An Applicative Module Calculus*

Judicaël Courant
Laboratoire d'Informatique du Parallélisme
CNRS URA 1398
46, allée d'Italie
69364 Lyon cedex 07
France
Judicael.Courant@ens-lyon.fr
tel. (+33) 4 72 72 85 82
fax (+33) 4 72 72 80 80

LIP
46, allée d'Italie
69364 Lyon cedex 07
FRANCE

Abstract. The SML-like module systems are small typed languages of their own. As is, one would expect a proof of their soundness following from a proof of subject reduction. Unfortunately, the subject-reduction property and the preservation of type abstraction seem to be incompatible.
As a consequence, in relevant module systems, the theoretical study of reductions is meaningless, and for instance, the question of normalization of module expressions can not even be considered.
In this paper, we analyze this problem as a misunderstanding of the notion of module definition. We build a variant of the SML module system — inspired from recent works by Leroy, Harper, and Lillibridge — which enjoys the subject reduction property. Type abstraction — achieved through an explicit declaration of the signature of a module at its definition — is preserved. This was the initial motivation. Besides our system enjoys other type-theoretic properties: the calculus is strongly normalizing, there are no syntactic restrictions on module paths, it enjoys a purely applicative semantics, every module has a principal type, and type inference is decidable. Neither Leroy's system nor Harper and Lillibridge's system has all of them.

1 Introduction

The ability to build a program from a collection of pieces of code is essential for software programming and reuse. Modern programming languages provide the programmer a way to decompose any program into modules of code that are small and as independent as possible.

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

However, these modules can not be completely independent since they have to interact within the whole program. Therefore, each module may have an associated description in the form of an *interface* file. This file should give the properties that the module intends to export.

These interface files help linking together the information about the modules. Thus, they should allow separate compilation of the whole program: one should be able to compile any given unit, provided that the interface files of the other units are present, even if some modules are not yet implemented.

On the contrary, a compiler should consider that any property of the module that is not described in this file is irrelevant, in the sense that it could be lost after a reimplementation of the given module. Therefore, it helps isolating modules one from each other.

1.1 Standard ML

The Standard ML language is particularly interesting, with respect to the modularity concerns because of the power of its module system [HMT87,HMT90]. Indeed, this system allows the definition and use of parameterized modules. This notion of parameterized modules allows to plug a module into another one. For instance, a module defining balanced trees over an ordered type can be parameterized by a module defining a type and a comparison function over elements of this type.

In the SML terminology, a non-parameterized module is called a *structure*, and a parameterized module is called a *functor*. Recent works about SML allow modules to be parameterized by a functor (that can itself be a module parameterized by a functor...). The interface of a structure is called a *signature*. In fact, a *module type* can be associated to each module, and signatures are particular cases of module types. Actually, the module system is a small typed language of its own.

Let us study a little example. The structure declaration

```
structure OrdInt = struct
  type t = int
  val compare = fn (x : int) (y : int) => x - y
end
```

binds the variable `OrdInt` to a structure with a type component `OrdInt.t` and a value component `OrdInt.compare` of type `int -> int -> int`. Therefore `OrdInt` is said to have the following signature

```
sig
  type t
  val compare : t -> t -> int
end
```

Here is another example: the signature of a module defining polymorphic association tables could be

```
sig
  type key (* type of keys *)
  type 'a t (* type of tables *)
  val empty: 'a t (* an empty table *)
  val add: key -> 'a -> 'a t -> 'a t (* add a binding *)
  val find: key -> 'a t -> 'a (* look for a binding *)
  val remove: key -> 'a t -> 'a t (* remove a binding *)
end
```

One could implement such a table in the form of a balanced tree. A comparison function of type `key -> key -> int` is needed to store elements in the tree. The natural way to do this is writing the following functor definition:

```
functor MakeTable(structure Ord:
                  sig
                    type t
                    val compare : t -> t -> int
                  end) = struct
  type key = Ord.t
  type 'a t = Empty
            | Node of key * 'a * 'a t * 'a t
  val find = ...
  val add = ...
  val remove = ...
end
```

Then a table over integers can be implemented as follows:

```
structure Table : sig type key
  type 'a t
  val empty: 'a t
  val add: key -> 'a -> 'a t -> 'a t
  val find: key -> 'a t -> 'a
  val remove: key -> 'a t -> 'a t
end
    = MakeTable(OrdInt)
```

It happens that a functor should take two structures S_1 and S_2 as arguments, and some relations between these structures are required for typechecking the functor. In that case, SML has a way to express that these structures share a common type; that is, one can declare that each time the functor will apply to actual structures, $S_1.t$ and $S_2.u$ will be the same. Such a declaration is called a *sharing constraint*.

