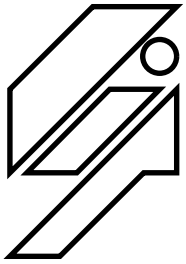
A module calculus enjoying the subject-reduction property. (Preliminary Version)

Judicael Courant



Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

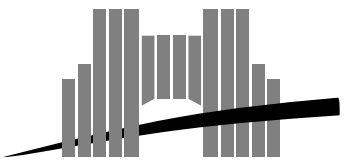
A module calculus enjoying the subject-reduction property

Preliminary version

Judicaël Courant

October 96

Research Report N^{O} 96-30



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

A module calculus enjoying the subject-reduction property

Preliminary version

Judicaël Courant

October 96

Abstract

The module system of SML is a small typed language of its own. As is, one would expect a proof of its soundness following from a proof of subject reduction, but none exists. As a consequence the theoretical study of reductions is difficult, and for instance, the question of normalization of the module calculus can not even be asked.

In this paper, we build a variant of the SML module system — inspired from recent works — which enjoys the subject reduction property. This was the initial motivation. Besides our system enjoys other type-theoretic properties: the obtained calculus is strongly normalizing, there are no syntactic restrictions on module paths, it enjoys a purely applicative semantic, every module has a principal type, and type inference is decidable. Moreover we conjecture that type abstraction — achieved through an explicit declaration of the signature of a module at its definition — is preserved.

Keywords: Module systems, subject-reduction, normalization, type inference, SML, lambda-calculus

Résumé

Le système de modules de SML est un vrai petit langage typé. Alors qu'on pourrait attendre une preuve de la consistance des systèmes de modules découlant de l'étude des réductions de modules, en particulier d'une preuve d'autoréduction, aucune preuve de ce genre ne semble exister. Or cela est un préliminaire nécessaire à toute étude des réductions, et en particulier à la question de la normalisation du système de modules.

Dans ce rapport, nous construisons une variante du système de modules de SML, inspirée de travaux récents, qui possède la propriété d'autoréduction. Cette motivation initiale conduit par ailleurs à un système de modules possédant des propriétés théoriques remarquables: le calcul de modules est fortement normalisant, aucune restriction syntaxique sur les chemins d'accès n'est nécessaire, la sémantique de notre système est purement applicative, tout module possède un type principal, et l'inférence de type est décidable. Nous conjecturons par ailleurs que l'abstraction de type — obtenue par un mécanisme de déclaration explicite de la signature d'un module lors de sa définition — est préservée.

Mots-clés: Systèmes de modules, autoréduction, normalisation, inférence de type, SML, lambda-calcul

A module calculus enjoying the subject-reduction property*

Preliminary version

Judicaël Courant

October 96

1 Introduction

Modularity is an essential technique for software programming and reuse. It is also needed for reasoning about programs: especially, it is a major issue of formal methods. In this respect, the SML language is particularly interesting, because of the power of its module language, which is a small typed language of its own [HMT87, HMT90]. This module system was designed for use at an interactive toplevel, and therefore separate compilation issues were not addressed. For instance, in order to type-check a functor application $m_1(m_2)$, knowing the module types of m_1 and m_2 was not enough: some knowledge of the underlying implementation of these modules was needed, thus preventing from true separate compilation. A solution to these problems has only been found recently, with the formalism of translucent sums [HL94], or manifest types [Ler94, Ler95]. These approach share the same idea: the implementation of types that can be seen outside a given module must appear in the module type; there is no possibility for knowing the implementation of the type component of a module if it does not appear in its type. Thus, module types of module variables belonging to the environment give all the information needed for type-checking a module, allowing true separate compilation: one needs only declaring the types of the modules needed by another one at the time of compiling it. Side-effects were at the core of the initial SML module language: as abstraction was implemented through *generative* data type declarations, module language constructs (especially the application) could generate new types. But, as this way understanding type abstraction is a too low-level point of view, generativity stopped being considered a key notion: in [HL94], there is no such notion, and in [Ler95], the generative behavior of functor application is replaced by an *applicative* one. Thus, module languages look more and more like *functional* languages (at least as soon as no side-effect is present in the base language). Therefore, one could expect a soundness proof of these systems to follow from the study of reductions in these calculus. But none seems to have been provided yet.

From a theoretical point of view, this is very unsatisfying, and as a consequence, the question of normalization makes little sense. Moreover, from a practical point of view, this could prevent us from adapting this module system to proof systems or logical frameworks. For instance in Elf [HP92], it has been chosen not to implement the *sharing* specification, in order to retain only theoretically well-established features. Indeed, having some strange features in a programming language might not be too dangerous, but in proof systems, it might make it inconsistent.

In order to study reductions in module systems in this paper, we decided to start from the system of [Ler95] because of its conceptual simplicity, and of its relative independence with respect to the base module language. Unfortunately, this system lacks the subject reduction property: a module expression of a given type may reduce to one that has not the same type or is even not syntactically well-formed. Indeed, in this module system as in SML, access to module components is only allowed through expressions of the form $p.n$ where n is a name of a field and p an access path, access paths being a syntactic fragment of module expressions:

- in SML and in the system of [Ler94], paths are of the form $x_1.\ldots.x_n$, where $x_1, \ldots, x_n$ are identifiers;

*This research was partially supported by the ESPRIT Basic Research Action Types and by the GDR Programmation cofinanced by MRE-PRC and CNRS.

- in [Ler95], they also contain simple functor application $p_1(p_2)$ of paths to paths.

Thus the structure `struct type` $elt = m.t$ `end` is syntactically well-formed if and only if m is indeed a path.

