A module calculus for Pure Type Systems. (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 for Pure Type Systems

Preliminary version

Judicaël Courant

October 96

Research Report N$^{\mathrm{o}}$ 96-31

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 for Pure Type Systems

Preliminary version

Judicaël Courant

October 96

Abstract

Several proof-assistants rely on the very formal basis of Pure Type Systems. However, some practical issues raised by the development of large proofs lead to add other features to actual implementations for handling namespace management, for developing reusable proof libraries and for separate verification of distincts parts of large proofs. Unfortunately, few theoretical basis are given for these features. In this paper we propose an extension of Pure Type Systems with a module calculus adapted from SML-like module systems for programming languages. Our module calculus gives a theoretical framework addressing the need for these features. We show that our module extension is conservative, and that type inference in the module extension of a given PTS is decidable under some hypotheses over the considered PTS.

Keywords: Module systems, PTS, higher-order type systems, subject-reduction, normalization, type inference

Résumé

Plusieurs assistants de preuves sont fondés sur les Systèmes de Types Purs (PTS). Cependant, des considérations pratiques provenant du développement de grandes preuves conduisent à ajouter aux implémentations des mécanismes permettant une gestion rationnelle des noms, le développement de bibliothèques de preuves réutilisables, et la vérification séparée des différentes parties d'un gros développement. Alors que la correction des PTS utilisés est théoriquement bien fondé, ces mécanismes sont en revanche peu étudiés, alors qu'ils peuvent mettre en péril la correction de l'ensemble de l'outil de démonstration. Pour répondre à ce problème, nous proposons dans ce rapport une extension des PTS par un système de modules similaire à celui de SML pour le langage de programmation ML. Notre système de modules donne un cadre théorique rigoureux pour l'étude des mécanismes que nous avons cités. Nous montrons que l'extension proposée est conservative, et que l'inférence de type est décidable moyennant quelques hypothèses raisonnables sur le PTS considéré.

Mots-clés: Systèmes de modules, PTS, systèmes de types d'ordre supérieur, autoréduction, normalisation, inférence de type

A module calculus for Pure Type Systems*

Preliminary version

Judicaël Courant

October 96

1 Introduction

The notion of Pure Type Systems has been first introduced by Terlouw and Berardi [Bar91]. These systems are well-suited for expressing specifications and proofs and are the basis of several proof assistants [CCF+95, Pol94, MN94, HHP93]. However, there is actually a gap between PTS and the extensions needed for proof assistants. Indeed, PTS are well-suited to type-theoretic study, but lack some features that a proof-assistant needs.

A first practical expectation when specifying and proving in a proof assistant is for definitions. Making a non-trivial proof or even a non-trivial specification in a proof assistant is often a long run task that would be impossible if one could not bind some terms to a name. The meta-theoretical study of definitions and their unfolding, although not very difficult is far from being obvious; it has been achieved for instance in [SP94].

Another highly expectable feature when developing large proofs is for a practical namespace management. Indeed, it is often difficult to find a new significant name for each theorem. In proof-assistants where proofs can be split across several files, a partial solution is to represent names as prefixed by the name of the file they are defined in. Then, the user may either refer to a theorem by its long name, or give only the suffix part which refers to the last loaded theorem with this suffix.

Another one is the ability to parameterize a whole theory with some axioms. For instance, when defining and proving sorting algorithms, it is very convenient to have the whole theory parameterized with a set A, a function $ord : A \rightarrow A \rightarrow bool$, and three axioms stating that ord is reflexive, antisymmetric, transitive, total and decidable. This feature is implemented in the Coq proof-assistant through the sectioning mechanism [CCF+95]. In a given section, one may declare axioms or variables and use them. When the section is closed, these axioms and variables are discharged. That is, every theorem is parameterized by these hypothesis and variables. Thus, one does not have to explicitly parameterize every theorem by these hypothesis and variables.

However, this sectioning mechanism is not a definite answer. Indeed, it does not allow to instantiate a parameterized theory. For instance, once the theory of sorting algorithms has been proved, if one wants to use this theory for a given set and an ordering, one has to give the five parameters describing the ordering each time he needs to use any of the results. In order to have a more convenient way to refer to these results, we have to imagine a mechanism allowing the instantiation of several results at once.

Finally, proof assistants also raise the problem of separate verification. Thus, in proof-assistants such as Coq, the verification of standard proof-libraries can take several hours. For the user, this is annoying if the proof-assistant needs to check them each time the user references them. Therefore, a feature allows to save and restore the global state of the proof-assistant on disk ; thus, standard libraries are checked once, then the corresponding state is saved, and users start their sessions with this state. But it is not possible to save all available libraries in a given state, because they would require too much memory. Rather, one would like to have a way to load only required libraries, but at a reasonable speed. Recently, the Lego and the Coq proof-assistants allowed to put theories they check into a compiled form. Such compiled forms can be loaded very fast — several seconds instead of several minutes or hours.

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

But the possibility of saving proofs in compiled forms is not a true separate verification facility. In fact, we lack a notion of *specification* of a proof. Such a notion is desirable for three reasons. The first one is this would provide a convenient way to describe what is proved in a given proof development. The second one is the user may like to give only a specification of a theory he needs to make a proof, in order to make his main proof first, then prove the specification he needed. The third one is that would help in making proofs robust with respect to changes: indeed, it is sometimes difficult to predict whether a change in a proof will break proofs depending on it, since there is no clear notion of the specification exported by a given file.

Some theorem provers already address some of these issue. Thus IMPS [FGT95] implements Bourbaki's notion of structures and theories [Bou70], allowing to instantiate a general theory on a given structure at once, getting every instantiations of theorems. Unfortunately, this notion is well-suited in a set-theoretic framework but less in a type-theoretic one.

The Standard ML programming language has a very powerful module system [Mac85] that allows the definition of parametric modules and their composition, although it does not support true separate compilation. This module system was adapted to the Elf implementation of LF [HP92]. However, only the part of the SML module system that was well-understood from the semantic and pragmatic point of view was adapted, hence leaving out significant power of SML. For instance, the `sharing` construct of SML had to be ruled out. This is annoying since this construct allows to express that two structures share a given component. For instance, it may be useful to make a theory over groups and monoïds that share the same base set.[1]

Recent works on module systems however bring hope: Leroy [Ler94, Ler95], Harper and Lillibridge [HL94] presented "cleaner" variants of the SML module system, allowing true separate compilation since only the knowledge of the type of a module is needed in order to typecheck modules using it. Unfortunately, no proof of correctness was given for any of these system, thus preventing us to be sure their adaptation to a proof system would not lead to inconsistency. We gave one in a variant of these systems in [Cou96].