1.2 Motivations and Aims

The ability to compose code through the module system of SML appears as a powerful and fruitful approach. But, to our knowledge, small-step operational semantics of SML-like module systems have been little studied. However, such a study would be very interesting for several reasons.

Type-Theoretical Motivations Studying the reductions in module systems would improve the theoretical understanding of modules. In particular, it seems that no soundness proof of module systems exists yet: such a proof for a call-by-value semantics is claimed to be an important direction for future research in [HL94], and it is clear that their module system does not enjoy the subject-reduction property for an arbitrary reduction strategy. This is very unsatisfactory from a theoretical point of view. And it is well-known that a clear understanding of the semantics of a programming language helps the casual programmer in writing programs in this language; on the contrary, writing programs in a language with an intricate semantics is often a difficult task (did you ever try to write some complex TEX macro?).

Adaptation to Proof Systems The module system of SML is quite independent from the base programming language. Therefore, one could imagine to adapt it to other languages. But the absence of subject-reduction property for a lazy reduction strategy in existing module systems does not allow their adaptation to pure functional languages such as lazy ML or Haskell. Also the adaptation of existing module systems to proof languages or logical frameworks could be interesting. But, the absence of soundness proof could prevent us from such an adaptation. For instance in Elf [HP92] which has an SML-like module system, it has been chosen not to implement the *sharing* specification of SML, in order to retain only theoretically well-established features. Indeed, having some strange features in a programming language might not be too dangerous, but this can make a proof system inconsistent.

Mobile Code Security The study of small-step semantics of a language is also very interesting with respect to security concerns about *mobile code*. Indeed, it has been proposed recently[NL96] that any mobile code could be provided with a formal proof of its safety with respect to a given security policy, so that clients only need to have a proof-checker verify this proof in order to trust the mobile code. It would be possible to build a compiler producing efficient code from SML programs together with a proof of their safety if we had a formal proof that well-typed programs have a safe behavior.

Goals Therefore, our aim in this paper is to study module reductions in an SML-like module system, and to prove the subject-reduction property. However, the SML module system suffers several limitations; therefore studying from a type-theoretic point of view is difficult. We expose these limitations in section 2. Fortunately, some recent works propose SML-like module systems that are better suited for this study; we shall briefly expose them, then expose their limitations. Then we propose a new variant of the SML module system. In section 3, we expose the main theoretical results about this system. Finally, we conclude in section 4, and draw possible future directions. It should be noticed that we can't give any detailed proof in this paper because of size restrictions.

2 Informal Design of a Module System Enjoying the Subject-Reduction Property

2.1 SML Limitations

The SML module system was designed for use at the interactive toplevel, and therefore separate compilation issues were not addressed. For instance, in the example of the previous section, the type checker knows that `Table.key` is equal to `int` whereas this property is not stated in the signature of `Table`. In other words, some knowledge of the underlying implementation of the module `Table` is needed to type-check some expressions involving it. This forbids a true separate compilation facility in the style of Modula2. Moreover, this problem is complicated by the problem of *sharing constraints*. Some works tried to address the issue of separate compilation of SML but gave only partial solutions that are merely of engineering nature [HLPR94,AM94].

Moreover, side-effects were at the core of the initial SML module language. Type abstraction was implemented through a mechanism of stamp generation: each time a functor was applied, new stamps corresponding to the definition of new types were generated. As module language constructs could generate new types, studying the semantics of the module language was rather difficult.

