subject reduction for expressions one needs a substitution lemma for subtyping, which implies substitution for type equivalence that needs expressions subject reduction). We believe that once this proof done for the system in [Ler94] it will be easy to use this result to prove the consistency of our type system. Much more unlikely, instead, is the use of semantic tools for our system: the only denotational models for such a kind of overloading that we are aware of cannot handle but early binding [Tsu92, CGL93, **?**].

Although important, the proof of soundness of the type system is not our main concern for the future. Indeed, other proposals of module systems lack of such a proof (e.g. [HL94]). We are much more interested at short term, in elaborating techniques to assure that the execution of every closed structure expression ends up with a value (i.e. either a structure or a signature restriction of a structure cfs. Section 3.3). This is actually not the case. This property fails because of the rewriting rule **[Overload]**. In fact, note that, in order to be applied, this rule requires the existence of a most specialized branch matched by the argument. But if such a branch does not exist (i.e. if the set $\{S_i \mid S <: S_i\}$ has not a least element) the computation is stuck and the execution ends with an application, which is not a value. To see how this may happen, consider the following expressions:

```
functor f = functor(x:sig  val a:int end) ...
              and (x:sig  val b:bool end) ... ;

structure s = struct
                 val a = 1;
                 val b = true
              end;
```

If the functor `f` is applied to `s` then the argument matches the parameter signatures of both branches, but there does not exist a most specialized branch since the two parameter signatures are incomparable. The static type system would reject the term `f(s)` since in this case it wouldn't be possible be able to choose one of the branches not even statically. However, in some cases it may happen that an expression can be statically typed but at run time it is not possible to perform the selection. This for example happens with the expression `f(Id(s))` where `Id` is defined as follows

```
functor Id = functor(x:sig  val a:int end) x
```

The expression `f(Id(s))` is statically typed as if the first branch of `f` were selected. But it is clear that at run-time, after one step of reduction, we would meet the same situation as the previous example.

There exist at least three type-theoretic solutions to assure that every execution of a closed structure expression will return a value.

The first solution is to impose in the well-formation of signatures that the parameter signatures of an overloaded functor must form a chain. This solution is simple but it would narrow

the field of application of overloading since the use of overloading would then be restricted to the specialization of already existing functors.

A second more articulated solution [CGL95] consists in restricting the definition of well-formed signature, by stating that a signature $\bigcup_{i=1}^{n} \texttt{functor}(x_i\colon S_i)S'_i$ is well formed if and only if it satisfies the covariance condition of Section 3.2.2 and

$$\forall i, j \in [1..n], \text{ if } S_i \text{ and } S_j \text{ have a common lower bound then } \exists h \in [1..n], S_h = S_i \cap S_j$$

(where $\cap$ denotes the greatest lower bound of two signatures). With this further condition (called *multiple inheritance condition*) the signature of the functor `f` in example above is not well formed since there is no branch whose parameter signature is the glb of the parameter signatures of the branches. With this condition the type system would force the programmer to add to `f` a further branch:

```
functor f = functor(x:sig  val a:int end) ...
               and (x:sig  val b:bool end) ...
               and (x:sig  val a: int; val b:bool end) ... ;
```

Though adapted to object-oriented programming, this rule would be infeasible with module systems. Indeed for an overloaded functor of n branches the system could require, in the worst case, the addition of $\frac{n(n-1)}{2}$ further branches. It is useless to say that hardly few of them would be interesting, since most of the branches would handle structures that the program will never use. Therefore the third solution is to ask the programmer to point out the structures signatures that he/she considers interesting. This can be done by introducing names for signatures and name subtyping, and by restricting parameter signatures of overloaded functors to named signatures.

In the example above the programmer could decide to name the following signatures

```
name A = sig  val a:int end;

name B = sig  val b:bool end ;

name C = sig  val a: int;
              val b:bool;
              val c:char
         end;
```

to declare that the name `C` is a subtype of `A`

```
subtype C of A
```

and he would then define the functor `f` as follows

```
functor f = functor(x:A) ...
               and (x:B) ...
```

Declaring that a name is a subtype of another name is feasible only if the corresponding signatures are in the same relation. Also, `C` is not a subtype of `B` since this has not been explicitly declared. This last observation implies that the problem of selection with `f` no longer happens. In fact even if `f` is applied to an argument of signature `C`,[8]. it matches `A` but not `B` Thus the first branch is selected. The condition to add to covariance to define well-formation is in this case [Cas95]:

$$\forall i, j \in [1..n], \text{ for every minimal signature of } LB(S_i, S_j), \exists h \in [1..n], S_h = D$$

[8]Elements whose signature is a name can be obtained, for exemple by modifying the typing rules for explicit restrictions

(where $LB(S_i, S_j)$ denotes the set of common lower bounds of S_i and S_j). Note that this condition is much less restrictive than the one before, and now it concerns only those signatures and subtypes that the programmer has indicated as interesting. Although this last solution is more flexible than the previous ones, it requires some extra work to the programmer, and looses the full generality of the overloading presented here.

What we have sketched above are three type theoretic solutions to the problem of the selection. However we believe that type theoretic techniques are not best fitted to handle such a problem. That is the reason why we did not use them in the presentation of the system and we just hinted them at the end of our work.