Let us now consider applying a functor implementing data structures on ordered types to a module m. If the functor is

```
functor(Ordered_type : sig type t val compare : t -> t -> bool end)
  struct type elt = Ordered_type.t ... end
```

and we try to reduce its application to m, we get `struct type elt=`m`.t end` which is not well-formed if m is not a path. Therefore, we must withdraw syntactic restrictions on access paths: in the following, access paths and module expressions are the same notions.

As pointed out in [Ler95] "This extension adds considerable expressive power but raises delicate issues". Actually, there are two of them.

The first one is type comparison: which definition of type equality has to be chosen? Is type comparison and type-checking decidable? We shall see in section 3 that, given a suitable notion of type normalization, we can compare types, and type inference is decidable.

The second one is the lost of type abstraction in systems which have a typing rule transforming abstract types into types manifestly equal to themselves. In [Ler95], it is stated that

> "If all structure expressions are allowed in paths, the "self" rule makes abstract types that happen to have the same implementation automatically compatible:
>
> `structure A = struct abstype t =` τ `with` *decls* `end`
> `structure B = struct abstype t =` τ `with` *decls* `end`
>
> By application of the "self" rule, we obtain the following signatures for `A` and `B`:
> `A : sig type t = (struct abstype t =` τ `with decls end).t ; ...end`
> `B : sig type t = (struct abstype t =` τ `with decls end).t ; ...end`
> Hence `A.t` = `B.t` which violate type abstraction.
>
> To avoid this problem, all module-language constructs whose evaluation can generate new types (by evaluating an `abstype` or `datatype` definition) must not occur in type projections."

We agree with this statement, but go further: all module-language constructs whose evaluation can generate new types must not occur in type projections nor even in module expressions. Indeed, we think that the signature constraint operation that applies to module expressions — $(m : M)$ denotes the module m whose type is constrained to be M — has nothing to do with the module language, but rather should be part of the definition mechanism. Type abstraction has to be expressed when defining a *module*, not when defining a type.[1] Thus, instead of declaring a module `A` as

```
structure A = struct abstype t = string end (* in SML *)
```

or as

```
module A = (struct type t = string end : sig type t end) (* in Leroy's system *)
```

we would rather declare it in the following way:

```
module A : sig type t end = struct type t = string end
```

In the first two cases, definitions are considered as *transparent* in the sense that `A` can be replaced by its definition, whereas in the third case, the definition of `A` is *opaque*, and export the signature `sig type t end` as the only information about `A`: the definition of `A` can be thrown away, only its declared signature matters. Hence, abstraction is no longer achieved through type generativity (the generation of a unique new type at its declaration) but through type abstraction at module definition time. In fact, this point of view is quite not new: thus, in Modula2 [Wir83], there is only one way to define a new type ; then one specifies in an interface file whether the definition has to be exported.

Let us sum up our proposal. We assume given a base language distinguishing types and values. We build a manifest types calculus on this language. This syntax is very close to that of [Ler95]. Changes are :

[1] As for concrete type definitions, we think the focus is on the definition of a recursive type, not on generativity.

- no syntactic notion of access path;
- no way to explicitly constrain a module with a signature;
- when defining a module, one must give the signature the defined module should export (though an effective implementation could infer the principal type of the module, and take it as the default signature if the user gives none).

It should be noticed that the resulting system is more expressive than that of [Ler95], since it allows modules to be local to an expression. However, module expressions remain second-class values, thus avoiding the difficulties inherent to module systems where modules are first-class values (see [Ler95, HL94]). These local modules might be useful to account for Haskell type classes [Jon91, H+92], or Alcool abstract types [Rou90, Rou92].

The remaining of this paper is organized as follows. In section 2, we present formally our system and prove the subject reduction property for two notions of reduction. We also prove the strong normalization theorem: any reduction strategy for reducing a modular program leads to a "monolithic" program, where no module expression appears. In section 3, we deal with the problem of type inference, and give therefore a deterministic inference system computing the principal type of a module expression; hence, every module enjoys a principal type. We show that the only remaining point, namely the comparison of type expressions, is decidable through a suitable notion of normalization of base-language type expressions. Finally, in section 4, we discuss a possible weakness of our module system with respect to module comparison, and give possible solutions.

2 A calculus enjoying the subject reduction property

We now formalize the previous remarks in a formal calculus derived from [Ler94, Ler95]. It is to be noticed that our calculus does not account for concrete types definitions nor for recursive type definition of ML; however, they can be accounted *via* the use of a fixpoint operator.

2.1 Syntax

We follow syntactic conventions from [Ler95]: v, t, x are names (for value, type, and module components of structures), and v_i, t_i, x_i are identifiers (for values, types and modules). Identifiers are composed of a name plus a stamp part (say an integer). To avoid name clashes, renamings can change the stamp parts of identifiers but the name parts must be preserved to support access by name to structure components.

Values :

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

Types :

$\tau ::= t_i$	identifier
$\mid m.v$	access to a type field of a structure
$\mid$ `int` $\mid \tau \to \tau \mid \ldots$	base-language-dependent type expressions

Module expressions :

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

Structure body :

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

Structure component :

$d ::=$	$\texttt{val}\ v_i = e$	value definition
$\mid$	$\texttt{type}\ t = \tau$	type definition
$\mid$	$\texttt{module}\ x_i : M = m$	module definition

Module type :

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

Signature body :

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

Signature component :

$D ::=$	$\texttt{val}\ v_i : \tau$	value declaration
$\mid$	$\texttt{type}\ t_i$	abstract type declaration
$\mid$	$\texttt{type}\ t_i = \tau$	manifest type declaration
$\mid$	$\texttt{module}\ x_i : M$	module declaration

Environments :

$E ::= S$

Finally, as we want to study of the reductions of the module calculus, we have to distinguish β-reductions at the level of the base-language calculus and at the level of the module calculus. In order not to confuse both of them, we call μ-reduction the β-reduction at the level of module system. That is, μ-reduction is the smallest context-stable relation on the syntax such that $(\texttt{functor}(x_i : M) m_1)(m_2) \rightarrow_\mu m_1\{x_i \leftarrow m_2\}$. We define μ-equivalence as the smallest equivalence relation including the μ-reduction.

