Signature subtyping modulo type isomorphisms

María Virginia Aponte[*] Roberto Di Cosmo[†] Catherine Dubois[‡]

Draft: February 11, 1998

Abstract

To manage large libraries of program components, essentially built out of modules, we introduce a novel notion of subtyping of modules that abstracts out inessential details: subtyping up to module type isomorphism. This results from the composition of the usual subtyping relation of modules with a natural extension of the notion of isomorphism for types in functional languages to the quite more complex case of module systems.

We formalize the notion of module equivalence, and show how subtyping modulo this equivalence can be naturally checked on appropriate canonical forms via an algorithm which is proved sound and complete.

This relaxed notion of subtyping allows to build search tools which are well adapted to modern module libraries, and we actually implemented a search tool based on the theoretical notion described here for the `Ocaml` system.

1 Introduction

1.1 Code reuse and type isomorphisms

Code reuse is an useful technique to write and maintain large and sophisticated programs. An important opportunity to reuse code is given by libraries containing many useful and generic pieces of code, but to take fully advantage from them, the existence of very general search tools for libraries becomes necessary.

Type isomorphisms have been successfully used as a basis for advanced library search tool [18, 5]. An isomorphism is a way to identify two types t and w for which any object of type t can be coerced into an object of type w, and conversely, without loosing information. That is, t and w contain the same information, eventually organized in a different way. A typical example of isomorphism, well known to all functional programmers, is the curry-uncurry isomorphisms between the types $(t \times w) \to r$ and $t \to w \to r$. In [18] Rittri shows how the same function can have very different looking types, depending on programmer's choices. For instance, the function `fold` in SML with type $(\alpha \times \beta \to \beta) \to \alpha \to List(\alpha) \to \beta$ is called `list_it` in Caml-Light, and has type $(\alpha \to \beta \to \beta) \to List(\alpha) \to \beta \to \beta$. These two types are *isomorphic*, which was the basis for Rittri's idea [18]: search tools search for types and not for names, and when looking for a function with a given type, one must retrieve all the functions with isomorphic types and not only those with syntactically equal types. The study of type isomorphism for the typed λ-calculus has lead to complete and decidable axiomatization of isomorphisms and to efficient library tools based on types. Examples are library search tools developed for typed functional languages: Caml-Light comes equipped with `camlsearch`, and

* CEDRIC - CNAM Conservatoire National des Arts et Métiers - 292, rue St. Martin - 75003 Paris France
E-mail:`aponte@cnam.fr`
† DMI-LIENS (CNRS URA 1347) Ecole Normale Supérieure - 45, Rue d'Ulm - 75230 Paris France
E-mail:`dicosmo@dmi.ens.fr`
‡ Université d'Evry Boulevard des Coquibus - 91330 Evry France
This work was done while visiting CNAM
E-mail:`dubois@lami.univ-evry.fr`

more recently, a search tool based on type isomorphisms is being developed for Haskell.

Search tools based on type isomorphisms are very general and powerful: on one side, they exploit quite meaningful information (types) which allows them to find more accurate answers to the user's request; on the other, isomorphisms allow different representations between the user's required type and the actual type in the library. Thus, search tools based on type isomorphisms are well adapted to the increasing requirements of programming environments: by giving appropriate answer to the user's specifications they alleviate the effort of writing code, they allow to give to the code more unity (by the use of very general programs in the library), reducing time and effort in the writing and maintaining of programs.

1.2 Module equivalence and library search

As programs become larger and more sophisticated, they end up being built using modular constructions in a fundamental way, and naturally, libraries tend to be written as a collection of many general modules relevant by themselves (and not only because of some of their components), that is, as abstractions of pieces of programs. It is then a necessity to be able to provide search tools adapted to this kind of libraries, which are more appropriately searched using full signatures as specifications and not just the type of some isolated function.

In [2] we proposed, as a first step in the study of search tools for modules in libraries, a rule system for checking isomorphisms for a variant of SML modules (Leroy's modules, [10, 12], implemented now in Objective Caml [8]). The equivalences proposed there start from the analogy between simple types and module types, but go further on: they capture structural differences more complex than simple re-ordering of components, handle type sharing and higher-order functors, leading to a very general solution for the isomorphisms problem between module signatures. The equivalences captured there include simple type isomorphisms in the types of individual components, differences in names and order of components, as well as differences in the way they are organized in inner modules. For instance, the two signatures below are equivalent according to [2]:

```
sig
 type ord
 val less : ord * ord -> bool
 type set = ord list
 val empty : set -> bool
 val insert: ord * set -> set
end
```

```
sig
  module Ord: sig type t val leq: t->t-> bool end
  type s = Ord.t list
  val add: s -> Ord.t -> s
  val null : Ord.t list -> bool
 end
```

Nevertheless, strict isomorphism is too rigid to be satisfactory as a basis for useful searching in libraries: modules are essentially products of named components (much as records [1]), and thus, a user searching for a module can hardly guess a module type having the exact quantity of information as the module actually has in the library. For instance, a module in the library containing more code than what the user explicitly specified would be automatically rejected by the search tool, even if the user could perfectly well use only the part relevant to him. This situation is quite similar to the one which motivated the introduction of the classical subtyping notion already present in powerful module systems: a module with type S satisfies (or *matches*) a given module type S' if they can be proved in an inclusion relation $S' <: S$, meaning that S' has at least all the components found in S.