However adaptation of these module systems to Pure Type Systems raises the problem of dealing with β-equivalence that appears in the conversion rule of PTS. In this paper, we give an adaptation of the system of [Cou96] to Pure Type Systems. This system applies to the LF logical framework, the Calculus of Construction [CH88], the Calculus of Constructions extended with universes [Luo89]. We do not deal with the problem of adding inductive types to these systems, but the addition of inductive types as first-class objects should not raise any problem as our proposal is quite orthogonal to the base language: as few properties of β-reduction were needed to prove our results, they should also be true in a framework with inductive types and the associated ι-reduction.

The remaining of this paper is organized as follows: we give in section 2 an informal presentation of the desired features for a module system. Then, in section 3, we expose formally our system. In section 4 we give its meta-theory. We compare our system with other approaches in section 5. Finally, we give possible directions for future work and conclude in section 6.

2 Informal presentation

In order to solve the problem of namespace management, we add to PTS the notion of *structure*, that is, package of definitions. An environment may now contain structures declarations. These structures can even contain sub-structures, which may help in structuring the environment. In fact, many mathematical structures own sub-structures. Thus, the polynomial ring $A[X]$ over a ring A may be defined as a structure having A as a component; a monoïd homomorphism may be defined as a structure having the domain and the range monoïds as components; *et cetera*.

In order to address the issue of robustness of proofs with respect to changes, we introduce a notion of specification. We require every module definition be given together with a specification. A specification for a structure is a declaration of the objects the module should export, together with their types, and possibly their definitions. The specification of a structure is called a *signature* of this structure. Then, the only thing the type-checker knows about a module in a given environment is its specification. The correction of a development is ensured as soon as for every specification, a module matching this specification is given.

[1] The mathematical structure of rings is defined as the data of a group and a monoïd that share the same base set, and verify some other conditions (distributivity).

Let us consider an example. Assume we want to work in the Calculus of Constructions, extended with an equality defined on any set A, $=_A$. Assuming $\diamond$ is any given term of type `Set`, we can define a monoïd structure on $\diamond \to \diamond$ in the following way:

$$
\begin{array}{lcl}
\texttt{module}\ M & : & \texttt{sig} \\
 & & \quad \begin{array}{lcl}
E & : & \texttt{Set} = \diamond \to \diamond \\
e & : & E \\
op & : & E \to E \to E \\
assoc & : & \forall x, y, z : E.(op\ (op\ x\ y)\ y) =_E (op\ x\ (op\ y\ z)) \\
left\_neutral & : & \forall x : E.(op\ e\ x) =_E x \\
right\_neutral & : & \forall x : E.(op\ x\ e) =_E x
\end{array} \\
 & & \texttt{end} \\
 & = & \texttt{struct} \\
 & & \quad \begin{array}{lcl}
base & = & \diamond \\
E & = & base \to base \\
e & = & \lambda x : base.x \\
op & = & \lambda f, g : base \to base.\lambda x : base.(f\ (g\ x)) \\
assoc & = & \ldots \\
left\_neutral & = & \ldots \\
right\_neutral & = & \ldots
\end{array} \\
 & & \texttt{end}
\end{array}
$$

This definition adds to the environment a module M of the given signature. Signatures are introduced by the keyword `sig`, structures by `struct`. Both are ended by the keyword `end`.

From inside the definition, components are referred to as E, e, op ; from outside, they must be referred to as $M.E$, $M.e$, $M.op$, ... Notice that *base* is not visible outside the definition of M since it is not declared in the signature. Only the definition of $M.E$ is known outside the module definition, so that for instance no one can take advantage of a particular implementation of op. The declaration $E : \texttt{Set} = \diamond \to \diamond$ is said to be *manifest* since it gives the definition of E.

The naming convention $M.S.c$ might become heavy when working on a given module. Therefore, in the SML module system, there is an `open` construct such that after an `open` M, any component c of M can be referred to as c instead of $M.c$. However, this is only syntactic sugar, so we will not consider it in our theoretical study.

Since we wish to handle parameterized theories, we extend the module language in order to allow parameterized modules. Then, one can develop for instance a general theory T of monoïds parameterized by a generic monoïd structure, then define the module T_M of the theory of the monoïd M. Parameterized modules are built through the `functor` keyword, that is the equivalent of a λ-abstraction at the module level, and of a $\forall$-quantification at the module type level:

$$
\begin{array}{l}
\texttt{module}\ T \\
\ :\ \texttt{functor}(\mathcal{M} : \texttt{<<}\mathit{monoid\ signature}\texttt{>>}) \\
\quad \texttt{sig} \\
\quad\quad unique\_left\_neutral \ :\ \forall x : \mathcal{M}.E.(\forall y : \mathcal{M}.E.(\mathcal{M}.op\ x\ y) =_{\mathcal{M}.E} y) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \to (x =_{\mathcal{M}.E} \mathcal{M}.e) \\
\quad\quad \vdots \\
\quad \texttt{end} \\
\ =\ \texttt{functor}(\mathcal{M} : \texttt{<<}\mathit{monoid\ signature}\texttt{>>}) \\
\quad \texttt{struct} \\
\quad\quad unique\_left\_neutral \ =\ \ldots \\
\quad\quad \vdots \\
\quad \texttt{end}
\end{array}
$$

Then one can instantiate the general theory on a given module as follows:

$$
\begin{array}{ll}
\texttt{module } T_M & \\
\quad : \;\; \texttt{sig} & \\
\qquad \mathit{unique\_left\_neutral} \;\; : \;\; \forall x : M.E.(\forall y : M.E.(M.op\ x\ y) =_{M.E} y) \to (x =_{M.E} M.e) & \\
\qquad \vdots & \\
\qquad \texttt{end} & \\
\quad = \;\; (T\ M) &
\end{array}
$$

Functors are also interesting for the construction of mathematical structures. For instance, the product monoïd of two generic monoïds can be defined easily through a functor, then instantiated on actual monoïds.

Finally, before we give a formal definition of our system, it should be noticed that a name conflict can appear when instantiating a functor: as in λ-calculus, $(\lambda y.x\ z)\{x \leftarrow y\}$ is not $(\lambda y.y\ z)$, if

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

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

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

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 $x \triangleright y = \ldots$ (or $x \triangleright y : \ldots$ or $x \triangleright y : \ldots = \ldots$), the first identifier being the name of the component and the second one its binder. This syntax has been proposed by Harper and Lillibridge in [HL94]. They suggested pronouncing "$\triangleright$" as "as". 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 by capture-avoiding substitutions. For instance, the monoïd previously defined could be written:

$$
\begin{array}{lllllll}
\texttt{module } M & : & \texttt{sig} & & & & \\
 & & \quad E & \triangleright & E' & : & \texttt{Set = } \diamond \to \diamond \\
 & & \quad e & \triangleright & e' & : & E' \\
 & & \quad op & \triangleright & op' & : & E' \to E' \to E' \\
 & & \quad assoc & \triangleright & assoc' & : & \forall x,y,z : E'.(op'\ (op'\ x\ y)\ y) =_{E'} (op'\ x\ (op'\ y\ z)) \\
 & & \quad left\_neutral & \triangleright & left\_neutral' & : & \forall x : E'.(op'\ e'\ x) =_{E'} x \\
 & & \quad right\_neutral & \triangleright & right\_neutral' & : & \forall x : E'.(op'\ x\ e') =_{E'} x \\
 & & \texttt{end} & & & & \\
 & = & \ldots & & & &
\end{array}
$$

Of course, we shall allow $x : t$ as a syntactic sugar for $x \triangleright x : t$ (similarly for $x = t$).