2.2 Typing rules

We define the following judgments figures 1 and 2 (we omit well-formedness conditions for module types and we assume base-language dependent rules defining typing judgments $E \vdash e : \tau$ and $E \vdash \tau : \texttt{type}$):

E	$\vdash$	M modtype	module type M is well-formed
E	$\vdash$	$m : M$	module expression m has type M
E	$\vdash$	$M_1 <: M_2$	module type M_1 is a subtype of M_2
E	$\vdash$	$m = m' : M$	considered as modules of type M, m and m' are defining compatible types

Compared to the systems of [HL94, Ler94, Ler95], the main novelty of our system is the comparison of modules *under a given module type*. For instance, under the type $\texttt{sig type}\ t_i\ \texttt{end}$, the module expressions

$$\texttt{struct type}\ t_i = int; \texttt{type}\ u_j = int\ \texttt{end}$$

$$\texttt{struct type}\ t_i = int; \texttt{type}\ u_j = float\ \texttt{end}$$

are equal, but under the type $\texttt{sig type}\ t_i; \texttt{type}\ u_j\ \texttt{end}$, they are not.

We write $BV(S)$ (resp. $BV(E)$) the set of identifiers bound by a signature body S (resp. a typing environment E). As in [Ler94, Ler95], one of the rule for typing module makes use of the strengthening M/m of a module type M by a module expression m: this rule is a way to express the "self" rule saying that every type is manifestly equal to itself. The strengthening operation is defined as follows:

$$\begin{aligned}
(\texttt{sig}\ S\ \texttt{end})/m &= \texttt{sig}\ S/m\ \texttt{end} \\
(\texttt{functor}(x_i : M_1) M_2)/m &= \texttt{functor}(x_i : M_1)(M_2/m(x_i)) \\
\epsilon/m &= \epsilon \\
(D; S)/m &= D/m; (S/m)
\end{aligned}$$

Module expressions ($E \vdash m : M$) and structures ($E \vdash s : S$):

$$E; x_i : M; E' \vdash x_i : M \qquad \frac{E \vdash m : \texttt{sig}\ S_1; \texttt{module}\ x_i : M; S_2\ \texttt{end}}{E \vdash m.x : M\{n_i \leftarrow m.n \mid n_i \in BV(S_1)\}}$$

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

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

$$\frac{E \vdash m : M' \quad E \vdash M' <: M}{E \vdash m : M} \qquad \frac{E \vdash m : M}{E \vdash m : M/m}$$

$$\frac{E \vdash s : S}{E \vdash (\texttt{struct}\ s\ \texttt{end}) : (\texttt{sig}\ S\ \texttt{end})} \qquad E \vdash \epsilon : \epsilon$$

$$\frac{E \vdash e : \tau \quad v_i \notin BV(E) \quad E; \texttt{val}\ v_i : \tau \vdash s : S}{E \vdash (\texttt{val}\ v_i = e_i; s) : (\texttt{val}\ v_i : \tau; S)}$$

$$\frac{E \vdash \tau\ \texttt{type} \quad t_i \notin BV(E) \quad E; \texttt{type}\ t_i = \tau \vdash s : S}{E \vdash (\texttt{type}\ t_i = \tau; s) : (\texttt{type}\ t_i = \tau; S)}$$

$$\frac{E \vdash m : M \quad x_i \notin BV(E) \quad E; \texttt{module}\ x_i : M \vdash s : S}{E \vdash (\texttt{module}\ x_i : M = m; s) : (\texttt{module}\ x_i : M; S)}$$

Module types subtyping $E \vdash M_1 <: M_2$:

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

$$\frac{\sigma : \{1, \ldots, m\} \to \{1, \ldots, n\} \quad \forall i \in \{1, \ldots, n\} \quad E; D_1; \ldots; D_n \vdash D_{\sigma(i)} <: D_i'}{E \vdash \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} <: \texttt{sig}\ D_1'; \ldots; D_m'\ \texttt{end}}$$

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

$$E \vdash \texttt{type}\ t_i = \tau <: \texttt{type}\ t_i \qquad E \vdash \texttt{type}\ t_i <: \texttt{type}\ t_i$$

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

Figure 1: Typing rules

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

$$\frac{\begin{array}{l}E \vdash m.t \ \texttt{type} \quad E \vdash m'.t \ \texttt{type} \\ m \text{ and } m' \text{ have the same head variable } c \\ \text{for all } m_i,\ m_i' \text{ argument of } c \text{ in } m,\ m' \text{ with type } M_i,\ E \vdash m_i \approx m_i' : M_i\end{array}}{E \vdash m.t \approx m'.t}$$

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

(base-language dependent rules, congruence, reflexivity, symmetry and transitivity rules omitted)

Module equivalence : $(E \vdash m \approx m' : M)$

$$\frac{\begin{array}{l}E \vdash m : \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} \\ E \vdash m' : \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} \\ \forall i \in \{1, \ldots, n\} \quad D_i = \texttt{type}\ t_j\,(= \tau) \Rightarrow E \vdash m.t \approx m'.t \\ D_i = \texttt{module}\ x_i : M \Rightarrow E \vdash m.x \approx m'.x : M\{n \leftarrow n_i \in BV(\texttt{sig}\ D_1; \ldots; D_n\ \texttt{end})\}\end{array}}{E \vdash m \approx m' : \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end}}$$

$$\frac{E \vdash m : \texttt{functor}(x_i : M_1)\,M_2 \quad E \vdash m' : \texttt{functor}(x_i : M_1)\,M_2 \quad E; \texttt{module}\ x_i : M_1 \vdash m(x_i) \approx m'(x_i) : M_2}{E \vdash m \approx m' : \texttt{functor}(x_i : M_1)\,M_2}$$