2.2 Translucent Sums and Manifest Types

A solution to these problems are the formalisms of translucent sums [HL94], or manifest types [Ler94,Ler95]. These approaches are variants of the SML module system. They both 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. In these approaches, sharing constraints are not needed since they can be replaced by judicious manifest type declarations [Ler96b].

For instance, the `OrdInt` structure of the previous subsection would have the following signature:

```
sig
  type t = int
  val compare : t -> t -> int
end
```

The `MakeTable` functor could be given the following signature:

```
functor(Ord : sig
                type t
                val compare : t -> t -> int
              end)
  sig
    type key = Ord.t
    type 'a t
    val empty: 'a t
```

```
    val add: key -> 'a -> 'a t -> 'a t
    val find: key -> 'a t -> 'a
    val remove: key -> 'a t -> 'a t
  end
```

If the functor `MakeTable` is declared with this signature, then the type of tables is abstract since it does not appear in the signature, but the type of key is manifestly equal to `Ord.t`, so that `MakeTable(OrdInt)` has signature

```
sig
  type key = OrdInt.t
  type 'a t
  val empty: 'a t
  val add: key -> 'a -> 'a t -> 'a t
  val find: key -> 'a t -> 'a
  val remove: key -> 'a t -> 'a t
end
```

and actual elements of type `OrdInt.t`, namely `int`, can be added to the table or retrieved from it.

In the manifest types approach [Ler94], a module definition is given together with a signature, and the type-checking of declarations that come after this definition relies only on the signature and can forget the actual implementation of the module. This allows true separate compilation: one needs only to declare the types of the modules needed by another one at compile time. Then separate compilation *à la* **Modula2** can be achieved: a compiler such as **Objective Caml** recognizes signature files and module implementation files; the compilation of a module implementation file needs only the other module signature files to be compiled.

In [HL94,Ler95], a module expression m can be coerced to a module type M so that the most general type of the coerced expression $(m : M)$ is M. Therefore, if every module definition is a coerced expression, compilation can be done separately.

But, as the generative way for understanding type abstraction is a too low-level point of view, generativity stopped being considered a key notion in these works. 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).

2.3 Informal Requirements

In this subsection, we informally address the issue arising in the design of a module system in the manifest types/translucent sums style enjoying the subject-reduction property.

Syntax Let us consider the following module expression:

```
(functor(Ord : sig type t val compare : t -> t -> int)
   struct
```

```
    type key = Ord.t
  end) (struct
          type t = int
          val compare =
            fn (x : int) (y : int) => x-y
        end)
```

One would like to say this expression reduces to

```
struct
  type key = (struct
                type t = int
                val compare =
                  fn (x : int) (y : int) => x-y
              end).t
end
```

Unfortunately, in the SML module system, as well as in [Ler94] and [Ler95] formalisms, this expression is not even syntactically well-formed. Therefore, these systems do not enjoy the subject-reduction property. Indeed, in these module systems 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 $x_1.x_2.\cdots.x_n$, where $x_1, x_2, \ldots, x_n$ are identifiers;
- in [Ler95], they may also contain simple functor application $p_1(p_2)$ of paths to paths.

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

Therefore, we shouldn't have any restriction on access paths. We should choose a calculus in which access paths and module expressions are the same notions.

Type Abstraction This extension adds considerable expressive power but raises a delicate issues: the possible loss of type abstraction.

Indeed, one needs a typing rule which transforms abstract types into types manifestly equal to themselves. This typing rule is generally called the "self" rule or the "strengthening" rule. Such a rule merely says that if a module `x` defines an abstract type `t`, that is `x : sig type t end` then `x` also defines a type `t` which is equal to `x.t`, that is `x : sig type t = x.t end`. This rule is useful when one want to type the application of a functor needing an argument of type `sig type t = x.t end` to `x`.