3 A module calculus

We now formalize our previous remarks in a module calculus derived from the propositions of [Ler94, Ler95, HL94, Cou96].

3.1 Syntax

Terms :

$e ::= v$	identifier
$\mid m.v$	access to a value field of a structure
$\mid (e_1\ e_2)$	application
$\mid \lambda v{:}e_1.e_2$	λ-abstraction
$\mid \forall v{:}e_1.e_2$	universal quantification

Module expressions :

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

Structure body :

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

Structure component :

$d ::= \texttt{term}\ v_1 \triangleright v_2\ \texttt{=}\ e$	term definition
$\mid \texttt{module}\ x_1 \triangleright x_2\ :\ M\ \texttt{=}\ m$	module definition

Module type :

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

Signature body :

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

Signature component :

$D ::= \texttt{term}\ v_1 \triangleright v_2\ :\ e$	term declaration
$\mid \texttt{term}\ v_1 \triangleright v_2\ :\ e_1\ \texttt{=}\ e_2$	manifest term declaration
$\mid \texttt{module}\ x_1 \triangleright x_2\ :\ M$	module declaration

Environments :

$E ::= \epsilon$	empty environment
$\mid v\ :\ e$	term declaration
$\mid v\ :\ e\ \ \texttt{=}\ \ e'$	term definition
$\mid \texttt{module}\ x\ :\ M$	module declaration

Notice that this syntax is an extension of the syntax of pre-terms in PTS, and that this extension is quite orthogonal to the syntax of these pre-terms. Since we intend to study the reductions of the module calculus, we shall distinguish β-reductions at the level of the base-language calculus and at the level of the module calculus. Therefore we call μ-reduction the β-reduction at the level of module system. That is, μ-reduction is the least context-stable relation on the syntax such that $((\texttt{functor}(x : M)m_1)\ m_2) \rightarrow_\mu m_1\{x_i \leftarrow m_2\}$. We define μ-equivalence as the least equivalence relation including the μ-reduction.

As for β-reduction, we shall consider it as the least relation on terms such that

$$(\lambda v : e_1.e_2\ e_3) \rightarrow_\beta e_2\{v \leftarrow e_3\}$$

$$\begin{array}{cc}
e_1 \rightarrow_\beta e_1' \Rightarrow (e_1\ e_2) \rightarrow_\beta (e_1'\ e_2) & e_2 \rightarrow_\beta e_2' \Rightarrow (e_1\ e_2) \rightarrow_\beta (e_1\ e_2') \\
e_1 \rightarrow_\beta e_1' \Rightarrow \lambda v : e_1.e_2 \rightarrow_\beta \lambda v : e_1'.e_2 & e_2 \rightarrow_\beta e_2' \Rightarrow \lambda v : e_1.e_2 \rightarrow_\beta \lambda v : e_1.e_2' \\
e_1 \rightarrow_\beta e_1' \Rightarrow \forall v : e_1.e_2 \rightarrow_\beta \forall v : e_1'.e_2 & e_2 \rightarrow_\beta e_2' \Rightarrow \forall v : e_1.e_2 \rightarrow_\beta \forall v : e_1.e_2'
\end{array}$$

That is, β-reduction of a term can not be performed inside any module expression.

Context rules ($E \vdash \mathsf{ok}$):

$$\epsilon \vdash \mathsf{ok}$$

$$\frac{E \vdash e : \sigma \quad \sigma \in \mathcal{S} \quad v \notin E}{E; v : e \vdash \mathsf{ok}}$$

Typing rules ($E \vdash e : e'$):

$$\frac{E; v : e; E' \vdash \mathsf{ok}}{E; v : e; E' \vdash v : e}$$

$$\frac{E \vdash \mathsf{ok} \quad (c, \sigma) \in \mathcal{A}}{E \vdash c : \sigma}$$

$$\frac{E \vdash e : \sigma_1 \quad E; v : e \vdash e' : \sigma_2 \quad (\sigma_1, \sigma_2, \sigma_3) \in \mathcal{R}}{E \vdash \forall v : e.e' : \sigma_3}$$

$$\frac{E \vdash e_1 : \forall v : e.e' \quad E \vdash e_2 : e}{E \vdash (e_1\ e_2) : e'\{v \leftarrow e_2\}}$$

$$\frac{E; v : e \vdash e' : e'' \quad E \vdash \forall v : e.e'' : \sigma \quad \sigma \in \mathcal{S}}{E \vdash \lambda v : e.e' : \forall v : e.e''}$$

$$\frac{E \vdash e : e' \quad E \vdash e'' : \sigma \quad \sigma \in \mathcal{S} \quad E \vdash e' \approx e''}{E \vdash e : e''}$$

Term equivalence ($E \vdash e \approx e'$):

$$\frac{e =_\beta e' \quad E \vdash \mathsf{ok}}{E \vdash e \approx e'} \qquad \frac{e =_\alpha e' \quad E \vdash \mathsf{ok}}{E \vdash e \approx e'}$$

(congruence rules omitted)

Figure 1: PTS rules

3.2 Typing rules

Let $\mathcal{S}$ a set of constants called the sorts, $\mathcal{A}$, a set of pair (c, σ) where c is a constant and $\sigma \in \mathcal{S}$, and $\mathcal{R}$ a set of triples of elements of $\mathcal{S}$. The Pure Type System (PTS) determined by the specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is defined in figure 1. Three kinds of judgments are defined: a given environment is well-formed, a given term is of a given type, and two given terms are convertible. In order to build a module system over this PTS, we add rules given figures 2 and 3, that define the following new judgments:

$$\begin{array}{lll}
E \vdash M\ \mathsf{modtype} & & \text{module type } M \text{ is well-formed} \\
E \vdash m : M & & \text{module expression } m \text{ has type } M \\
E \vdash M_1 <: M_2 & & \text{module type } M_1 \text{ is a subtype of } M_2 \\
E \vdash m = m' : M & & \text{considered as modules of type } M,\ m \text{ and } m' \text{ are defining equal terms}
\end{array}$$