This paper comes as an attempt to find a way out of this tension. We propose a notion of subtyping up to isomorphisms that combines the two approaches of type comparison mentioned before: on one side type isomorphisms to allow flexibility in the choice of type representation, component names as well as their organization; on the other, type inclusion to require retrieved modules to fit user's specified signature, yet allowing retrieved modules to be richer than the specified one. We consider the user's specified signature as the minimum amount of type information the user wants to find in the library. Our subtyping mechanisms allows a search tool based on it to successfully match richer module types in the library against the user's specification up to a set of very general module type equivalences. The modules satisfying the user's signature can be written in very different ways and overpass the user needs, but by our subtyping notion they fit at least with the user's type requirements. For instance, an incomplete specification for sets given by the user (on the right) can be matched against the type of the actual module in the library (on the left):

```
sig
 module Ord:
   sig
    type t
    val less: t*t-> bool
   end
 type set
 val empty: set -> bool
 val emptyset : set
 val mem: Ord.t -> set -> bool
 val insert: Ord.t * set -> set
end
```

```
sig
 type item
 type s
 val isnull: s -> bool
 val isin: s*item -> bool
end
```

The module on the left is richer than the one in the right: it contains at least all the type information in the user's specification, but only if we consider them *up to module type isomorphisms*. If we try to show isomorphisms on both modules we would fail, but we can show that there exists equivalent transformation of both signatures (following equivalences in [2], that we also describe in section 3) leading to signatures in a classical subtyping relation.

1.3 The problem of converters

As is already apparent in the very simple example given above, a search tool based on isomorphisms can retrieve modules that look quite different from the user's specification: different component names, different structure, different types. This means that in order to use the retrieved module, it is necessary to built a *converter*, i.e. a functor able to take as parameter the module in the library and building as result a module having exactly the user's signature.

For example, the converter associated to the previous example is:

```
module Converter (S : sig
           module Ord : sig type t val less : t * t -> bool end
           type set
           val empty : set -> bool
           val emptyset : set
           val mem : Ord.t -> set -> bool
           val insert : Ord.t * set -> set
         end) =
   (struct
     type item = S.Ord.t
     type s = S.set
     let isnull = S.empty
     let isin = fun (s,i) -> S.mem i s
   end
  :
   sig
     type item
     type s
     val isnull: s -> bool
     val isin: s*item -> bool
   end)
```

Which can be applied to the retrieved implementation to produce a module with the user supplied signature.

This kind of converters can be extremely complex to build in general (consider for example that we had to currify `S.mem` in order to get `isin` and flatten the submodule `Ord`), so one *cannot* leave this burden

on the user: it is necessary to be able to build them automatically. This is a significant departure from what is traditionally done in isomorphism-based search tools for simple functions, where, due to their usual simplicity, converters are never provided.
In this paper we describe a procedure to build them, as is implemented in our prototype.

1.4 Previous and related work

There has been a growing interest in using isomorphisms of types as a guide to search or organize libraries of software components: starting from the seminal works of Rittri [18, 17] and Runciman and Toyn [20], several authors have worked on the theoretical foundations of the subject [6, 21] and towards real implementations for various languages like LML [16], Camllight [5], Haskell [13].
More recently, for all the reasons already stressed above, people started to look at modules instead of functions as the right level of abstraction of a library component, and to apply the same ideas to them: this is the case of part of the work carried on in the Venari project at CMU [19, 22], of some recent work by Muller [14], and of some previous work of two of the authors [2].
These previous attempts to handle module libraries show the interest of the subject, but in these authors' opinion, they all fall short of providing a satisfactory solution: the work in the Venari project and Muller's proposal do not take into consideration the quite complex structural transformations and component renamings that can show up in a module library, while our previous work only laid a minimal foundation to library search of modules, as it only provided the notion of equivalence of modules, which is, as we stressed above, too strong to be used as a match criterion.

On a different, but related track, we should mention Parot's work [15], which shows how logic programming tools can be used to implement automatic module composition.

1.5 This work

In this paper we propose a novel and, in these authors' opinion, fully satisfactory notion of match between an user query and a library module: subtyping up to module type isomorphisms. This allows to build search tools that match richer library module types against a minimal module type specification given by the user, abstracting out all inessential details identified in previous works in the literature.

We first introduce an elegant equational theory expressing the equivalence between module signatures that neatly captures all the transformations discussed in [2]. This theory is described in section 3 and is related to the signature language we present in the section 2. Building on it, we introduce the notion of subtyping up to isomorphisms is defined in section 4. Then, we extract from our theory a set of rules that allow to transforms a signature into an equivalent and simpler canonical form. These rules are discussed in section 5, and allow to propose in section 6 an algorithm to decide whether a user's signature matches a signature found in the library. This algorithm is proved correct and complete with respect to our definition. The section 7 describes a procedure to build converters. We have experimented all these ideas with a prototype implementation written in Objective Caml: it implements our decision algorithm and builds converters. It is very briefly described in the last section.

2 The signature language

Our signature language is a subset of the module calculus presented by Leroy [10, 12] as a variant of the SML module system. Since then, his work have progressed into a new module system now avalaible in the Objective Caml language [8, 11]. Modules are either *structures* or *functors*. A structure is a collection of named components composed by value, type and inner module declarations. A functor is a function ranging from modules to modules. A structure type (also called *signature*), contains specifications for all components in a structure, while a functor type is a dependent function type built on top of the types of a named argument and the functor body.

The syntax of module types is shown in the grammar below, where v, t and x range respectively over value, type and module names. We follow [10] in the distinction between names and identifiers. An identifier v_i is composed from a name v and a mark i, and marks are used to distinguish identifiers corresponding to the same name. This distinction is necessary in order to avoid name capture while renaming identifiers during subtyping [10] and during the signature transformations we describe later. Additionally, we assume that all the identifiers bound in a given signature have different names.

Structure paths p are formed by a structure identifier followed by structure names. We allow identifiers v_i, t_i, x_i in [10] to be paths, v_p, t_p, x_p, that is, structure paths ending respectively in a value, a type and a module name. We call them *path identifiers*. In their general form, path identifiers are introduced by signature transformation rules, and they are used only during the phase of equivalence checking. We do not specify completely type expressions from the base language, since the signature language makes few assumptions about them. Actually, the module calculus presented in [10] can be implemented on top of a variety of base languages, giving it a great generality.