But if abstraction is achieved through a coercion operator as in [Ler95] and [HL94], we have the following problem. Let us define a module `x` defining an abstract type `t`: we define `x` as a module expression coerced to the signature `sig type t end`. Formally, we introduce the definition

`module x =` (m `: sig type t end`)

Let us define another module y defining also an abstract type t:

$$\texttt{module y} = (m' : \texttt{sig type t end})$$

It might be that $m = m'$, so that the implementation of x.t would accidentally be the same that the one of y.t. Because of the "self" rule, we would then have

$$\texttt{x.t} = (m : \texttt{sig type t end}).\texttt{t} = \texttt{y.t}$$

Whereas we wished the implementation of x.t and y.t were irrelevant: type abstraction has been lost !

In order to prevent this, [HL94] restricts the use of the self rule so that it only applies to values, and [Ler95] restricts module paths to a syntactic fragment of module expressions.

We think that the authors of the aforementioned papers missed the following point: the coercion operation is a non-applicative notion coming from a too operational point of view since it *generates* new types; instead, *module definitions* should be thought of as abstractions *in nature*.

That is, when we bind an identifier x to a module expression m, we just want to define a module x having a given signature, but we do not need the addition of this definition to make x be convertible with m. So, the definition of a module identifier x should be a module expression m plus a signature M such that m has module type M. After the addition of this definition to the environment, the module expression m is forgotten by the type system, even if its most general type is more precise than M. That is, from the type system point of view, the definition of x is equivalent to the declaration of a module variable x of type M.

As we express module type restriction at module definition, we do not need a coercion operator. This way for defining a module also gives a simple status to type abstraction: outside the scope of its definition, a type is abstract if and only if it has been hidden at the time of the definition of its enclosing structure. In other words, type abstraction and type definition are two distinct concerns, and therefore, one only need one way for defining a type (on the contrary there are two constructs for defining a new type in SML). Hence, type abstraction is no longer achieved through the generation of a unique new type at declaration time but through type abstraction at module definition time. In fact, this point of view is not really new: thus, in Modula2 [Wir83], a module is equal to a compilation unit, and there is only one way to define a new type; whether the definition has to be exported is specified in an interface file.

3 The Module Calculus

Let us synthesize the main features of our proposition.

- As in [Ler95], our module calculus is stratified, and is therefore quite independent of the base language;
- as in [HL94], our calculus should not have any syntactic restrictions on access paths;

- contrarily to [HL94] and [Ler95], there is no coercion operator for modules;
- 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).[1]

It is to be noticed that the base language of our calculus is left mostly unspecified, as in [Ler94,Ler95]. That is, few assumptions are made about it, and therefore, it is not dependent of a particular language. We only need a language distinguishing types and values where functions are first-class. Our calculus does not account for concrete type definitions nor for recursive definitions of ML. In fact, an ML concrete type definition has also a generative behavior that our calculus can not account for. However, if concrete types were given a first-class status and a non-generative semantics,[2] our calculus could perfectly account for them. As usual, recursive definitions can be accounted for *via* the use of a fixpoint operator.

3.1 Syntax

The Variable Clashes Problem First, we would like to point out a subtle problem that happens when instantiating a functor type: as in λ-calculus, $(\lambda y.x\ z)\{x \leftarrow y\}$ is not $(\lambda y.y\ z)$, if

$$f : \texttt{functor}(x : \ldots)\texttt{sig type}\ y = \ldots\ \texttt{type}\ z = x.n\ \texttt{end}$$

then $(f\ y)$ is not of type

$$\texttt{sig type}\ y = \ldots\ \texttt{type}\ z = y.n\ \texttt{end}$$

The usual solution in λ-calculus is capture-avoiding substitutions that rename binders if necessary. Here, a field of a structure can not be renamed since we want to be able to access components of a structure by their names. In fact, the problem is a confusion between the notion of component name and binder. Therefore, we modify the syntax of declarations and specifications: declarations and specifications shall be of the form $x\ \texttt{as}\ y = \ldots$ (or $x\ \texttt{as}\ y : \ldots$ or $x\ \texttt{as}\ y : \ldots = \ldots$), the first identifier being the name of the component and the second one its binder (this syntax has been proposed in [HL94]). From inside a structure or signature, the component is referred by its binder, and from outside, it is referred by its name. Then, we avoid name clashes through renamings of binders. The old syntax $x = t$ should be only a syntactic sugar for $x\ \texttt{as}\ x = t$. For instance, the following module definition:

---

[1] The reader may notice that this in fact was done in [Ler94]; unfortunately, it seems that Leroy did not realize this point ensured type abstraction without any syntactic restriction on projections.