In these rules we make use of the following definitions. The first one helps in introducing a field of a module in the environment, the second one gives the set of fields defined in a structure body and the third one gives the set of couples (names,identifier) appearing in a given structure:

$$\begin{array}{rcl}
\overline{\mathtt{term}\ v \triangleright w : e} & = & w : e \\
\overline{\mathtt{term}\ v \triangleright w : e = e'} & = & w : e = e' \\
\overline{\mathtt{module}\ x \triangleright y : M} & = & \mathtt{module}\ y : M
\end{array}$$

$$\begin{array}{rcl}
N(\mathtt{term}\ v \triangleright w = e; s) & = & \{v\} \cup N(s) \\
N(\mathtt{module}\ x \triangleright y : M = m; s) & = & \{x\} \cup N(s) \\
N(\epsilon) & = & \emptyset
\end{array}$$

$$\begin{array}{rcl}
BV(\epsilon) & = & \emptyset \\
BV(\mathtt{term}\ v \triangleright w : e[= e']; s) & = & \{(v, w)\} \cup BV(s) \\
BV(\mathtt{module}\ x \triangleright y : M; s) & = & \{(x, y)\} \cup BV(s) \\
BV(E; v : e[= e']) & = & \{v\} \cup BV(E) \\
BV(E; \mathtt{module}\ x : M) & = & \{x\} \cup BV(E)
\end{array}$$

Following [Ler94, Ler95], one typing rule for modules 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 even if the component v of a module m is declared as abstract, one knows that this component is equal to $m.v$, and may add this information to the type of m. The strengthening operation is defined as follows:

$$\begin{array}{rcl}
(\mathtt{sig}\ S\ \mathtt{end})/m & = & \mathtt{sig}\ S/m\ \mathtt{end} \\
(\mathtt{functor}(x : M_1)M_2)/m & = & \mathtt{functor}(x : M_1)(M_2/m(x)) \\
\epsilon/m & = & \epsilon \\
(D; S)/m & = & D/m; (S/m) \\
(\mathtt{term}\ v \triangleright w{:}e)/m & = & \mathtt{term}\ v \triangleright w{:}e{=}m.v \\
(\mathtt{term}\ v \triangleright w : e_1{=}e_2)/m & = & \mathtt{term}\ v \triangleright w : e_1{=}e_2 \\
(\mathtt{module}\ x \triangleright y : M)/m & = & \mathtt{module}\ x \triangleright y : (M/m.x)
\end{array}$$

4 Meta-theory

We now give our main theoretical results about our module extension: this extension is sound since it is conservative, and if type inference is possible in a PTS, it is possible in its module extension.

Context formation ($E \vdash \mathsf{ok}$):

$$\frac{E \vdash M\ \mathsf{modtype} \quad x \notin BV(E)}{E; \mathtt{module}\ x : M \vdash \mathsf{ok}} \qquad \frac{E \vdash e : e' \quad w \notin BV(E)}{E; w : e' = e \vdash \mathsf{ok}}$$

Module type and signature body formation ($E \vdash M\ \mathsf{modtype}$):

$$\frac{E \vdash \mathsf{ok}}{E \vdash \epsilon\ \mathsf{modtype}} \qquad \frac{E; \mathtt{module}\ x : M \vdash S\ \mathsf{modtype} \quad y \notin N(S)}{E \vdash \mathtt{module}\ y \triangleright x : M; S\ \mathsf{modtype}}$$

$$\frac{E; v : e \vdash S\ \mathsf{modtype} \quad w \notin N(S)}{E \vdash \mathtt{term}\ w \triangleright v : e; S\ \mathsf{modtype}} \qquad \frac{E; v : e = e' \vdash S\ \mathsf{modtype} \quad w \notin N(S)}{E \vdash \mathtt{term}\ w \triangleright v : e = e'; S\ \mathsf{modtype}}$$

$$\frac{E \vdash S\ \mathsf{modtype}}{E \vdash \mathtt{sig}\ S\ \mathtt{end}\ \mathsf{modtype}} \qquad \frac{E \vdash M\ \mathsf{modtype} \quad x \notin BV(E) \quad E; \mathtt{module}\ x : M \vdash M'\ \mathsf{modtype}}{E \vdash \mathtt{functor}(x : M)M'\ \mathsf{modtype}}$$

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

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

$$\frac{E; \mathtt{module}\ x : M \vdash m : M' \quad E \vdash \mathtt{functor}(x : M)M'\ \mathsf{modtype}}{E \vdash \mathtt{functor}(x : M)m : \mathtt{functor}(x : M)M'}$$

$$\frac{E \vdash m_1 : \mathtt{functor}(x : M)M' \ E \vdash m_2 : M}{E \vdash (m_1\ m_2) : M'\{x \leftarrow m_2\}} \qquad \frac{E \vdash m : M' \quad E \vdash M' <: M}{E \vdash m : M}$$

$$\frac{E \vdash m : M}{E \vdash m : M/m} \qquad \frac{E \vdash s : S}{E \vdash (\mathtt{struct}\ s\ \mathtt{end}) : (\mathtt{sig}\ S\ \mathtt{end})} \qquad \frac{E \vdash \mathsf{ok}}{E \vdash \epsilon : \epsilon}$$

$$\frac{E \vdash e : e' \quad v \notin BV(E) \quad E; v : e' = e \vdash s : S \quad w \notin N(s)}{E \vdash (\mathtt{term}\ w \triangleright v = e; s) : (\mathtt{term}\ w \triangleright v : e = e'; S)}$$

$$\frac{E \vdash m : M \quad x \notin BV(E) \quad E; \mathtt{module}\ x : M \vdash s : S \quad y \notin N(s)}{E \vdash (\mathtt{module}\ y \triangleright x : M = m; s) : (\mathtt{module}\ y \triangleright x : M; S)}$$

Figure 2: Typing rules

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

$$\frac{E \vdash M \ \mathsf{modtype} \quad E \vdash M' \ \mathsf{modtype} \quad M =_\alpha M'}{E \vdash M <: M'}$$

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

$$\frac{E \vdash M_2 <: M_1 \quad E; \mathtt{module}\ x : M_2 \vdash M'_1 <: M'_2}{E \vdash \mathtt{functor}(x : M_1) M'_1 <: \mathtt{functor}(x : M_2) M'_2} \qquad \frac{E \vdash M <: M'}{E \vdash \mathtt{module}\ x \ \triangleright\ y : M <: \mathtt{module}\ x \ \triangleright\ y : M'}$$

$$\frac{E \vdash e \approx e'}{E \vdash \mathtt{term}\ v \ \triangleright\ w : e[= e''] <: \mathtt{term}\ v \ \triangleright\ w : e'} \qquad \frac{E \vdash e_1 \approx e'_1 \quad E \vdash w \approx e'_2}{E \vdash \mathtt{term}\ v \ \triangleright\ w : e_1[= e_2] <: \mathtt{term}\ v \ \triangleright\ w : e'_1 = e'_2}$$