Module types:

S	$::= \texttt{sig}\ D\ \texttt{end}$	structure type
	$\mid \texttt{functor}\ (x_p : S)S'$	functor dependent type

Structure type body:

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

Signature components:

C	$::= \texttt{val}\ v_p : \tau$	value specification
	$\mid \texttt{type}\ t_p$	abstract type specification
	$\mid \texttt{type}\ t_p = \tau$	manifest type specification
	$\mid \texttt{module}\ x_p : S$	module specification

Base type expressions:

τ	$::= t_i \mid p.t \mid \ldots$	type identifiers and type paths

Structure paths:

$p ::= x_i \mid p.x$

Path identifiers:

$v_p ::= v_i \mid p.v$

$t_p ::= t_i \mid p.t$

$x_p ::= p$

3 Signature equivalence

The problem of type isomorphisms can be addressed from a semantic or a syntactic point of view. The semantic approach is useful when building converters for two types known to be equivalent, as we do in section 7. Syntactic definitions, laying usually on equational theories, can be used to derive algorithms for checking equivalence between two given types. In this section we present an equational theory for module types corresponding to the algorithm we developed in [2] for checking signature equivalence. The algorithm in [2], as well as the equational theory we show below, capture very general equivalences: equivalences on base type representation, equivalences on types with respect to manifest type declarations in the type context, differences in the names of components and differences in the way components are organized into inner modules.

3.1 The equational theory $\simeq$

We consider two theories named $\cong$ and $\simeq$. The former describes equations between module types while the latter captures the theory $\cong$ together with renaming of component names. In both theories we consider two sets of equations: *componentwise* and *structural* equations. We write $D\{t \leftarrow \sigma\}$ the substitution of the type σ for the type identifier t in the declaration D. All equations use path identifiers to avoid name capture while doing substitutions, but to improve readability, we use sometimes names or identifiers instead of path identifiers. Thus, when using an identifier n_i (n ranging on any kind of name) in the equations we actually mean a path identifier of the form $x_i.y \ldots .n$. Componentwise equations are shown below. They use equivalences between base types (*base equivalences*) to compare structures componentwise and propagation of manifest type declarations[1] (*type expansion*). Base equations use an equational theory $\approx_{\mathtt{iso}}$ which depends on the base type language: $\sigma \approx_{\mathtt{iso}} \sigma'$ is sastified when σ and σ' are isomomorphic base types in the theory $\approx_{\mathtt{iso}}$.

base equivalence

$$\begin{array}{rcll}
\mathtt{val}\ v_i : \sigma & \cong & \mathtt{val}\ v_i : \sigma' & \text{iff}\quad \sigma \approx_{\mathtt{iso}} \sigma' \\
\mathtt{type}\ t_i & \cong & \mathtt{type}\ t_i & \\
\mathtt{type}\ t_i = \sigma & \cong & \mathtt{type}\ t_i = \sigma' & \text{iff}\quad \sigma \approx_{\mathtt{iso}} \sigma' \\
\mathtt{module}\ x_i : S & \cong & \mathtt{module}\ x_i : S' & \text{iff}\quad S \cong S'
\end{array}$$

type expansion

$$\begin{array}{rcl}
\mathtt{type}\ t_i = \sigma; D & \cong & \mathtt{type}\ t_i = \sigma; D\{t_i \leftarrow \sigma\} \\
\mathtt{module}\, x_i : \mathtt{sig}\, D_1; \mathtt{type}\ t_j = \sigma; D_2\, \mathtt{end}; D_3 & \cong & \mathtt{module}\, x_i : \mathtt{sig}\, D_1; \mathtt{type}\ t_j = \sigma; D_2\, \mathtt{end}; D_3\{x_i.t \leftarrow \sigma\}
\end{array}$$

module type equivalence

$$\begin{array}{rcll}
\mathtt{sig}\ D_1; \ldots; D_n\ \mathtt{end} & \cong & \mathtt{sig}\ D'_1; \ldots; D'_n\ \mathtt{end} & \text{iff}\quad \exists \sigma : \{1, \ldots, n\} \to \{1, \ldots, n\} \text{ s.t } D_{\sigma(i)} \cong D'_i \\
\mathtt{functor}(x_i : S)\mathtt{sig}\, D\, \mathtt{end} & \cong & \mathtt{functor}(x_i : S_1)\mathtt{sig}\, D_1\, \mathtt{end} & \text{iff}\quad S \cong S_1 \\
& & & \text{and } \mathtt{module}\, x_i : S; D \cong \mathtt{module}\, x_i : S_1; D_1
\end{array}$$

Componentwise equivalences capture very simple equivalences, roughly speaking, base type equivalences when comparing components. But they do not capture more subtle equivalences as those present in the example below:

```
sig
 module x:
  sig
   type t = int
   val  v:t
  end
 val w:x.t
end
```

```
sig
 val m:int
 type q = int
 val n: q
end
```

Structural equations capture some of these equivalences: they equate signatures with different shapes and, in some cases, with different names of components. *Structure flattening* equate a signature with an inner module declaring some components and the same signature where there is no inner module but only all its components. Any path identifier n_j local to an inner module is renamed as a path prefixed by the name of the inner module. This transformation preserves external dependencies on local identifiers (see [2] for examples).

structure flattening

$$\mathtt{module}\ x_i : \mathtt{sig}\ D\ \mathtt{end}\ ; D' \quad \cong \quad D\{n_j \leftarrow x_i.n \mid n_j \in BV(D)\}; D'$$

[1] In this work, we consider only non-recursive base types.