Figure 2: Typing rules

$$\begin{array}{rcl}(\texttt{val}\ v_i{:}\tau)/m & = & \texttt{val}\ v_i{:}\tau \\ (\texttt{type}\ t_i)/m & = & \texttt{type}\ t_i{=}m.t \\ (\texttt{type}\ t_i{=}\tau)/m & = & \texttt{type}\ t_i{=}\tau \\ (\texttt{module}\ x_i : M)/m & = & \texttt{module}\ x_i : (M/m.x)\end{array}$$

2.3 Module reductions

We now focus on reductions in the module language. We give our results first, then explain briefly at the end of this subsection how to prove them.

Theorem 1 (subject reduction for μ-reduction) *If $E \vdash m : M$, and $m \rightarrow_\mu m'$, then $E \vdash m' : M$.*

Theorem 2 (Confluence of μ-reduction) *The μ-reduction is confluent*

Theorem 3 (Strong normalization for μ-reduction) *The μ-reduction is strongly normalizing.*

However, μ-reduction in itself is not very interesting. Indeed, module expressions are very often in μ-normal form. Instead, we can study what happens when we replace a module by its definition, that is, what happens when we add to μ-reduction the ρ-reduction defined as the smallest context-stable relation such that

$$\begin{array}{rl}\texttt{struct}\ S_1; \texttt{type}\ t_i = \tau; S_2\ \texttt{end}.t & \rightarrow_\rho \\ \quad \tau\{n_i \leftarrow \texttt{struct}\ S_1; \texttt{type}\ t_i = \tau; S_2\ \texttt{end}.n \mid n_i \in BV(S_1)\} & \\ \texttt{struct}\ S_1; \texttt{val}\ v_i = e; S_2\ \texttt{end}.t & \rightarrow_\rho \\ \quad e\{n_i \leftarrow \texttt{struct}\ S_1; \texttt{val}\ v_i = e; S_2\ \texttt{end}.n \mid n_i \in BV(S_1)\} & \\ \texttt{struct}\ S_1; \texttt{module}\ x_i : M = m; S_2\ \texttt{end}.t & \rightarrow_\rho \\ \quad m\{n_i \leftarrow \texttt{struct}\ S_1; \texttt{module}\ x_i : M = m; S_2\ \texttt{end}.n \mid n_i \in BV(S_1)\} &\end{array}$$

A functional program being of the form $\texttt{struct}\ s\ \texttt{end}.result$ in an empty environment, $\mu\rho$-reducing it is an easy way to transform it into a single base-language expression where no module construct appear, provided that the reduction process terminates.

Then we have the following results:

Theorem 4 (Subject reduction for $\mu\rho$ reduction) *If $E \vdash m : M$, and $m \to_{\mu\rho} m'$, then $E \vdash m' : M$.*

Theorem 5 (Confluence of $\mu\rho$-reduction) *The $\mu\rho$-reduction is confluent*

Theorem 6 (Strong normalization for $\mu\rho$-reduction) *The $\mu\rho$-reduction is strongly normalizing.*

Theorem 6 means we can transform every modular program into one involving only base-language constructs. In the following section, we address the question to know whether the modular program and the base-language program have the same semantics. This result is a kind of "conservativity" property. Indeed, in a proof language, this result imply that every inhabited type in the empty environment for the module language is inhabited in the base language, that is that every proposition provable within the module system is provable in the base proof language.

For both reduction notions, confluence properties are proved with the standard Tait and Martin-Löf's method [Tak93].

Subject reduction for μ and ρ is proved the usual way (substitution property and study of possible types of a functor)

In this proof, we have in particular to prove the following proposition:

Proposition 1 *If $E \vdash M$* `modtype` *and $E \vdash$* `(functor(`$x_i : M'$`)`m`)(`x_i`)` $: M$ *then $E \vdash$* `(functor(`$x_i : M'$`)`m`)(`x_i`)` $\approx m : M$

This proposition implies that two μ-equivalent modules for a given type are equal for this type.

In a first attempt, we put this property as a rule of our system, as is done in [HL94], but this rendered the proof of type normalization untractable in section 3 (in the system of [HL94], type inference is anyway undecidable).

As for theorems 3 and 6, strong normalization is proved first for a typing system $\vdash_w$ that is weaker than $\vdash$, obtained by requiring that signatures in a subtype relation have the same number of component ($m = n$ in the subtyping rule for signatures). Thus, `sig type` $t = \tau$ `type` $u = \tau'$ `end` is a subtype of `sig type` t `type` $u = \tau'$ `end` but not of `sig type` t `end`.

We can do for $\vdash_w$ a proof similar to [Coq87] for the Calculus of Constructions (in fact, we only need the part of the proof concerning dependent types): we define a notion of *full premodel* for our calculus (that is, an infinite set of constants such that for every module type build upon this set there is a constant of that type in the set), and interpret the terms of our calculus in a way such that every interpretation of a module type is strongly normalizing, and the interpretation of a module type is the set of module expressions of this type.

The case of $\vdash$ is then handled by the study of explicit coercions. These two proofs are not detailed because of their lengths.