Term equivalence ($E \vdash e \approx e'$):

$$\frac{E_1; w : e \ \mathtt{=}\ e'; E_2 \vdash \mathsf{ok}}{E_1; w : e \ \mathtt{=}\ e'; E_2 \vdash w \approx e'} \qquad \frac{E \vdash m : \mathtt{sig}\ \ S_1; \mathtt{term}\ v \ \triangleright\ w \ \mathtt{=}\ e; S_2 \ \ \mathtt{end}}{E \vdash m.v \approx e\{n \leftarrow m.n' \mid (n', n) \in BV(S_1)\}}$$

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

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

$$\frac{\begin{array}{l} E \vdash m : \mathtt{sig}\ \ D_1; \ldots; D_n \ \ \mathtt{end} \\ E \vdash m' : \mathtt{sig}\ \ D_1; \ldots; D_n \ \ \mathtt{end} \\ \forall i \in \{1, \ldots, n\} \quad D_i = \mathtt{term}\ v \ \triangleright\ w : e \ \mathtt{=}\ e' \Rightarrow E \vdash m.v \approx m'.v \\ D_i = \mathtt{module}\ x \ \triangleright\ y : M \Rightarrow E \vdash m.x \approx m'.x : M\{n \leftarrow m.n' \mid (n', n) \in BV(\mathtt{sig}\ \ D_1; \ldots; D_n \ \ \mathtt{end})\} \end{array}}{E \vdash m \approx m' : \mathtt{sig}\ \ D_1; \ldots; D_n \ \ \mathtt{end}}$$

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

Figure 3: Typing rules

4.1 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 we proved 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, modules expressions are very often in μ-normal form. Instead, we can study what happens when we unfold modules and terms definitions, that is, what happens when we add to μ-reduction the ρ-reduction defined as the least context-stable relation such that

$$\begin{array}{l} \texttt{struct}\ s_1;\texttt{term}\ v \ \triangleright\ w : e = e'; s_2\ \texttt{end}.v \\ \quad \rightarrow_\rho e\{n \leftarrow \texttt{struct}\ s_1;\texttt{type}\ v \ \triangleright\ w : e = e'; s_2\ \texttt{end}.n' \mid (n', n) \in BV(s_1)\} \\ \texttt{struct}\ s_1;\texttt{module}\ x \ \triangleright\ y : M = m; s_2\ \texttt{end}.x \\ \quad \rightarrow_\rho m\{n \leftarrow \texttt{struct}\ s_1;\texttt{module}\ x \ \triangleright\ y : M = m; s_2\ \texttt{end}.n' \mid (n', n) \in BV(s_1)\} \end{array}$$

In an empty environment, a $\mu\rho$-normalizing expression $\texttt{struct}\ s\ \texttt{end}.result$ normalizes to a term where no module construct appears: $\mu\rho$-normalization is a way to transform any expression of a Pure Type System extended with modules into a term of the corresponding Pure Type System.

We have the following results:

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

As a consequence of theorem 6, we have:

Theorem 7 (Conservativity of the module extension) *In the empty environment, a type T of a PTS is inhabited if and only if it is inhabited in its module extension.*

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 as usual (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 $E \vdash M$ $\texttt{modtype}$ and $E \vdash ((\texttt{functor}(x : M')m)\ x_i) : M$ then $E \vdash ((\texttt{functor}(x : M')m)\ x_i) \approx m : M$*

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

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 components ($m = n$ in the subtyping rule for signatures). Thus, $\texttt{sig term}\ v \ \triangleright\ w : f = e\ \texttt{term}\ t \ \triangleright\ u : f' = e'\ \texttt{end}$ is a subtype of $\texttt{sig term}\ v \ \triangleright\ w : f\ \texttt{term}\ t \ \triangleright\ u : f' = e'\ \texttt{end}$ but not of $\texttt{sig term}\ v \ \triangleright\ w : f\ \texttt{end}$.

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 built upon this set there is a constant of that type in this set), and interpret the terms of our calculus in such a way 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 proofs are not detailed because of their lengths.

Context rules ($E \vdash_{\mathcal{A}} \mathsf{ok}$):

$$\epsilon \vdash_{\mathcal{A}} \mathsf{ok}$$

$$\frac{E \vdash_{\mathcal{A}} e : \sigma \quad \sigma \in \mathcal{S} \quad v \notin E}{E; v : e \vdash_{\mathcal{A}} \mathsf{ok}}$$

Typing rules ($E \vdash_{\mathcal{A}} e : e'$):

$$\frac{E; v : e; E' \vdash_{\mathcal{A}} \mathsf{ok}}{E; v : e; E' \vdash_{\mathcal{A}} v : e}$$

$$\frac{E \vdash_{\mathcal{A}} \mathsf{ok} \quad (c, \sigma) \in \mathcal{A}}{E \vdash_{\mathcal{A}} c : \sigma}$$

$$\frac{E \vdash_{\mathcal{A}} e : \sigma_1 \quad E; v : e \vdash_{\mathcal{A}} e' : \sigma_2 \quad (\sigma_1, \sigma_2, \sigma_3) \in \mathcal{R}}{E \vdash_{\mathcal{A}} \forall v : e.e' : \sigma_3}$$

$$\frac{E \vdash_{\mathcal{A}} e_1 : \forall v : e.e' \quad E \vdash_{\mathcal{A}} e_2 : e'' \quad E \vdash_{\mathcal{A}} e \approx e''}{E \vdash_{\mathcal{A}} (e_1\ e_2) : e'\{v \leftarrow e_2\}}$$

$$\frac{E; v : e \vdash_{\mathcal{A}} e' : e'' \quad E \vdash_{\mathcal{A}} \forall v : e.e'' : \sigma \quad \sigma \in \mathcal{S}}{E \vdash_{\mathcal{A}} \lambda v : e.e' : \forall v : e.e''}$$

Term equivalence ($E \vdash_{\mathcal{A}} e \approx e'$):

$$\frac{e =_{\beta} e' \quad E \vdash \mathsf{ok}}{E \vdash_{\mathcal{A}} e \approx e'} \qquad \frac{e =_{\alpha} e' \quad E \vdash \mathsf{ok}}{E \vdash_{\mathcal{A}} e \approx e'}$$

(congruence rules omitted)

Figure 4: Type inference in a PTS

4.2 Type inference

In this subsection, we intend to give a type inference algorithm for our module extension. A sufficient condition for the type of a given term to be unique up to β-equivalence in a given PTS is that the PTS is *singly sorted*[2]. A sufficient condition in such PTS for type inference to be decidable is strong normalization of β-reduction, since term equivalence can then be decided by comparison of normal forms of terms. A type inference system for such PTS is given figure 4. Therefore, we shall in this subsection consider only singly-sorted PTS such that β-reduction is strongly normalizing.