Following [10], we write $BV(D)$ for the set of bound path identifiers in the specification D. This set is defined as:

$$\begin{array}{rcl} BV(\epsilon) & = & \emptyset \\ BV(\texttt{val } v_i; D) & = & \{v_i\} \cup BV(D) \\ BV(\texttt{type } t_i; D) & = & \{t_i\} \cup BV(D) \\ BV(\texttt{type } t_i = \sigma; D) & = & \{t_i\} \cup BV(D) \\ BV(\texttt{module } x_i : S; D) & = & \{x_i\} \cup BV(D) \end{array}$$

We call *product type splitting* the equivalence of a signature specifying a value (or a manifest type) with a product type $t_1 * t_2$ to a signature specifying two separate values (resp. two different manifest types) declared with types t_1 and t_2. Finally, we consider as equivalent functors having several arguments with functors with a single argument, by means of a sort of uncurry transformation.

product type splitting

$$\begin{array}{rcl} \texttt{val } a : t_1 * t_2 & \cong & \texttt{val } b : t_1; \texttt{val } c : t_2 \\ \texttt{type } t = \sigma_1 * \sigma_2; D & \cong & \texttt{type } q = \sigma_1; \texttt{type } r = \sigma_2; D' \quad \text{where } D' = D\{t \leftarrow q * r\} \end{array}$$

curry-uncurry

$$\begin{array}{rcl} \texttt{functor}(x_i : S_1)\texttt{functor}(y_j : S_2)S_3 & \cong & \texttt{functor}(z_k : \texttt{sig module } x_i : S_1;\ \texttt{module } y_j : S_2 \texttt{ end}) \\ & & S_3\{x_i.n \leftarrow z_k.x.n, y_j.n \leftarrow z_k.y.n\} \end{array}$$

The equations above allow equivalence only in few cases of differences for the names of components: essentially they come from the choice of product types in one signature while non-product type components are in the other. To capture all possible renaming of components, we define the equational theory $\simeq$, that considers all equations above up to α-conversion of names of components.

Definition 1 (Module type equivalence $\simeq$):

We say that two module types S and S' are equivalent in the theory $\simeq$, written $S \simeq S'$, if given the domains of names $Dom(S)$ of S and $Dom(S')$ of S', and a set N of fresh names, that is, $N \cap Dom(S) = N \cap Dom(S') = \emptyset$, there exists renamings $\sigma_1 : D(S) \to N$, and $\sigma_2 : D(S') \to N$, such that: $\sigma_1(S) \cong \sigma_2(S')$.
□

We want to single out here a special subset of equivalences included in the theory above that will be important later.

component reordering

$$\begin{array}{rcl} \texttt{val } a : t_i; D & \cong & D; \texttt{val } a : t_i \text{ if } t_i \notin TD(D) \\ \texttt{type } t_i = \tau; D & \cong & D; \texttt{type } t_i = \tau \text{ if } \left\{ \begin{array}{l} t_i \notin TD(D) \\ t_i \notin FTV(D) \\ FTV(\tau) \cap FTD(D) = \emptyset \end{array} \right. \\ \texttt{type } t_i; D & \cong & D; \texttt{type } t_i \text{ if } \left\{ \begin{array}{l} t_i \notin TD(D) \\ t_i \notin FTV(D) \end{array} \right. \end{array}$$

Where $FTV(D)$ denotes the set of free type variables in D, and $TD(D)$ is the set of type names definitions in D. Notice that the restrictions on the reorderings, needed to avoid variable capture, are equivalent to those introduced in [12].

need to check whether the conditions on types can be given in term of index crossing restrictions

Definition 2 (Reordering equivalence $\cong_r$):

We write $\cong_r$ for the equality generated by the two reordering axioms above and the two product splitting axioms. □

4 Subtyping up to isomorphism

Classical subtyping compares signatures component to component modulo some permutations in the order of their declaration. An example is given by the relation $<:$ defined by Leroy ([10, 12]) for the module calculus we use in this work. According to this relation, a signature S being subtype of a signature S' (written $S' <: S$) can have more components and also more manifest type declarations than those in S', while inner modules from each signature must be in the subtyping relation. We show in appendix A the set of rules coming from [10, 12] that defines Leroy's $<:$ relation. We use this relation later in this section to define subtyping up to isomorphisms.

As we already mentioned in the introduction, both classical subtyping and strict type equivalence are too rigid to be realistic solutions for module retrieval from libraries. A practical tool for module retrieval must combine both approaches: type isomorphisms allow flexibility in the choice of type representation, component names, and their organization and type inclusion allows retrieved modules to fit the user's specification but also to be richer than it. The definition of module subtyping up to isomorphisms below captures these two approaches of type comparison:

Definition 2 (Signature inclusion up to isomorphims):

A signature S is a subtype up to isomorphism of another signature S', written $S <_{\simeq} S'$, if there exists signatures T and T' such that: $S \simeq T <: T' \simeq S'$.
□

In the two following sections we formalize a decision algorithm for the relation $<_{\simeq}$ between two signatures. To do so, we show how to pick in the equivalence class of a given signature a suitable canonical form over which the problem of deciding $<_{\simeq}$ become much simpler. We start by defining these canonical forms by means of inference systems formalized in a natural semantics style, for which we prove the usual properties of definedness and determinacy.

5 Transformation into canonical forms

We transform signatures into canonical forms that preserve equivalence with respect to the theory $\simeq$. The first step in the transformation is to introduce marks that turns names into identifiers. This is necessary in order to avoid name capture during subsequent transformations. A signature like:

$$\texttt{sig type } t\texttt{; module } x\texttt{: sig type } t \texttt{ = int end val } y\texttt{: } t \texttt{ end}$$

becomes:

$$\texttt{sig type } t_1\texttt{; module } x_1 : \texttt{sig type } t_2 = \texttt{int end val y :} t_1 \texttt{ end}$$

Marks can be straightforwardly introduced following scoping rules of the module calculus: we omit the details here. Once the marks are introduced, three consecutive steps are performed during the transformation into canonical forms: first, inner signatures are flattened ($\rightarrow_{flat}$), second, manifest type declarations are propagated ($\rightarrow_{exp}$), and finally, product types are split in value and type specifications ($\rightarrow_{split}$).
The transformation rules are described below in the Natural Semantics style [9].