2.4 Denotational semantics

Following [Ler95], the denotational semantics of the calculus (for the functional fragment of the base language) is obtained by erasing all type information, mapping structures to records and functors to functions. We easily have the following result:

Theorem 7 *The $\mu\rho$-reduction preserves the denotational semantics. More precisely, if e is a well-typed expression of the base language involving module expressions, then the semantics of e is not* wrong*, and if e $\mu\rho$-reduces to e' then e and e' have the same semantics.*

Proof: Since $\mu\rho$ is strongly normalizing, we can prove this statement by induction on the maximal length of a $\mu\rho$-reduction path starting from e. The proof is then straightforward.

As a corollary, the above transformation of a modular program into a monolithic one preserves its semantics.

Typing:

$$E; x_i : M; E' \vdash_{\mathcal{A}} x_i : M \qquad \frac{E \vdash_{\mathcal{A}} m : \texttt{sig}\ S_1; \texttt{module}\ x_i : M; S_2\ \texttt{end}}{E \vdash_{\mathcal{A}} m.x : M\{n_i \leftarrow m.n \mid n_i \in BV(S_1)\}}$$

$$\frac{E \vdash_{\mathcal{A}} s : S}{E \vdash_{\mathcal{A}} (\texttt{struct}\ s\ \texttt{end}) : (\texttt{sig}\ S\ \texttt{end})}$$

$$\frac{E \vdash M\ \texttt{modtype}\ \ x_i \notin BV(E)\ \ E; \texttt{module}\ x_i : M \vdash_{\mathcal{A}} m : M'}{E \vdash_{\mathcal{A}} \texttt{functor}(x_i : M)m : \texttt{functor}(x_i : M)M'/m}$$

$$\frac{E \vdash_{\mathcal{A}} m_1 : \texttt{functor}(x_i : M)M'\ E \vdash_{\mathcal{A}} m_2 : M''\ \ E \vdash_{\mathcal{A}} M''/m_2 <: M}{E \vdash_{\mathcal{A}} m_1(m_2) : M'\{x_i \leftarrow m_2\}}$$

$$E \vdash_{\mathcal{A}} \epsilon : \epsilon$$

$$\frac{E \vdash_{\mathcal{A}} e : \tau\ \ v_i \notin BV(E)\ \ E; \texttt{val}\ v_i : \tau \vdash_{\mathcal{A}} s : S}{E \vdash_{\mathcal{A}} (\texttt{val}\ v_i = e_i; s) : (\texttt{val}\ v_i : \tau; S)}$$

$$\frac{E \vdash \tau\ \texttt{type}\ \ t_i \notin BV(E)\ \ E; \texttt{type}\ t_i = \tau \vdash_{\mathcal{A}} s : S}{E \vdash_{\mathcal{A}} (\texttt{type}\ t_i = \tau; s) : (\texttt{type}\ t_i = \tau; S)}$$

$$\frac{E \vdash_{\mathcal{A}} m : M'\ \ E \vdash_{\mathcal{A}} M'/m <: M\ \ x_i \notin BV(E)\ \ E; \texttt{module}\ x_i : M \vdash_{\mathcal{A}} s : S}{E \vdash_{\mathcal{A}} (\texttt{module}\ x_i : M = m; s) : (\texttt{module}\ x_i : M; S)}$$

Subtyping:

$$\frac{E \vdash_{\mathcal{A}} M_2 <: M_1\ \ E; \texttt{module}\ x_i : M_2 \vdash_{\mathcal{A}} M_1' <: M_2'}{E \vdash_{\mathcal{A}} \texttt{functor}(x_i : M_1)M_1' <: \texttt{functor}(x_i : M_2)M_2'}$$

$$\frac{\sigma : \{1, \ldots, m\} \to \{1, \ldots, n\}\ \ \forall i \in \{1, \ldots, n\}\ \ E; D_1; \ldots; D_n \vdash_{\mathcal{A}} D_{\sigma(i)} <: D_i'}{E \vdash_{\mathcal{A}} \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} <: \texttt{sig}\ D_1'; \ldots; D_m'\ \texttt{end}}$$

$$\frac{E \vdash_{\mathcal{A}} \tau \approx \tau'}{E \vdash_{\mathcal{A}} \texttt{val}\ v_i : \tau <: \texttt{val}\ v_i : \tau'} \qquad \frac{E \vdash_{\mathcal{A}} M <: M'}{E \vdash_{\mathcal{A}} \texttt{module}\ x_i : M <: \texttt{module}\ x_i : M'}$$

$$E \vdash_{\mathcal{A}} \texttt{type}\ t_i = \tau <: \texttt{type}\ t_i \qquad E \vdash_{\mathcal{A}} \texttt{type}\ t_i <: \texttt{type}\ t_i$$

$$\frac{E \vdash_{\mathcal{A}} \tau \approx \tau'}{E \vdash_{\mathcal{A}} \texttt{type}\ t_i = \tau <: \texttt{type}\ t_i = \tau'} \qquad \frac{E \vdash_{\mathcal{A}} t_i \approx \tau}{E \vdash_{\mathcal{A}} \texttt{type}\ t_i <: \texttt{type}\ t_i = \tau}$$

Figure 3: Type inference system

3 Type inference

In order to obtain a type inference algorithm, we provide in figures 4 and 4 an inference system which runs in a deterministic way for a given module expression except for type comparison $\approx$ (where two main rules plus reflexivity, symmetry, transitivity and context stability may filter the same type expressions). We show here that this system gives the most general type of a given module expression. The only remaining point to have a type inference algorithm is to get a procedure to decide if two types of the base-language are in the $\approx$ comparison relation.

This system is obtained from the one given figures 1 and 2 in the usual way by moving subsumption and strengthening rules in the application rule, and a notion of δ-reduction of a type is added in order to orient the equality between a field of structure and the corresponding declaration in its signature.