It is to be noticed that the module extension preserves strong normalization of β-reduction.

In order to obtain a type inference algorithm, we provide in figures 5 and 6 an inference system which runs in a deterministic way for a given module expression except for term comparison $\approx$ (where two main rules plus reflexivity, symmetry, transitivity and context stability may filter the same terms). We show in subsection 4.2.1 that this system gives the most general type of a given module expression if this expression is well-typed. Then we give in subsection 4.2.2 a procedure to decide if two types of the base-language are in the $\approx$ comparison relation. Finally, we state in subsection 4.2.3 that this algorithm stops even if the given module is ill-typed.

The inference system is obtained from the one given figures 2 and 3 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.

4.2.1 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 $E \vdash_{\mathcal{A}} e \approx e'$ then $E \vdash e \approx e'$.*

[2] The PTS determined by the specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is said *singly-sorted* or *functional* if and only if the relations $c \mapsto \sigma$ for $(c, \sigma) \in \mathcal{A}$ and $(\sigma_1, \sigma_2) \mapsto \sigma_3$ for $(\sigma_1, \sigma_2, \sigma_3) \in \mathcal{R}$ are functional.

Context formation ($E \vdash_{\mathcal{A}} \mathsf{ok}$):

$$\frac{E \vdash_{\mathcal{A}} M \ \mathsf{modtype} \quad x \notin BV(E)}{E; \texttt{module}\ x : M \vdash_{\mathcal{A}} \mathsf{ok}} \qquad \frac{E \vdash_{\mathcal{A}} e : e' \quad w \notin BV(E)}{E; w : e' = e \vdash_{\mathcal{A}} \mathsf{ok}}$$

Module type and signature body formation ($E \vdash_{\mathcal{A}} M\ \mathsf{modtype}$):

$$\frac{E \vdash_{\mathcal{A}} \mathsf{ok}}{E \vdash_{\mathcal{A}} \epsilon\ \mathsf{modtype}} \qquad \frac{E; \texttt{module}\ x : M \vdash_{\mathcal{A}} S\ \mathsf{modtype} \quad y \notin N(S)}{E \vdash_{\mathcal{A}} \texttt{module}\ y \triangleright x : M; S\ \mathsf{modtype}}$$

$$\frac{E; v : e \vdash_{\mathcal{A}} S\ \mathsf{modtype} \quad w \notin N(S)}{E \vdash_{\mathcal{A}} \texttt{term}\ w \triangleright v : e; S\ \mathsf{modtype}} \qquad \frac{E; v : e = e' \vdash_{\mathcal{A}} S\ \mathsf{modtype} \quad w \notin N(S)}{E \vdash_{\mathcal{A}} \texttt{term}\ w \triangleright v : e = e'; S\ \mathsf{modtype}}$$

$$\frac{E \vdash_{\mathcal{A}} S\ \mathsf{modtype}}{E \vdash_{\mathcal{A}} \texttt{sig}\ S\ \texttt{end}\ \mathsf{modtype}} \qquad \frac{E \vdash_{\mathcal{A}} M\ \mathsf{modtype} \quad x \notin BV(E) \quad E; \texttt{module}\ x : M \vdash_{\mathcal{A}} M'\ \mathsf{modtype}}{E \vdash_{\mathcal{A}} \texttt{functor}(x : M) M'\ \mathsf{modtype}}$$

Module expressions ($E \vdash_{\mathcal{A}} m : M$) and structures ($E \vdash_{\mathcal{A}} s : S$):

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

$$\frac{E; \texttt{module}\ x : M \vdash_{\mathcal{A}} m : M' \quad E \vdash \texttt{functor}(x : M) M'\ \mathsf{modtype}}{E \vdash_{\mathcal{A}} \texttt{functor}(x : M) m : \texttt{functor}(x : M) M'}$$

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

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

$$\frac{E \vdash_{\mathcal{A}} e : e' \quad v \notin BV(E) \quad E; v : e' = e \vdash_{\mathcal{A}} s : S \quad w \notin N(s)}{E \vdash_{\mathcal{A}} (\texttt{term}\ w \triangleright v = e; s) : (\texttt{term}\ w \triangleright v : e = e'; S)}$$

$$\frac{E \vdash_{\mathcal{A}} m : M \quad x \notin BV(E) \quad E; \texttt{module}\ x : M \vdash_{\mathcal{A}} s : S \quad y \notin N(s)}{E \vdash_{\mathcal{A}} (\texttt{module}\ y \triangleright x : M = m; s) : (\texttt{module}\ y \triangleright x : M; S)}$$

Figure 5: Type inference system

Module types subtyping ($E \vdash_{\mathcal{A}} M_1 <: M_2$):

$$\frac{E \vdash_{\mathcal{A}} M \ \texttt{modtype} \quad E \vdash_{\mathcal{A}} M' \ \texttt{modtype} \quad M =_{\alpha} M'}{E \vdash_{\mathcal{A}} M <: M'}$$

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

$$\frac{E \vdash_{\mathcal{A}} M_2 <: M_1 \quad E; \texttt{module} \ x : M_2 \vdash_{\mathcal{A}} M'_1 <: M'_2}{E \vdash_{\mathcal{A}} \texttt{functor}(x : M_1) M'_1 <: \texttt{functor}(x : M_2) M'_2}$$

$$\frac{E \vdash_{\mathcal{A}} M <: M'}{E \vdash_{\mathcal{A}} \texttt{module} \ x \triangleright y : M <: \texttt{module} \ x \triangleright y : M'}$$

$$\frac{E \vdash_{\mathcal{A}} e \approx e'}{E \vdash_{\mathcal{A}} \texttt{term} \ v \triangleright w : e[= e''] <: \texttt{term} \ v \triangleright w : e'}$$

$$\frac{E \vdash_{\mathcal{A}} e_1 \approx e'_1 \quad E \vdash_{\mathcal{A}} w \approx e'_2}{E \vdash_{\mathcal{A}} \texttt{term} \ v \triangleright w : e_1[= e_2] <: \texttt{term} \ v \triangleright w : e'_1 = e'_2}$$

Term equivalence ($E \vdash_{\mathcal{A}} e \approx e'$):

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

Reduction:

$$\frac{E_1; w : e \ \texttt{=} \ e'; E_2 \vdash_{\mathcal{A}} \texttt{ok}}{E_1; w : e \ \texttt{=} \ e'; E_2 \vdash_{\mathcal{A}} w \to_{\delta} e'} \qquad \frac{E \vdash_{\mathcal{A}} m : \texttt{sig} \ S_1; \texttt{term} \ v \triangleright w \ \texttt{=} \ e; S_2 \ \texttt{end}}{E \vdash_{\mathcal{A}} m.v \to_{\delta} e\{n \leftarrow m.n' \mid (n', n) \in BV(S_1)\}}$$