5.1 Signature flattening

Transformation rules for flattening are shown in figure 1. Inner structures are flattened by rules 1 and 3. Rule 1 renames local identifiers into paths. Rule 4 performs curry-uncurry transformation. The result signature contains only value, type and functor specifications (but no structures), where functors have a single argument that has been flattened (as well as the body). In the example below, the signature:

```
sig
 module X:
```

```
    sig
     type t = int
     val v: t
    end
  val w: X.t
  module z (X: sig type t = string end) (Y: sig type t end):
    sig
    type t = X.t * Y.t
    end
 end
```

becomes after flattening:

```
 sig
  type x.t = int
  val x.v: x.t
  val w: x.t
  module z (U: sig type x.t = string type y.t end):
    sig
    type t = U.x.t * U.y.t
    end
 end
```

$$\frac{D\{n_i \leftarrow x_j.n \mid n_i \in BV(D)\} \rightarrow_{flat} D'}{\texttt{module } x_j : \texttt{sig } D \texttt{ end} \rightarrow_{flat} D'} \quad (1) \qquad \frac{S \text{ functor signature} \qquad S \rightarrow_{flat} S'}{\texttt{module } x_i : S \rightarrow_{flat} \texttt{module } x_i : S'} \quad (2)$$

$$\frac{C_i \rightarrow_{flat} D_i \qquad \text{for } i \in \{1, \ldots, n\}}{C_1; \ldots; C_n \rightarrow_{flat} D_1; \ldots; D_n} \quad (3)$$

$$\frac{\begin{array}{c}\texttt{module } x_i : S_1; \texttt{module } y_j : S_2 \rightarrow_{flat} D \\ S = S_3\{n_j \leftarrow z_k.n \mid n_j \in BV(D)\} \qquad \texttt{functor}(z_k : \texttt{sig } D \texttt{ end})S \rightarrow_{flat} F\end{array}}{\texttt{functor}(x_i : S_1)\texttt{functor}(y_j : S_2)S_3 \rightarrow_{flat} F} \quad (4)$$

$$\frac{D \rightarrow_{flat} D_1 \qquad D' \rightarrow_{flat} D'_1}{\texttt{functor}(x_i : \texttt{sig } D \texttt{ end})\texttt{sig } D' \texttt{ end} \rightarrow_{flat} \texttt{functor}(x_i : \texttt{sig } D_1 \texttt{ end})\texttt{sig } D'_1 \texttt{ end}} \quad (5)$$

Figure 1: Signature flattening

5.2 Type expansion

Type expansion rules are shown in figures 2 and 3. They use a type context that contains manifest type declarations: the expansion of S into S' in the type context A is written $A \vdash S \rightarrow_{exp} S'$. During the expansion, each occurrence of a type (path) identifier is replaced by its definition if there exists a manifest one in the current context. For instance, after type expansion, the previous flattened signature becomes:

```
 sig
  type x.t = int
  val x.v: int
  val w: int
  module z (u: sig type x.t = string type y.t end):
    sig
    type t = string * u.y.t
```

```
    end
end
```

$$A; \texttt{type}\ t_i = \tau \vdash t_i \rightarrow_{exp} \tau \quad (6) \qquad \frac{A \vdash \tau \rightarrow_{exp} \tau'}{A \vdash C[\tau] \rightarrow_{exp} C[\tau']} \quad (7)$$

$$A; \texttt{module}\ x_i : \texttt{sig}\ D_1;\ \texttt{type}\ t_j = \tau; D_2\ \texttt{end} \vdash x_i.t \rightarrow_{exp} \tau \quad (8)$$

Figure 2: Base type expansion

$$\frac{A \vdash \tau \rightarrow_{exp} \tau'}{A \vdash \texttt{type}\ t_i = \tau \rightarrow_{exp} \texttt{type}\ t_i = \tau'} \quad (9) \qquad A \vdash \texttt{type}\ t_i \rightarrow_{exp} \texttt{type}\ t_i \quad (10)$$

$$\frac{A \vdash \tau \rightarrow_{exp} \tau'}{A \vdash \texttt{val}\ v_i : \tau \rightarrow_{exp} \texttt{val}\ v_i : \tau'} \quad (11) \qquad \frac{A \vdash S \rightarrow_{exp} S' \quad S \text{ functor signature}}{A \vdash \texttt{module}\ x_i : S \rightarrow_{exp} \texttt{module}\ x_i : S'} \quad (12)$$

$$\frac{A \vdash S \rightarrow_{exp} S_1 \quad A; \texttt{module}\ x_i : S_1 \vdash S' \rightarrow_{exp} S'_1}{A \vdash \texttt{functor}\ (x_i : S)S' \rightarrow_{exp} \texttt{functor}\ (x_i : S_1)S'_1} \quad (13)$$

$$\frac{A; C_1; \ldots; C_{i-1} \vdash C_i \rightarrow_{exp} C'_i}{A \vdash \texttt{sig}\ C_1; \ldots; C_n\ \texttt{end} \rightarrow_{exp} \texttt{sig}\ C'_1; \ldots; C'_n\ \texttt{end}} \quad (14)$$

Figure 3: Module type expansion

5.3 Splitting product types

Rules for type splitting are shown in figure 4. We write $S \rightarrow_{split} S'$ the transformation of a signature S into a signature S' via these rules. During this transformation, each manifest type declaration isomorphic to a product type is split in a sequence of atomic declarations, as showed by the following example:

$$(\texttt{type}\ t = \texttt{int} * \texttt{bool}) \rightarrow_{split} (\texttt{type}\ t_1 = \texttt{int}; \texttt{type}\ t_2 = \texttt{bool})$$

Also, each value specified with a type isomorphic to a product type is split in a sequence of atomic value specifications:

$$(\texttt{val}\ a : \texttt{int} * \texttt{bool}) \rightarrow_{split} (\texttt{val}\ a_1 = \texttt{int}; \texttt{val}\ a_2 = \texttt{bool})$$

Side conditions in rules 15 and 16 guarantee that new names do not appear already in the signature being transformed.