Compared to the type inference system for the system of [Ler95], our system has only one straightforwards case for application whereas the syntactic restriction on access paths leads to the definition of the notion of least subtype of a type where a given module variable does not appear. This notion is rather complex, and above all is not always defined, therefore this system does not have the principal type property ([Ler96]).

Types equivalence: $(E \vdash_{\mathcal{A}} \tau \approx \tau')$

$$\frac{E \vdash_{\mathcal{A}} \tau \to_\delta \tau'}{E \vdash_{\mathcal{A}} \tau \approx \tau'} \qquad \frac{\begin{array}{l} E \vdash m.t \ \texttt{type} \quad E \vdash m'.t \ \texttt{type} \\ m \text{ and } m' \text{ have the same head variable } c \\ \text{for all } m_i, m_i' \text{ argument of } c \text{ in } m, m' \text{ with type } M_i,\ E \vdash m_i \approx m_i' : M_i \end{array}}{E \vdash m.t \approx m'.t}$$

(reflexivity, symmetry and transitivity omitted)
Reduction:

$$E_1; \texttt{type}\ t_i = \tau; E_2 \vdash_{\mathcal{A}} t_i \to_\delta \tau \qquad \frac{E \vdash_{\mathcal{A}} m : \texttt{sig}\ S_1; \texttt{type}\ t_i = \tau; S_2\ \texttt{end}}{E \vdash_{\mathcal{A}} m.t \to_\delta \tau\{n_i \leftarrow m.n \mid n_i \in BV(S_1)\}}$$

(context rules for base-language types omitted)
Module equivalence: $(E \vdash_{\mathcal{A}} m \approx m' : M)$

$$\frac{\begin{array}{l} E \vdash_{\mathcal{A}} m : N \quad E \vdash_{\mathcal{A}} N/m <: \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} \\ E \vdash m' : N' \quad E \vdash_{\mathcal{A}} N'/m' <: \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} \\ \forall i \in \{1, \ldots, n\} \quad D_i = \texttt{type}\ t_j (= \tau) \Rightarrow E \vdash_{\mathcal{A}} m.t \approx m'.t \\ D_i = \texttt{module}\ x_i : M \Rightarrow E \vdash_{\mathcal{A}} m.x \approx m'.x : M\{n \leftarrow n_i \in BV(\texttt{sig}\ D_1; \ldots; D_n\ \texttt{end})\} \end{array}}{E \vdash_{\mathcal{A}} m \approx m' : \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end}}$$

$$\frac{\begin{array}{l} E \vdash_{\mathcal{A}} m : M \quad E \vdash_{\mathcal{A}} M/m <: \texttt{functor}(x_i : M_1) M_2 \\ E \vdash_{\mathcal{A}} m' : M' \quad E \vdash_{\mathcal{A}} M'/m' <: \texttt{functor}(x_i : M_1) M_2 \\ E; \texttt{module}\ x_i : M_1 \vdash_{\mathcal{A}} m(x_i) \approx m'(x_i) : M_2 \end{array}}{E \vdash_{\mathcal{A}} m \approx m' : \texttt{functor}(x_i : M_1) M_2}$$

Figure 4: Type inference system

3.1 Soundness and completeness

Theorem 8 (Soundness) *If $E \vdash_{\mathcal{A}} m : M$ then $E \vdash m : M$ (and thus $E \vdash m : M/m$) ; if $E \vdash_{\mathcal{A}} M <: M'$ then $E \vdash M <: M'$; if $E \vdash_{\mathcal{A}} \tau \approx \tau'$ then $E \vdash \tau \approx \tau'$.*

Proof: Induction on the derivation.

Theorem 9 (Completeness) *If $E \vdash m : M$, then there exists a unique M' such that $E \vdash_{\mathcal{A}} m : M'$ and $E \vdash_{\mathcal{A}} M'/m <: M$. Thus M'/m is the principal type of m. If $E \vdash M <: M'$ then $E \vdash_{\mathcal{A}} M <: M'$; if $E \vdash \tau \approx \tau'$ then $E \vdash_{\mathcal{A}} \tau \approx \tau'$.*

Proof: Induction on the derivation

3.2 Type normalization

In order to compare two types, we shall give a notion of type normalization in our system in order to have for each type a canonical form. The first notion coming in mind is δ-normalization. However, it is not enough; thus in environment

$$E; x_h : \texttt{functor}(x_i : \texttt{sig type}\ t_i\ \texttt{end})\texttt{sig type}\ u_j\ \texttt{end}$$

the expressions

$$x_h\,((\texttt{functor}(x_i : \texttt{sig}\ \ \texttt{end})\texttt{struct type}\ t_i = int\ \texttt{end})(\texttt{struct}\ \ \texttt{end})).u$$

and

$$x_h(\texttt{struct type}\ t_i = int\ \texttt{end}).u$$

are in δ-normal form, and syntactically distinct though they are equivalent as

$$\begin{array}{ll} E \vdash_{\mathcal{A}} & (\texttt{functor}(x_i : \texttt{sig end})\texttt{struct type}\, t_i = int\ \texttt{end})(\texttt{struct end}) \\ & \approx\ \texttt{struct type}\, t_i = int\ \texttt{end} \\ & :\texttt{sig type}\, t_i\ \texttt{end} \end{array}$$

However, we shall see that we can always proceed in this way to compare types, that is, by δ-normalizing them, then comparing module expressions that are arguments of the head variable (in [Ler94, Ler95], it seems that types are compared through the same normalization process but δ-normal access paths obtained are compared syntactically; hence, if t is an abstract type of a functor x, $x(y).t$ is different from $x(z).t$ even if the definition of z is y).