Module equivalence ($E \vdash_{\mathcal{A}} m \approx m' : M$):

$$\frac{\begin{array}{l} E \vdash_{\mathcal{A}} m : N \quad E \vdash_{\mathcal{A}} N/m <: \texttt{sig} \ D_1; \ldots; D_n \ \texttt{end} \\ E \vdash_{\mathcal{A}} m' : N' \quad E \vdash_{\mathcal{A}} N'/m' <: \texttt{sig} \ D_1; \ldots; D_n \ \texttt{end} \\ \forall i \in \{1, \ldots, n\} \quad D_i = \texttt{term} \ v \triangleright w : e \ \texttt{=} \ e' \Rightarrow E \vdash_{\mathcal{A}} m.v \approx m'.v \\ D_i = \texttt{module} \ x \triangleright y : M \Rightarrow E \vdash_{\mathcal{A}} m.x \approx m'.x : M\{n \leftarrow m.n' \mid (n', n) \in BV(\texttt{sig} \ D_1; \ldots; D_n \ \texttt{end})\} \end{array}}{E \vdash_{\mathcal{A}} m \approx m' : \texttt{sig} \ D_1; \ldots; D_n \ \texttt{end}}$$

$$\frac{E \vdash_{\mathcal{A}} m : N \quad E \vdash_{\mathcal{A}} N/m <: \texttt{functor}(x : M_1) M_2 \quad E; \texttt{module} \ x_i : M_1 \vdash_{\mathcal{A}} (m \ x_i) \approx (m' \ x_i) : M_2}{E \vdash_{\mathcal{A}} m \approx m' : \texttt{functor}(x : M_1) M_2}$$

Figure 6: Type inference system

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 $E \vdash e \approx e'$ then $E \vdash_{\mathcal{A}} e \approx e'$.*

Proof: Induction on the derivation.

4.2.2 Term normalization

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 to mind is $\delta\mu$-normalization. However, this does not work; thus in environment

$$E; x : \texttt{functor(}x : \texttt{sig term}\ v \triangleright v' : e\ \texttt{end)sig term}\ u \triangleright u' : e'\ \texttt{end}$$

where $f : e$, the expressions

$$(x\ ((\texttt{functor(}x : \texttt{sig}\ \ \texttt{end)struct term}\ v \triangleright v' = f\ \texttt{end)}\ \texttt{struct}\ \ \texttt{end})).u$$

and

$$(x\ \texttt{struct term}\ v \triangleright v' = f\ \texttt{end}).u$$

are in δ-normal form, and syntactically distinct though they are easily proved equivalent:

$$\begin{array}{ll} E \vdash_{\mathcal{A}} & ((\texttt{functor(}x : \texttt{sig}\ \ \texttt{end)struct term}\ v \triangleright v' = f\ \texttt{end)}\ \texttt{struct end}) \\ & \approx\ \texttt{struct term}\ v \triangleright v' = f\ \texttt{end} \\ & : \texttt{sig term}\ v \triangleright v'\ \texttt{end} \end{array}$$

However, we shall see that we can always proceed in this way to compare types, that is, $\delta\beta$-normalizing them first, then comparing recursively modules expressions that are arguments of the head variable.

Then, we may wonder whether this process always terminates or not. In order to answer this question, we first give the following definition:

Definition 1 ($\approx$-reducible terms and $\approx$-reducible modules for a given module type) *In an environment E, we say a module m is $\approx$-reducible for module type M if $E \vdash m : M$, and one of the following cases is verified:*

- *$M = \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end}$, for all i such that $D_i = \texttt{term}\ v \triangleright v'[= e]$, $m.v$ is $\approx$-reducible and for all i such that $D_i = \texttt{module}\ x \triangleright x' : N$, $m.x$ is $\approx$-reducible for type $N\{n \leftarrow m.n' \mid (n', n) \in BV(D_1, \ldots, D_{i-1})\}$;*
- *$M = \texttt{functor(}x : M_1\texttt{)}M_2$, and $m(x)$ is $\approx$-reducible for type M_2 in $E; \texttt{module}\ x : M_1$;*

A term e is said to be $\approx$-reducible if and only if it is strongly $\beta\delta$-normalizing and its $\beta\delta$-normal is $\approx$-reducible. A $\beta\delta$-normal term e is said to be $\approx$-reducible if and only if one of the following cases is verified:

- *$e = (e_1\ e_2)$ and e_1 and e_2 are $\approx$-reducible ;*
- *$e = \lambda v : e_1.e_2$ and e_1 and e_2 are $\approx$-reducible ;*
- *e has form $(\ldots((xm_1 \ldots m_i).x_1 n_1 \ldots n_j)\ldots).v$ where the arguments $m_1, \ldots, m_i, \ldots, n_1, \ldots, n_j, \ldots$ of the head variable x are $\approx$-reducible for types expected by $x, (x\ m_1), \ldots$.*

Notice the expression "its $\beta\delta$ normal form" is justified by the easily proved confluence of $\beta\delta$-reduction.

We then have the following results:

Theorem 10 (Term $\approx$-reducibility) *If $E \vdash_{\mathcal{A}} m : M$ then m is $\approx$-reducible for M; if $E \vdash_{\mathcal{A}} e : e'$ then e is $\approx$-reducibility.*

Sketch of proof: First, we can prove that we can deal only with δ-normalization instead of $\beta\delta$-normalization in the definition of $\approx$-reducible terms. This can be done because of strong normalization of β-reduction together with the fact that if e β-reduces to e', the δ-normal form of e β-reduces to the δ-normal form of e'. Then, the proof can be done by defining a reducibility notion as in [GLT89] for the simply-typed lambda-calculus.

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

Lemma 1 *For all terms e and e' such that $E \vdash_{\mathcal{A}} e \approx e'$, $\delta\beta$-normal forms of e and e' have the same head variables; moreover 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.

4.2.3 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, *i.e.* an algorithm which 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: 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.

5 Comparison with other works

Compared to the module system of Elf [HP92], our system is much more powerful, because of manifest declarations. Moreover, we can give a proof of its consistency through the study of reductions. Finally, we are not aware of separate compilation mechanism for the module system of Elf.

Extended ML [San90, KSTar] is a very interesting framework for developing SML modular (functional) programs together with their specification and the proof of their specification. However, it is not as general as provers based on PTS can be for developing mathematical theories. Moreover, we are not aware of any proof of consistency of the EML approach.