[2] As far as we know, giving a non-generative semantics to concrete type definitions only complicates type inference a little bit.

```
module X =
 struct
  module Y =
   struct
    type t = int
    type u = t -> t
   end
  type v = Y.u -> Y.t
 end
```

could be written:

```
module X =
 struct
  module Y as Y' =
   struct
    type t as t' = int
    type u = t' -> t'
   end
  type v = Y'.u -> Y'.t
 end
```

Reductions As we want to study 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 least relation on module expressions such that

$$(\texttt{functor}\,(x : M)\,m_1)\,(m_2) \to_\mu m_1\{x \leftarrow m_2\}$$

$$m_1 \to_\mu m_1' \Rightarrow (m_1\ m_2) \to_\mu (m_1'\ m_2)$$

$$m_2 \to_\mu m_2' \Rightarrow (m_1\ m_2) \to_\mu (m_1\ m_2')$$

$$m \to_\mu m' \Rightarrow \texttt{functor}\,(x : M)\,m \to_\mu \texttt{functor}\,(x : M)\,m'$$

We define μ-equivalence as the least equivalence relation including the μ-reduction.

3.2 Typing Rules

We assume given base-language dependent rules defining typing judgments $E \vdash e : \tau$ and $E \vdash \tau : \texttt{type}$. We make use of the following judgments:

$E \vdash$ **ok**	the context E is well-formed
$E \vdash M$ **modtype**	module type M is well-formed
$E \vdash m : M$	module expression m has type M
$E \vdash s : S$	structure body s has type S
$E \vdash M_1 <: M_2$	module type M_1 is a subtype of M_2
$E \vdash \tau \approx \tau'$	type τ is convertible to τ'

We define the last four figure 1.

In these rules we make use of four auxiliary definitions. Firstly, as in [HL94] the overline function ($\overline{D}$) merely strips off the field name of a signature component D. Secondly, the function $Names$ gives the set of fields appearing in a signature body. Thirdly, the BV function gives the set of couples (names,identifier) appearing in a given signature body, and the set of binders appearing in a given environment. Fourthly, as in [Ler94,Ler95], one of the rules for typing modules makes use of the strengthening M/m of a module type M by a module expression m: merely, the strengthening M/m of M replaces every abstract type t of M by a manifest type t equal to $m.t$; this rule is a way to express the "self" rule saying that every type is manifestly equal to itself.

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

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

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

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

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

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

$$\frac{E \vdash e : \tau\ \ E; \texttt{val}\ v : \tau \vdash s : S\ \ w \notin Names(S)}{E \vdash (\texttt{val}\ w\ \texttt{as}\ v = e; s) : (\texttt{val}\ w\ \texttt{as}\ v : \tau; S)}$$

$$\frac{E \vdash \tau\ \texttt{type}\ \ u \notin Names(S)\ \ E; \texttt{type}\ t = \tau \vdash s : S}{E \vdash (\texttt{type}\ u\ \texttt{as}\ t = \tau; s) : (\texttt{type}\ u\ \texttt{as}\ t = \tau; S)}$$

$$\frac{E \vdash m : M\ \ y \notin Names(S)\ \ E; \texttt{module}\ x : M \vdash s : S}{E \vdash (\texttt{module}\ y\ \texttt{as}\ x : M = m; s) : (\texttt{module}\ y\ \texttt{as}\ x : M; S)}$$