Then, we may wonder whether this process always terminates. In order to answer this question, we first give the following definition:

Definition 1 (Normalizing types and normalizing modules for a given module type) *In an environment E, we say a module m is normalizing for module type M if $E \vdash m : M$, and one of the following case is verified:*

- $M = \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end}$*, for all i such that $D_i = \texttt{type}\, t_j (= \tau)$, $m.t$ is normalizing and for all i such that $D_i = \texttt{module}\, x_j : N$, $m.x$ is normalizing for type $N\{n_h \leftarrow m.n \mid n_h \in BV(D_1, \ldots, D_{i-1})\}$;*
- $M = \texttt{functor}(x_i : M_1)M_2$*, and $m(x_i)$ is normalizing for type M_2 in $E;\texttt{module}\, x_i : M_1$.*

A type τ is said to be normalizing if and only if it has a δ-normal form, and the arguments of the head variables of the access path of its normal form are normalizing (for types expected by the head variables).

We have the following result:

Theorem 10 (Type normalization) *If $E \vdash_{\mathcal{A}} m : M$ then m is normalizing for M; if $E \vdash_{\mathcal{A}} \tau$ type then τ is normalizing.*

Proof: The proof can be done by defining a reducibility notion as in [GLT89] for the simply-typed lambda-calculus. We define the notion of reducible type and reducible module expression for a given type as follows:

- reducible types are normalizing types;
- m is reducible for $\texttt{sig}\ S\ \texttt{end}$ if for every $\texttt{type}\, t_i (= \tau)$ in S, $m.t$ is reducible and for every $\texttt{module}\, x_i : M$ in s, $m.x$ is reducible for $M\{n_i \leftarrow m.n \mid n_i \in BV(s)\}$;
- m is reducible for $\texttt{functor}(x_i : M_1)M_2$ if for every m' reducible for M_1, $m(m')$ is reducible for $M_2\{x_i \leftarrow m'\}$.

The reader may check that this definition is well-founded (by induction on the size of module types for a suitable notion of size). One can then prove the two following lemmas:

Lemma 1 *If $E \vdash m : M$ then m is reducible of type M*

Lemma 2 *Every reducible term is normalizing*

Then we have to check that normalization is a way to compare base-language types:

Lemma 3 *For all types τ and τ' such that $E \vdash_{\mathcal{A}} \tau \approx \tau'$, δ-normal forms of τ and τ' have the same head variables, and field selections and arguments applied to these variables are equal (for the expected types for the head variables).*

Proof: By induction on the derivation of the equality.

3.3 Termination

We have seen that we have a way to compare well-formed type. We now only have to see that we have a typing algorithm, that is the algorithm stops even if the given module is ill-typed.

Theorem 11 *The $\vdash_{\mathcal{A}}$ gives a type inference algorithm, terminating on every module expression. Therefore, type inference for the module system is decidable.*

Proof: Theorem 9 says the inference system terminates on every well-typed module expression. Hence, the subtyping inference system terminates on every couple of well-formed module types (since the sum of their size decreases, until we have to infer the type of well-typed module expressions). Then, typing rules terminates, since the size of module expressions we want to infer the type of are decreasing and the subtyping test needed for the application rule is only performed between well-formed module types.

4 Discussion

Now we would like to discuss strengths and weaknesses of our proposal, and especially of one key notion: module equality.

In order to eliminate the somewhat artificial distinction between module types and access paths, we had to slightly complicate the comparison of base-language types, and we needed to add the notion of module equality for a given type. On the one hand, we believe this comparison is now more intuitive. Moreover our module comparison provides a simple semantics to manifest modules in signatures — that would be an equivalent of substructures sharing in SML[2] — in terms of a syntactic sugar:

```
sig
  module x : sig
               type t
               val compare : t -> t -> bool
             end
           = StringOrd
end
```

would expands to

```
sig
  module x : sig
               type t = StringOrd.t
               val compare : t -> t -> bool
             end
end
```

On the other hand, generating abstract types is now more difficult, in the sense that a functor cannot generate an abstract type by itself.

Let us study this problem on an example: implementing finite sets over types equiped with comparison functions. This can be done by a functor `SORT` of the following type:

$$\texttt{SET:functor(}\mathit{Ordered\_type} : \texttt{sig type } t \texttt{ val } \mathit{compare} : t \to t \to \mathit{bool} \texttt{ end)sig type } \mathit{set} \ \ldots \texttt{ end}$$

Now consider implementing finite sets of strings. Which comparison function do we choose? We may want the ASCII lexicographic comparison function, or instead to sort according to the lexicographic ordering based on the natural ordering on the French alphabet lexicographic ordering (where "é" is smaller than "f", whereas it is greater than "f" in ASCII). Then, we would have two modules `StringOrd1` and `StringOrd2` of signature $\texttt{sig type } t = \mathit{string}; \texttt{val } \mathit{compare} : t \to t \to \mathit{bool} \texttt{ end}$, where the *compare* functions are different. If we naively define

[2] The *right* semantics for this notion seems to be still unclear

```
module StringSet1 = SET(StringOrd1)
module StringSet2 = SET(StringOrd2)
```

where the notation `module` x `=` m is a syntactic sugar for `module` x `:` M `=` m where M is the principal type inferred for M, then

$$E \vdash \texttt{StringSet1.set} \approx \texttt{StringSet2.set}$$

since

$$E \vdash \texttt{StringOrd1.t} \approx \texttt{StringOrd2.t}$$

hence

$$E \vdash \texttt{StringOrd1} \approx \texttt{StringOrd2} : \texttt{sig type } t = \mathit{string}; \texttt{val } \mathit{compare} : t \to t \to \mathit{bool} \texttt{ end}$$

.