5.4 Canonical forms

We transform a signature S into a signature S', written $S \rightarrow_{norm} S'$, by applying the sequence of the three rule systems described above:

$$\frac{S \rightarrow_{flat} S_1 \quad \emptyset \vdash S_1 \rightarrow_{exp} S_2 \quad S_2 \rightarrow_{split} S'}{S \rightarrow_{norm} S'}$$

$$\frac{\tau \approx_{iso} \tau_1 * \ldots * \tau_n \qquad t_1, \ldots, t_n \text{ fresh identifiers}}{\texttt{type } t = \tau \rightarrow_{split} \texttt{type } t_1 = \tau_1; \ldots; \texttt{type } t_n = \tau_n} \quad (15)$$

$$\frac{\tau \approx_{iso} \tau_1 * \ldots * \tau_n \qquad v_1, \ldots, v_n \text{ fresh identifiers}}{\texttt{val } v : \tau \rightarrow_{split} \texttt{val } v_1 : \tau_1; \ldots \texttt{val } v_n : \tau_n} \quad (16) \qquad \frac{S \rightarrow_{split} S' \qquad S \text{ functor signature}}{\texttt{module } f : S \rightarrow_{split} \texttt{module } f : S'} \quad (17)$$

$$\frac{S \rightarrow_{split} S_1 \qquad S' \rightarrow_{split} S'_1}{\texttt{functor}(x : S)S' \rightarrow_{split} \texttt{functor}(x : S_1)S'_1} \quad (18)$$

$$\frac{C_i \rightarrow_{split} D_i}{\texttt{sig } C_1; \ldots; C_n \texttt{ end} \rightarrow_{split} \texttt{sig } D_1; \ldots; D_n \texttt{ end}} \quad (19)$$

Figure 4: Product type splitting

Proposition 1 (Definedness)

Given a signature S, there exists a signature S' such that $S \rightarrow_{norm} S'$.
Proof:
the proposition is proved by induction on the structure of S. □

The following proposition guarantees the uniqueness of canonical forms up to renaming of components.

Proposition 2 (Determinacy):

Given a signature S, if $S \rightarrow_{norm} S'$ and $S \rightarrow_{norm} S''$ then $S' = S''$ up to renaming of components. (We assume that any base type equivalence of the form $\sigma \approx_{\texttt{iso}} \sigma_1 \times \ldots \sigma_n$ is in canonical form). □

Uniqueness of canonical forms allow us to introduce the following notation: we write $C_f(S) = S'$ when $S \rightarrow_{norm} S'$. Transformation into canonical forms preserves both the strict inclusion relation <: and the relation $\simeq$ of module type equivalence:

Proposition 3 ($\rightarrow_{norm}$ preserves <:)

Given two signatures S and S_1, if $S <: S_1$ and $S \rightarrow_{norm} S'$ and $S_1 \rightarrow_{norm} S'_1$, then $\exists S'', S'' \cong_r S', S'' <: S'_1$.

Proof

We assume that during the splitting process, generation of new names is deterministic, that is, each time an identifier is split into new names in S, the same name is split into the same new names in S_1. The proof follows by induction in the structure of S and S_1: if $S <: S_1$, then one can mimic on S_1 each reduction step necessary to transform S into its canonical form, maintaining the <: relation. The only case when this parallel splitting cannot be performed is when the bigger signature contains an abstract type t matched in the smaller signature by a manifest type $t = \tau$ which can be split. But then the reordering theory allows us to find an equivalent signature S'' where this splitting is regrouped, recovering th subtyping relation. □

Proposition 4

If $S \rightarrow_{norm} S'$ then $S \simeq S'$.

Proof

We first show three properties corresponding to the different steps in the transformation into canonical form:

1. If $S \to_{flat} S'$ then $S \simeq S'$.
2. If $A \vdash S \to_{exp} S'$ then $S \simeq_A S'$.
3. If $S \to_{split} S'$ then $S \simeq S'$.

Proof of 1,2,3: These three propositions are easily showed by induction on the structure of S, but property 2 needs the definition of a theory $\simeq_A$ that extends $\simeq$ with base type equivalences induced by manifest types declarations in the type context A. For instance, if A contains the declaration `type t = int`, then `t` is equivalent to `int` (`t` $\simeq_A$ `int`) and also `t * string` is equivalent to `int * string`. We can show that any property holding in $\simeq$ holds also in the theory $\simeq_A$. □

Proposition 4 follows immediately from properties *1,2,3* and from transitivity of $\simeq$. □

Finally the following results give us a way to relate canonical forms and the $\simeq$ equivalence relation:

Theorem 1 (Relating $\simeq$ and canonical forms):

Given two signatures S and S_1, if $S \simeq S_1$ and $S \to_{norm} S'$ and $S_1 \to_{norm} S_1'$ then $S' = S_1'$ up to renaming and re-ordering of components. □

6 The decision algorithm $\mathcal{A}_{\simeq}$

The equivalence checking algorithm we show in this section proceeds roughly in three steps: first, signatures to be compared are transformed into normal forms by application of the $\to_{norm}$ rewriting; second, both normal forms are renamed into signatures with a common set of names; finally, renamed signatures are compared using Leroy's strict inclusion relation <: modified as described in the appendix to handle isomorphic types of components. The two last steps are repeated until success or until all (equivalent) renamings have been considered.

Given two signatures S_u (for user's signature) and S_l (for a signature in the library), the algorithm $\mathcal{A}_{\simeq}(S_u, S_l)$ is defined by

Let	$S_u' = C_f(S_u)$ and $S_l' = C_f(S_l)$
and	N be a set of fresh names of size $\mathcal{M}ax(\#S_u', \#S_l')$,
	and such that $N \cap Dom(S_u') = N \cap Dom(S_l') = \emptyset$
and	the set of all pairs of renamings $\mathcal{R} = \{(\rho, \nu) \mid \rho : Dom(S_u') \to N, \nu : Dom(S_l') \to N\}$
if	$\exists(\sigma_1, \sigma_2) \in \mathcal{R}$
and	$\exists S_l'' \cong_r S_l'$
such that	$\sigma_2(S_l') <: \sigma_1(S_u')$ then true else false

□

Proposition 5(Termination)

The algorithm $\mathcal{A}_{\simeq}$ always terminates
Proof
Since the input signatures are finite, so are the name spaces of their canonical forms, due to definedness of the normalization rules. Now, the modified relation for <: is still decidable because we use a base language isomorphisms theory which is decidable, and this relation is tested only a finite number of times (equal to the number of possible pairs of renamings). □

Theorem 2(Soundness and completeness of the algorithm)
Given two signatures S_u and S_l, then $\mathcal{A}_{\simeq}(S_u, S_l) \Longleftrightarrow S_l <_{\simeq} S_u$ holds.