We believe that the definition of the system must be the one we gave in Sections 3.2 and 3.3. Thus our attitude is to consider that if the programmer has defined the functor `f` as

```
functor f = functor(x:sig  val a:int end) ...
              and (x:sig  val b:bool end) ... ;
```

then he/she meant that this functor will be never applied to an argument matching both the first and the second parameter signatures. Of course, it is necessary to provide to the programmer some tools that statically assure him that this effectively will be never happen. But to this end we believe that instead of using type system solutions —as the three hinted before— the answer is rather to be found in the use of data-flow analysis [**?**] or of more general techniques based on "abstract interpretation" [JN95]. Very briefly, a data flow analysis would consist in constructing the set of all types that an expression in a program may take at run time. We call this set the *dynamic set* of the expression. This set is constructed by a static analysis of the program through an iterative process: starting from the static type of each term it is possible to extend the current dynamic set of a given expression by repeatedly adding new types according to the flow of the program. We stop when we reach a fix-point. It is clear that if after this analysis no dynamic set of an argument of `f` contains a subtype of the parameter signatures of `f` then the above definition of `f` is type safe. Such a technique is already used in object-oriented programming; for example, Eiffel use it to verify that covariantly specialized methods do not cause the raising of a "message not understood" exception [Mey91]. A more general technique consists in defining an "abstract interpretation" of the data and to run the program on in. The interpretation is refined by an iterative process, which ends with a fix-point. This interpretation is defined so that if the program executed on the interpretation has some properties then so it has the program for any data.

We leave the study of these techniques to future work.

5 Conclusion

References

[BL90] K.B. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990. A preliminary version can be found in *3rd Ann. Symp. on Logic in Computer Science*, 1988.

[Cas95] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3), 1995.

[CGL93] G. Castagna, G. Ghelli, and G. Longo. A semantics for $\lambda\&$-*early*: a calculus with overloading and early binding. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in

Computer Science, pages 107–123, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995. A preliminary version has been presented at the ACM Conference on LISP and Functional Programming, San Francisco, June 1992.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Annual Symposium on Principles Of Programming Languages*, pages 123–137, Portland, Oregon, January 1994. ACM Press.

[JN95] N. D. Jones and F. Nielson. Abstract interpretation. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 527–636. Oxford Science Publication, 5 edition, 1995.

[KM89] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: a new perspective based on type inference. In *Proc. of Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, 1989.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, January 1994.

[Mac85] David MacQueen. Modules for standard ML. *Polymorphism Newsletter*, II, 1985.

[Mey91] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[Tsu92] Hideki Tsuiki. *A record calculus with a merge operator*. PhD thesis, Faculty of Environmental Information, Keio University, November 1992.

Typing of module expressions and definitions:

$$E_1; \texttt{structure}\ x : S; E_2 \vdash x : S$$

$$\frac{E \vdash p : \texttt{sig}\ D_1; \texttt{structure}\ x : S; D_2\ \texttt{end}}{E \vdash p.x : S\{n \leftarrow p.n \mid n \in BV(D_1)\}}$$

$$\frac{E;\ \texttt{structure}\ x : S \vdash s : S'}{E \vdash \texttt{functor}(x : S)s : \texttt{functor}(x : S)S'} \quad x \notin BV(E)$$

$$\frac{E \vdash f : \texttt{functor}(x : S')S \qquad E \vdash s : S'' \qquad E \vdash S'' <: S'}{E \vdash f(s) : S\{x \leftarrow s\}}$$

$$\frac{E \vdash f : \bigcup_{i=1}^{n-1} \texttt{functor}(x : S'_i)S_i \qquad E;\ \texttt{structure}\ x : S'_n \vdash s : S_n}{E \vdash f\ \texttt{and}\ (x{:}\, S_n)s : \bigcup_{i=1}^{n} \texttt{functor}(x{:}\, S'_i)S_i} \quad (*)$$

$$\frac{E \vdash f : \bigcup_{i=1}^{n} \texttt{functor}(x{:}\, S_i)S'_i \qquad E \vdash s : S}{E \vdash f(s) : S'_j\{x \leftarrow s\}} \quad (**)$$

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

$$\frac{E \vdash s : S' \qquad E \vdash S' <: S}{E \vdash (s : S) : S}$$

$$\frac{E \vdash d : D}{E \vdash \texttt{struct}\ d\ \texttt{end} : \texttt{sig}\ D\ \texttt{end}}$$

$$\frac{E;\ \texttt{type}\ t = \tau \vdash d : D}{E \vdash (\texttt{type}\ t = \tau; d) : (\texttt{type}\ t = \tau; D)} \quad t \notin BV(E)$$

$$\frac{E \vdash s : S \qquad E;\ \texttt{structure}\ x : S \vdash d : D}{E \vdash (\texttt{structure}\ x = s; d) : (\texttt{structure}\ x : S; D)} \quad x \notin BV(E)$$

(*) $x \notin BV(E)$ and $WF_E(\bigcup_{i=1}^{n} \texttt{functor}(x{:}\, S'_i)S_i)$

(**) $S_j = \min_{i=1..n}\{S_i \mid E \vdash S <: S_i\}$

Figure 4: Type rules