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

$$\frac{\begin{array}{c}E \vdash \texttt{sig}\ D'_1; \ldots; D'_m\ \texttt{end}\ \textbf{modtype}\ \ E \vdash \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end}\ \textbf{modtype} \\ \sigma : \{1, \ldots, m\} \to \{1, \ldots, n\}\ \ \forall i \in \{1, \ldots, m\}\ \ E; \overline{D}_1; \ldots; \overline{D}_n \vdash D_{\sigma(i)} <: D'_i\end{array}}{E \vdash \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} <: \texttt{sig}\ D'_1; \ldots; D'_m\ \texttt{end}}$$

$$\frac{E \vdash M_2 <: M_1\ \ E; \texttt{module}\ x : M_2 \vdash M'_1 <: M'_2}{E \vdash \texttt{functor}\,(x : M_1)\,M'_1 <: \texttt{functor}\,(x : M_2)\,M'_2}$$

$$\frac{E \vdash M <: M'}{E \vdash \texttt{module}\ x\ \texttt{as}\ y : M <: \texttt{module}\ x\ \texttt{as}\ y : M'}$$

$$\frac{E \vdash \tau \approx \tau'}{E \vdash \texttt{val}\ w\ \texttt{as}\ v : \tau <: \texttt{val}\ w\ \texttt{as}\ v : \tau'}$$

$$\frac{E \vdash M <: M'}{E \vdash \texttt{module}\ y\ \texttt{as}\ x : M <: \texttt{module}\ y\ \texttt{as}\ x : M'}$$

$$\frac{E \vdash \textbf{ok}}{E \vdash \texttt{type}\ u\ \texttt{as}\ t[= \tau] <: \texttt{type}\ u\ \texttt{as}\ t} \qquad \frac{E \vdash t \approx \tau'}{E \vdash \texttt{type}\ u\ \texttt{as}\ t[= \tau] <: \texttt{type}\ u\ \texttt{as}\ t = \tau'}$$

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

$$\frac{m =_\mu m'\ \ E \vdash m.t\ \texttt{type}\ \ E \vdash m'.t\ \texttt{type}}{E \vdash m.t \approx m'.t}$$

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

$$\frac{E \vdash m : \texttt{sig}\ S_1; \texttt{type}\ t\ \texttt{as}\ u{=}\tau; S_2\ \texttt{end}}{E \vdash m.t \approx \tau\{n \leftarrow m.n' \mid (n', n) \in BV(S_1)\}}$$

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

Fig. 1. Typing rules

3.3 Module Reductions

We have the following results:

Theorem 1 (Confluence of μ-reduction). *The μ-reduction is confluent*

Proof. The standard Tait and Martin-Löf's method applies.

Theorem 2 (Subject reduction for μ-reduction). *If $E \vdash m : M$, and $m \to_\mu m'$, then $E \vdash m' : M$.*

Proof. The proof is done the usual way, that is we prove a substitution lemma, and study the possible types of functors.

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

Proof. In fact, the proof is quite easy through a translation of module expressions to simply-typed lambda-calculus extended with records and records subtyping.

Notice that this theorem does not rely on any assumption about normalization with respect to reductions of the base language. Indeed, this result only means that a module expression can be reduced until no *module* reduction can take place; independently, the base language reductions may or may not terminate.

3.4 $\mu\rho$-Reductions

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 least context-stable relation such that

$$\begin{array}{l}
\texttt{struct } S_1\texttt{;type } t \texttt{ as } t' = \tau\texttt{;} S_2 \texttt{ end.} t \qquad \to_\rho \\
\qquad \tau\{n' \leftarrow \texttt{struct } S_1\texttt{;type } t \texttt{ as } t' = \tau\texttt{;} S_2 \texttt{ end.} n \mid (n, n') \in BV(S_1)\} \\
\texttt{struct } S_1\texttt{;val } v \texttt{ as } v' = e\texttt{;} S_2 \texttt{ end.} v \qquad \to_\rho \\
\qquad e\{n' \leftarrow \texttt{struct } S_1\texttt{;val } v \texttt{ as } v' = e\texttt{;} S_2 \texttt{ end.} n \mid (n, n') \in BV(S_1)\} \\
\texttt{struct } S_1\texttt{;module } x \texttt{ as } x' : M = m\texttt{;} S_2 \texttt{ end.} x \to_\rho \\
m\{n' \leftarrow \texttt{struct } S_1\texttt{;module } x \texttt{ as } x' : M = m\texttt{;} S_2 \texttt{ end.} n \mid (n, n') \in BV(S_1)\}
\end{array}$$

A 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. This result shows that the extension of the base language with modules is "conservative". Indeed, in a proof language, this result implies 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. In the following subsection, we address the question to know whether the modular program and the base-language program have the same semantics.