Proof

Correctness: we assume that $\mathcal{A}_{\simeq}(S_u, S_l)$ holds, let us prove $S_l <_{\simeq} S_u$. By hypothesis, there exists renamings σ_1 and σ_2 such that $\sigma_2(S_l') <: \sigma_1(S_u')$, where $S_u' = C_f(S_u)$ and $S_l' = C_f(S_l)$. Furthermore we have $S_l \simeq S_l'$ and $S_u \simeq S_u'$ (from Proposition 4). Because σ_1 and σ_2 are renamings towards fresh names, we also have $S_u' \simeq \sigma_1(S_u')$ and $S_l' \simeq \sigma_2(S_l')$. Then it follows that $S_l \simeq S_l' \simeq \sigma_2(S_l') <: \sigma_1(S_u') \simeq S_u' \simeq S_u$, which implies $S_l <_{\simeq} S_u$.

Completeness: we assume that $S_l <_{\simeq} S_u$ holds, let us prove then $\mathcal{A}_{\simeq}(S_u, S_l)$ yields true. ¿From the hypothesis, there exist two signatures L and U such that $S_l \simeq L <: U \simeq S_u$. As $\rightarrow_{norm}$ preserves type inclusion (by proposition 3) between L and U, we have $C_f(L) <: C_f(U)$. The property of uniqueness of the normal form (proposition 2) allows us to write $C_f(S_l) = C_f(L)$ and $C_f(S_u) = C_f(U)$ modulo renaming and re-ordering of components. Then $\mathcal{A}_{\simeq}(S_u, S_l) = \text{true}$. $\square$

7 Module converters

In this section we outline the procedure that is used to automatically build module converters. The converters are built incrementally following the steps of the transformation of signatures into canonical form.

To each step in the transformations, and to the substitution step used in the algorithm, it is possible to associate a pair of simple functors converting back and forth between the signatures before and after the transformation or the substitution. More precisely:

flattening the converters are built out of the input and output signatures by writing a very simple body properly linking corresponding components

type expansion the converters are identity functors plus a restriction on the output signature (the ML system does all the work): if S expands to S', then the converters are simply `module cvt = functor(X:S) = (X:S')` and `module cvt' = functor(X:S') = (X:S)`

type splitting these are the only nontrivial converters: here for each split value declaration, we need to take the converter functions coming from the theory of isomorphisms for the base type language and plug them in in the appropriate components

substitution the converters are simply renaming functors

subtyping the modified procedures for checking <: now allows components with the same name to have isomorphic types, instead of identical ones, and returns a converter which is essentially the identity functor, but for some base language type converters plugged in where necessary

We can summarize the work done by the algorithm in the following diagrams, where the stepwise converters actually used are made explicit:

$$
\begin{tikzcd}
S \arrow[d, shift left, "F_f"] & & flat & T \arrow[d, shift left] \\
{} \arrow[u, shift left] \arrow[d, shift left, "F_e"] & & exp & {} \arrow[u, shift left, "G_f"] \arrow[d, shift left] \\
{} \arrow[u, shift left] \arrow[d, shift left, "F_s"] & & split & {} \arrow[u, shift left, "G_e"] \arrow[d, shift left] \\
S' \arrow[u, shift left] \arrow[r, shift left, "F_r"] & {} \arrow[l, shift left] \arrow[r, shift left, "R"] & S'' \overset{H}{\longrightarrow}\!\!\!\!\!\!\!\!\!\!\; <: T'' \arrow[l, shift left] \arrow[r, shift left, "G_r"] & T' \arrow[l, shift left] \arrow[u, shift left, "G_s"]
\end{tikzcd}
$$

Having these functors at hand, it is easy to build the final converter as

$$\texttt{module Converter}\ (X : S) = G_f(G_e(G_s(G_r(H(R(F_r(F_s(F_e(F_f(X))))))))))$$

Where $F_r(F_s(F_e(F_f(X))))$ has signature S'', which is a subtype of T'' via the converter H, so can be passed in argument to the G functors.

8 A prototype implementation

We have implemented a prototype based on `Ocaml 1.05` [8] that checks for subtyping up to isomorphisms. This prototype has been built in a rather modular way following the steps described in the paper, yet it uses rather sophisticated machinery: we use part of the routines of `camlsearch` that implement a check of isomorphisms for the base language types as described in [5, 6], augmented with a simple lambda calculus notation used to build converters, which are not provided by `camlsearch`. In order to make these converters human readable, a necessity, since they will appear in the final conversion functors, we use a normalizer that incorporates some of the rules described in [7, 4].
The flattening, expansion and splitting phases follow closely the definition of the rules given in the paper. Finally, To check for subtyping, we rely on the internal routines of the `Ocaml` compiler.

The current implementation does not incorporate yet, at the time of the submission, any heuristic to cut down the search space of renaming (that can be quite big), so it is only usable for the small examples like the ones in this paper (that actually are properly checked by our prototype). We are currently studying various heuristics to make the system more efficient, on which we will comment in the final version of the paper.

9 Concluding remarks

We have presented a general notion of module type equivalence and subtyping for a very general and powerful module calculus, motivated by the need of providing a sound a satisfactory basis to the construction of efficient and powerful search tools for modern module libraries. We have given both a simple equational presentation of the equivalence and an effective, sound and complete algorithm for deciding the relaxed subtyping relation, which has been actually implemented on top of the `Ocaml` system. We believe that these results, both theoretical and practical, greatly improve over previous propositions and are quite satisfactory. Yet, they open up the way to new developments, in combination with some existing ideas in the field: we think for example of the automatic composition of modules proposed by Parot [15], that would give a very powerful tool when combined with our definition, or the extension to object systems and libraries.