Notice that this problem is *semantic* in nature: since the manipulation of `StringSet1.set` and `StringSet2.set` is highly dependent upon the `compare` functions, letting them be equal can give strange results but no type error can occur. Nonetheless some safety brought by abstract data types is lost. The same problem arises in Jones's proposal for modular programming [Jon96] with parameterized signatures since in this framework, a type can only be parameterized by other types. There are at least two ways towards a solution:

- Force the programmer to always give an explicit signature when defining a module.
- Extend the system of abstract/manifest types to abstract/manifest values and make the comparison of module checking that values components are the same. The comparison of values should be done through a decidable equivalence relation included in the semantic equivalence (which is itself generally undecidable), for instance through α-equivalence or — in normalizing languages such as system F or the Calculus of Constructions — β-equivalence. Such an extension would solve this problem as it would guarantee that if m and m' are the same for `sig val` $v : \tau$ `end`, then $m.v$ and $m'.v$ have the same semantics. A comparison over values may seem unusual to an ML programmer, but it is usual in type systems where types may depend on values. Anyway, adding it would make the module comparison more restrictive, but closer to intuition since equal modules would have equal denotational semantics; more precisely, they would have to be intentionally equal.

The first solution however cannot solve the problem in case of local modules. Moreover, it relies on the discipline of the module user, not on that of the module provider. The second one is better since this kind of unexpected use of a module is no longer legal.

5 Conclusion

Our module system is close to those of [Ler95, HL94]. However, to our knowledge, it is the first SML-like module system whose subject reduction property is proven. This allows the theoretical study of reductions, leading to the strong normalization proofs. Also, we establish a kind of "conservativity" theorem: a modular functional program can be expanded to a monolithic non-modular one.

In the system of [HL94], type inference is undecidable. In that of [Ler95] syntactic restrictions on access paths make some modules lack a principal type and complicate type inference [Ler96]. On the contrary, in our system every module expression enjoys a principal type, and type inference is decidable.

We think the replacement of type generativity by abstraction at definition gives a less operational account for type abstraction, which seems to be preserved. We conjecture that the representation independence proof of [Ler95] is adaptable to our system. It would give a more formal result to this respect.

We think our system helps in understanding modules from a type-theoretical point of view. The study of module reductions in the system itself helps bringing the study of module systems back to the study of typed lambda-calculi. Moreover, it seems to provide a firm basis for its use in proofs systems.

In this respect, we are currently working on its adaptation to the Calculus of Constructions [CH88, CCF+95], which should be quite easy (despite the fact there is no distinction between types and terms)

in order to have a modular proof language well-suited to proving modular programs. Since the Calculus of Construction is both a programming language and a proof language, this would have the advantage to provide a unified framework, simpler than the Extended ML approach [San90, KSTar] because of the inherent complexity of the semantic of the SML module system. We also believe our system may help in designing a safe and powerful module system for Elf.

In the same direction, it would be interesting to compare our module system with Bourbaki's mathematical notion of theory [Bou70], which is for instance implemented in the IMPS [FGT95] prover.

References

[Bou70] Nicolas Bourbaki. *Eléments de Mathématique; Théorie des Ensembles*, chapter IV. Hermann, Paris, 1970.

[Car89] L. Cardelli. Typeful programming. In E. J Neuhold and M. Paul, editors, *Formal description of programming concepts*, pages 431–507. Springer-Verlag, 1989.

[CCF+95] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual Version 5.10. Technical Report 0177, INRIA, July 1995.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comp.*, 76:95–120, 1988.

[CL90] L. Cardelli and X. Leroy. Abstract types and the dot notation. In M. Broy and C. B. Jones, editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*. North Holland, 1990.

[Coq87] Thierry Coquand. A meta-mathematical investigation of a Calculus of Constructions. Private Communication, 1987.

[FGT95] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. *The IMPS User's Manual*. The MITRE Corporation, first edition, version 2 edition, 1995.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[H+92] P. Hudak et al. Report on the Programming Language Haskell: A no-strict purely Functional Language. *ACM Sigplan Notices*, 27(5), May 1992.

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

[HMT87] R. Harper, R. Milner, and M. Tofte. A type discipline for program modules. In *TAPSOFT 87*, volume 250 of *LNCS*, pages 308–319. Springer-Verlag, 1987.

[HMT90] R. Harper, R. Milner, and M. Tofte. *The definition of Standard ML*. The MIT Press, 1990.

[HP92] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. Technical Report CMU-CS-92-191, Carnegie Mellon University, Pittsburgh, Pennsylvania, september 1992.

[Jon91] Mark. P. Jones. An Introduction to Gofer. Available by www at http://www.cs.chalmers.se, 1991.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structures. In *23rd Symposium on Principles of Programming Languages*. ACM Press, 1996. To appear.

[KSTar] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, To appear.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symp. Principles of Progr. Lang.*, pages 109–122. ACM Press, 1994.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd Symposium on Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[Ler96] Xavier Leroy, 1996. Private Communication.

[MP88] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Prog. Lang. Syst.*, 10(3):470–502, 1988.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *19th Symposium on Principles of Programming Languages*, pages 305–315. ACM Press, 1992.

[Rou90] François Rouaix. *Alcool 90, Typage de la surcharge dans un langage fonctionnel.* Thèse, Université Paris VII, 1990.

[Rou92] François Rouaix. The Alcool 90 report. Technical report, INRIA, 1992. Included in the distribution available at `ftp.inria.fr`.

[San90] Don Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, pages 99–130. Springer Workshops in Computing, 1990.

[Tak93] M. Takahashi. Parallel reductions in λ-calculus. Technical report, Department of Information Science, Tokyo Institute of Technology, 1993. Internal report.

[Wir83] N. Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer-Verlag, 1983.