Another way to structure a development and make parameterized theories is to add dependent record types to PTS. In systems with dependent sum types such as the Extended Calculus of Construction [Luo89], or inductive types such as the Calculus of Construction with Inductive Types [PM93], this is quite easy, and is more or less a syntactic sugar [Sai96]. This approach have some advantages over ours.

Firstly, functors are represented by functions from a record type to another. Therefore, there is no need for specific rules for abstraction and application of modules, since they are only particular cases of the type system rules.

Secondly, having "modules" as first-class citizens allows powerful operations since it gives the "module" language the whole power of the base language. For instance, one can define a function taking as input a natural n and a monoïd structure M and giving back as output the monoïd M^n. Such a function has to be recursive whereas a functor cannot be recursive in our approach.

However the module-as-record approach suffers severe disadvantages.

Firstly, the addition of records may be difficult from a theoretical point of view. Indeed, too powerful elimination schemes can make a system logically inconsistent. For instance, Russel's paradox can be formulated in the Calculus of Construction where one can have records of type `Set` having a set as only component if strong elimination is allowed. Hence, records are mainly useful in systems with a universes hierarchy, such as the Calculus of Construction with Inductive Types and Universes, or the Extended Calculus of Construction. Thus, the conceptual simplicity of the record approach is lost with the complexity of universes. On the other hand, our system is orthogonal to the considered PTS, and therefore much more robust to changes in the base language from a logical point of view.

Secondly, the abstraction mechanism is very limited. Indeed, either every component of a record is known (in the case of an explicit term or of a constant) or every component is hidden (in the case of a variable or an opaque constant)[3]. For instance, the product of two vectorial spaces is defined only if their field component is the same. This restriction is easily expressed in our system where we can define a module as

$$\begin{array}{l}\texttt{functor}(V_1 : \texttt{<<}\mathit{vectorial\ space}\texttt{>>})\\ \quad \texttt{functor}(V_2 : \texttt{<<}\mathit{vectorial\ space\ with}\ K.E = V_1.K.E,\ K.+ = V_1.K.+, \ldots\texttt{>>})\ldots\end{array}$$

But, it is very difficult to define such a functor in a record-based formalism since there is no way to express that two given field are convertible. One could of course think of defining a notion of K-vectorial space, but this would require the addition of one parameter for each function on vectorial space.

Moreover, separate compilation of non-closed code fragments is not possible. Indeed, one sometimes needs the definition of a term in order to type-check an expression e, but the only way to know a component of a record is to know the whole record, hence it has to be compiled before e is checked. On the contrary, our notion of specification allows us to give in an interface file a specification containing only the level of details needed from the outside of a module.

6 Conclusion

We propose a module system for Pure Type Systems. This module system can be seen as a typed lambda-calculus of its own, since it enjoys the subject reduction property. This system has several desirable properties:

- it is independent of the considered PTS, hence should be robust to changes in the base type system (addition of inductive types for instance);
- it is powerful enough to handle usual mathematical operations on usual structures;
- it is strongly normalizing;
- it is conservative with respect to the considered Pure Type System, especially it does not introduce any logical inconsistency;
- type inference is decidable provided the β-reduction in the considered PTS is strongly normalizing thus allowing an effective implementation of it;
- it allows true separate compilation of non-closed code fragments.

Our approach also brings several new issues.

Firstly, it would also be interesting to see which mechanisms are needed for helping the user search through module libraries. The work done in [Rou90, Rou92, DC95] may be of great interest in this respect.

Another issue is how to integrate proof-assistant tools in our module system. Thus, it would be interesting to add tactics components to modules helping the user by constructing proof-terms in a semi-automatic way. Similar work has been done for the IMPS prover [FGT95]: each theory comes together with a set of *macetes* that are specific tactics for a proof in this theory. A similar idea can be found in the prover CiME [CM96], where the user may declare he is in a given theory in order to get associated simplification rules.

It would also be interesting to see how far the idea of independence with respect to the base language can be formalized. In order to adapt the system of [Cou96] to PTS, we had to deal with β-equivalence and the interaction of β-reduction with δ-reduction; is it possible to give an abstract notion of equivalence on a base language, and general conditions allowing to extend this base language with modules (one may especially think of the Calculus of Constructions with Inductive Types and the associated ι-reduction, or of the Calculus of Constructions with $\beta\eta$-equivalence rule for conversion...).

[3] It should be noticed that Jones [Jon96] proposed a way to solve this problem in a programming language with records and the ability to define abstract types, but this approach applies only in system where polymorphism is implicit and where types do not depend on terms.

Finally, possible extensions of our system have to be studied. Allowing signature abbreviations as structure components may seem to be a slight extension. But, as pointed out in [HL94], such an extension can lead to subtype-checking undecidability if one allows abstract signature abbreviation components in signatures. However, while one allows only manifest abbreviations, no problem arises. More generally, a challenging extension is to add type signatures variables, type signatures operators,... without losing type inference decidability. Another direction would be the addition of overloaded functors as in [Cas94, AC96].

We also hope to implement soon ideas given in this paper in the Coq proof assistant.

References

[AC96] María Virginia Aponte and Giuseppe Castagna. Programmation modulaire avec surcharge et liaison tardive. In *Journées Francophones des Langages Applicatifs*, January 1996.

[Bar91] H. Barendregt. Lambda calculi with types. Technical Report 91-19, Catholic University Nijmegen, 1991. in Handbook of Logic in Computer Science, Vol II.

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

[Cas94] G. Castagna. *Surcharge, sous-typage et liaison tardive : fondements fonctionnels de la programmation orientée objets*. Thèse de doctorat, Laboratoire d'Informatique de l'Ecole Normale Supérieure, January 1994.

[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.

[CM96] Evelyne Contejean and Claude Marché. Cime: Completion modulo e. In Harald Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, Rutgers University, NJ, USA,, July 1996.

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

[Cou96] Judicaël Courant. A module calculus enjoying the subject-reduction property. Research Report RR 96-30, LIP, 1996. Preliminary version.

[DC95] Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[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.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993. Preliminary version appeared in Proc. 2nd IEEE Symposium on Logic in Computer Science, 1987, 194–204.

[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.

[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.

[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.

[Luo89] Zhaohui Luo. Ecc: an extended calculus of constructions. In *Proc. of IEEE 4th Ann. Symp. on Logic In Computer Science*, Asilomar, California, 1989.

[Mac85] David B. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2), 1985. 35 pages. An earlier version appeared in Proc. 1984 ACM Conf. on Lisp and Functional Programming.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[PM93] C. Paulin-Mohring. Inductive definitions in the system Coq : Rules and Properties. In M. Bezem, J.F. Groote, editor, *Proceedings of the TLCA*, 1993.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[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`.

[Sai96] Amokrane Saibi, 1996. Private Communication.

[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.

[SP94] P. Severi and E. Poll. Pure type systems with definitions. *Lecture Notes in Computer Science*, 813, 1994.

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