References

[1] María Virginia Aponte. Extending records typing to type parametric modules with sharing. In *20th symposium on Principles of Programming Languages*, 1993.

[2] María Virginia Aponte and Roberto DiCosmo. Type isomorphisms for module signatures. In *PLILP'96, Lectures Notes in Computer Science*, volume 1140, pages 334–346. Springer Verlag, September 1996.

[3] Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction system for the λ-calculus with surjective pairing and terminal object. In Leach, Monien, and Artalejo, editors, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, volume 510 of *Lecture Notes in Computer Science*, pages 291–302. Springer-Verlag, July 1991.

[4] Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction system for the λ-calculus with surjective pairing and terminal object. *Journal of Functional Programming*, 6(2):299–327, 1996. This is an extended and revised version of [3].

[5] Roberto Di Cosmo. Type isomorphisms in a type assignment framework. In *19th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 200–210. ACM, 1992.

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

[7] Roberto Di Cosmo and Delia Kesner. A confluent reduction for the extensional typed λ-calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, editor, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, volume 700 of *Lecture Notes in Computer Science*, pages 645–656. Springer-Verlag, July 1993.

[8] Xavier Leroy et all. *The Objective Caml system, documentation and user guides.* INRIA, 1997. `http://pauillac.inria.fr/ocaml/htmlman/`.

[9] Gilles Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Science*, 1987.

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

[11] Xavier Leroy. Le système caml special light: modules et compilation efficace en caml. Rapport de recherche 2721, INRIA, November 1995.

[12] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[13] Brian Matthews. Reusing functional code using type classes for library search. Systems Engineering, Rutherford Appleton Laboratory, Didcot, OXON, OX11 0QX, U.K., e-mail: bmm@inf.rl.ac.uk.

[14] Roberto Muller. A software library architecture based on ml module signatures. Available as `ftp://ftp.cs.bc.edu/pub/muller/library.ps`.

[15] Patrick Parot. Automatisation d'une bibliothèque de modules. In *Journées Francophones des Langages Applicatifs*, pages 75–98, 1995.

[16] Mikael Rittri. Retrieving library identifiers by equational matching of types in 10th Int. Conf. on Automated Deduction. *Lecture Notes in Computer Science*, 449, July 1990.

[17] Mikael Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation.* PhD thesis, University of Göteborg, Göteborg, Sweden, 1990.

[18] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[19] Eugene R. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Eighth International Conference on Logic Programming*, pages 173–187. MIT Press, 91.

[20] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, 1991.

[21] Sergei V. Soloviev and Alexander E.Andreev. Linear isomorphism of types: a low upper bound for complexity. Technical report, BRICS reports in Computer Science, 1994.

[22] Amy Moormann Zaremsky and Jeannette M. Wing. Signature matching: a key to reuse. In *SIGSOFT*, December 1993. Also available as CMU-CS-93-151, May 1993.

A Appendix

We present Leroy's rules [10, 12] for verifying subtyping on module types. They use an equivalence relation $\approx$ on base types induced by manifest type declarations that we show in figure 5. Subtyping rules are given in figure 6. We use this exact set of rules with only two differences: first, we assume the use of path names there any time when simple identifiers are used in the original rules; second, we extend the original equivalence relation on manifest types $\approx$ to our equivalence relation $\approx_{iso}$ on base types modulo base type isomorphisms, and we consider subtyping between base types implies isomorphisms on base types.

Rule 2 uses the sub-system of module typing presented in [10], but we actually need a weaker version of this rule that does not need this sub-system:

$$E; \texttt{module}\ p_i : \texttt{sig}\ D_1;\ \texttt{type}\ t_j = \tau; D_2\ \texttt{end} \vdash p_i.t \approx \tau\{t_n \leftarrow p_i.t \mid t_n \in BV(D_1)\}$$

This version of the rule is powerful enough in our system, as we compare by subtyping only normalized signatures: the only case where inner module signatures are compared is when comparing functor arguments. By rule 9, the inner module type is added to the typing context, which is exactly the condition of the rule above.

$$E; \texttt{type}\ t_i = \tau \vdash t_i \approx \tau \quad (1) \qquad \frac{E \vdash p : \texttt{sig}\ D_1;\ \texttt{type}\ t_j = \tau; D_2\ \texttt{end}}{E \vdash p.t \approx \tau\{t_n \leftarrow p.t, x_k \leftarrow p.x \mid q_n, x_k \in BV(D_1)\}} \quad (2)$$

Figure 5: Manifest type equivalence

$$\frac{\sigma : \{1,\ldots,m\} \mapsto \{1,\ldots,n\} \quad E; C_1;\ldots;C_n \vdash C_{\sigma(i)} <: C'_i \text{ for } i \in \{1,\ldots,m\}}{E \vdash \texttt{sig}\ C_1;\ldots;C_n\ \texttt{end}\ <: \texttt{sig}\ C'_1;\ldots;C'_m\ \texttt{end}} \quad (3)$$

$$\frac{E \vdash \tau <: \tau'}{E \vdash \texttt{val}\ v_i : \tau <: \texttt{val}\ v_i : \tau'} \quad (4)$$

$$E \vdash \texttt{type}\ t_i <: \texttt{type}\ t_i \quad (5)$$

$$E \vdash \texttt{type}\ t_i = \tau <: \texttt{type}\ t_i \quad (6)$$

$$\frac{E \vdash t_i \approx \tau}{E \vdash \texttt{type}\ t_i <: \texttt{type}\ t_i = \tau} \quad (7)$$

$$\frac{E \vdash \tau \approx \tau'}{E \vdash \texttt{type}\ t_i = \tau <: \texttt{type}\ t_i = \tau'} \quad (8)$$

$$\frac{E \vdash S <: S_1 \qquad E; \texttt{module}\ x : S \vdash S' <: S'_1}{E \vdash (\texttt{module}\ f : \texttt{functor}(x : S)S') <: (\texttt{module}\ f : \texttt{functor}(x : S_1)S'_1)} \quad (9)$$

Figure 6: Subtyping rules