3.5 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.*

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

3.6 Type Inference

In order to obtain a type inference algorithm, we define an inference system $\vdash_{\mathcal{A}}$ which runs in a deterministic way for a given module expression. A notion of δ-reduction of a type is defined in order to compare terms through $\mu\delta$-normalization. We give in figure 2 the rule for application, which replaces the previous one and the previous rule for strengthening, plus the rules for type comparison and type reduction.

Because of size limitation, we have to summarize in few words the main results about our type inference system: it is sound, complete, and leads to a type inference algorithm; also every well-typed expression has a principal type which whereas the system of [Ler95] does not enjoys the principal type property [Ler96a].

4 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, and allows the design of a well-understood module system for lazy ML or Haskell. Also, we establish that our module system is "conservative": 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

Application rule

$$\frac{E \vdash_A m_1 : \texttt{functor}\,(x : M)\,M'\ E \vdash_A m_2 : M''\ E \vdash_A M''/m_2 <: M}{E \vdash_A m_1\,(m_2) : M'\{x \leftarrow m_2\}}$$

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

$$\frac{m =_\mu m'\ \ E \vdash_A m.t\ \texttt{type}\ \ E \vdash_A m'.t\ \texttt{type}}{E \vdash m.t \approx m'.t} \qquad \frac{E \vdash_A \tau \rightarrow_\delta \tau'}{E \vdash \tau \approx \tau'}$$

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

Type reduction ($E \vdash_A \tau \rightarrow_\delta \tau'$):

$$E_1; \texttt{type}\ t{=}\tau; E_2 \vdash_A t \rightarrow_\delta \tau$$

$$\frac{E \vdash_A m : \texttt{sig}\ \ S_1; \texttt{type}\ t\ \texttt{as}\ u{=}\tau; S_2\ \ \texttt{end}\ \ m \text{ is in } \mu\text{-normal form}}{E \vdash_A m.t \rightarrow_\delta \tau\{n \leftarrow m.n' \mid (n', n) \in BV(S_1)\}}$$

Fig. 2. Type inference

type inference [Ler96a]. On the contrary, in our system, every module expression enjoys a principal type, and type inference is decidable.

The replacement of type generativity by abstraction at definition gives a less operational account for type abstraction. Moreover type abstraction is preserved (the representation independence proof of [Ler95] is adaptable to our system). In Jones's proposal for modular programming [Jon96], some type safety brought by the type system is lost since two modules are considered equal by the type system provided they declare the same types, independently of the implementation of the functions they declare. For instance two modules that export a type together with an ordering shall be equal provided the same type is exported. On the contrary, our proposal does not suffer this problem, since module comparison is done through μ-reduction.

We think our system improves the type-theoretical understanding of modules. 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], which is quite easy, in spite of the interaction of β-reduction with typing, in order to have a modular proof language well-suited to proving modular programs [Cou97]. Since the Calculus of Construction is both a programming language and a proof language, this have the advantage to provide a unified framework, simpler than the Extended ML approach [San90] because of the inherent complexity of the semantics of the SML module system. We also believe our system may help in designing a safe and powerful module system for Elf.

Acknowledgements

We would like to thank Philippe Audebaud and Xavier Leroy for their comments on this work.

References

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Programming Language Design and Implementation 1994*, pages 13–23. ACM Press, 1994.

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

[Cou97] Judicaël Courant. A module calculus for pure type systems. In *Typed Lambda Calculi and Applications 97*, LNCS. Springer-Verlag, 1997.

[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.

[HLPR94] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. Technical Report CMU-CS-94-116, Carnegie-Mellon University, 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.

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

[Ler96a] Xavier Leroy, 1996. Private Communication.

[Ler96b] Xavier Leroy. A syntactic theory of type generativity and sharing. To appear in *Journal of Functional Programming*, 1996.

[NL96] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. In *second symposium on Operating Systems Design and Implementation*, 1996.

[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.

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