Type isomorphisms for module signatures

María-Virginia Aponte[1] and Roberto Di Cosmo[2]

[1] CNAM Conservatoire National des Arts et Métiers - 292, rue St. Martin - 75003 Paris France
E-mail:aponte@cnam.fr
[2] DMI-LIENS (CNRS URA 1347) Ecole Normale Supérieure - 45, Rue d'Ulm - 75230 Paris CEDEX 05 - France
E-mail:dicosmo@dmi.ens.fr

Abstract. This work focuses on software reuse for languages equipped with a module system. To retrieve modules from a library, it is quite reasonable to use module signatures as a search key, up to a suitable notion of signature isomorphism.

We study a formal notion of isomorphism for module signatures, which naturally extends the notion of isomorphism for types in functional languages. Isomorphisms between module types surprisingly have non-trivial interactions with the theory of isomorphisms of the base language to which the module system is added. We investigate the power of this notion in equating module signatures, and we study its decidability. This work does not impose any limitative assumption on the module system as we handle type declarations in signatures, type sharing and higher order modules.

KEYWORDS: typing and structuring systems, programming environments, module systems, ML, retrieval of functions in function libraries.

1 Introduction

Powerful module systems such as SML's [11, 5, 12] or Ada's [13] are essential tools for code reuse. Reuse requires a method to find modules. We suggest to use signatures as search keys, up to type isomorphism, a notion which have been proven relevant as a basis for advanced library search tools in functional libraries [16], and additionally for proposing extensions to the ML type inference mechanism [2].

An isomorphism is one way to identify types t and w for which any object of type t can be coerced into an object of type w and conversely, without losing information. That is, t and w contain the same information, organized in a different way. There is a well-known example: the types $(t \times w) \to r$ and $t \to (w \to r)$ are equivalent.

A highly illustrating example of Rittri's shows how the same function can happen to have very different-looking types, depending on some unpredictable coding choices on the programmer's side (see table 1).

Language	Name	Type
ML of Edinburgh LCF	itlist	$\forall X.\forall Y.(X \to Y \to Y) \to List(X) \to Y \to Y$
CAML	list_it	"
Haskell	foldl	$\forall X.\forall Y.(X \to Y \to X) \to X \to List(Y) \to X$
SML of New Jersey	fold	$\forall X.\forall Y.(X \times Y \to Y) \to List(X) \to Y \to Y$
The Edinburgh SML Library	fold_left	$\forall X.\forall Y.(X \times Y \to Y) \to Y \to List(X) \to Y$

Table 1. Different typings for the same functionality

Nevertheless, these different types turn out to be *isomorphic* types, which was the basis for Rittri's idea [15]: when looking for a function with a given type, a search tool must retrieve all the functions with isomorphic types, and not just those with syntactically equal types. For this, we need a decidable theory of type isomorphisms.

The study of type isomorphism for the typed λ-calculus, both explicit (system F) and implicit (Core-ML) has already been solved [3], leading to complete and decidable axiomatizations of isomorphisms and to efficient library search tools based on types. Exemples of application are library search systems for functions in typed functional languages (like LML, Haskell and CamlLight). The CamlLight system [17], for example, comes equipped with such a tool, `camlsearch`; here are some typical examples of its use:

```
> camlsearch -s -e "string*int*int -> string" /usr/local/lib/caml-light/
sub_string : string -> int -> int -> string

> camlsearch -s -e "'a*('a->'b->'a)*'b list ->'a" /usr/local/lib/caml-light/
it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
list_it : ('b -> 'a -> 'a) -> 'b list -> 'a -> 'a
```

It is now clear that in the future, libraries will not be a flat set of functions, but rather a set of well-organized *modules*, each coming with its set of defined types and functions. In this framework, the necessity to retrieve programs (or modules) in an independent way from the unpredictable structuring and coding choices of the programmer, is an even greater challenge. We show below an example of two functors that we intuitively want to identify.

```
module UnifyCurry :
       functor (t:TERMS)
       functor (s:SUBSTITUTION with s.termtype = t.termtype):
       sig ... unify: t.termtype -> t.termtype -> s.substtype end

module UnifyUnCurry :
   functor (sig module t:TERMS
                module s:SUBSTITUTION with s.termtype = t.termtype end):
         sig ... findunifier: t.termtype * t.termtype -> s.substtype end
```

Table 2. Different typings and structuring for the same functionalities

The goal of the present work is to tackle this challenge, studying in a precise way how to combine SML-like modules and isomorphisms of types. We develop a notion of equivalence of module types (also called signatures in SML) that parallels the equivalence of types induced by type isomorphisms. This equivalence notion can be applied to the retrieval of software components in libraries of modules.

The key idea is to export the analogy between simple types with products and module types [1] to derive from each of the type isomorphisms for core-ML a similar module type equivalence. These equivalences capture structural modifications of modules that are significantly more complex than a simple reordering of components. We strive to give a *general* solution, handling explicitly type declarations in signatures, type sharing specifications and higher-order modules. We have chosen to work with Leroy's proposal for module systems [6], which is now implemented in the Objective Caml language (available as [10]). to be quite easier to work with than the original SML module system.

Let us finally mention that all the work done here can be immediately applied to module systems like ADA's packages and *generics*.

The remainder of this paper is organized as follows. In the next section we briefly recall the essential features of Leroy's module system. In section 3, we informally define the equivalence between module signatures, which is formalized in section 4. In section 5, we discuss the decidability of our notion of equivalence. Finally, we present conclusions and directions for future research.

2 The module system

Our signature language is a subset of the manifest module language developed by X. Leroy [6, 9]. This language is sintactically similar to SML modules [12, 7]. The module language described has two base constructs: basic modules (structures), and functions from modules to modules (functors). Signatures are the types of modules (structure signatures or functor signatures). Structure signatures are sequences of type specifications. Functor signatures are actually dependent types: the type of a functor body may refer to a type identifier $x.t$ coming from the functor argument x. The single keyword `module` introduces both structure and functor specifications. In the grammar of figure 1, v ranges over value identifiers, t over type identifiers and x over module names.

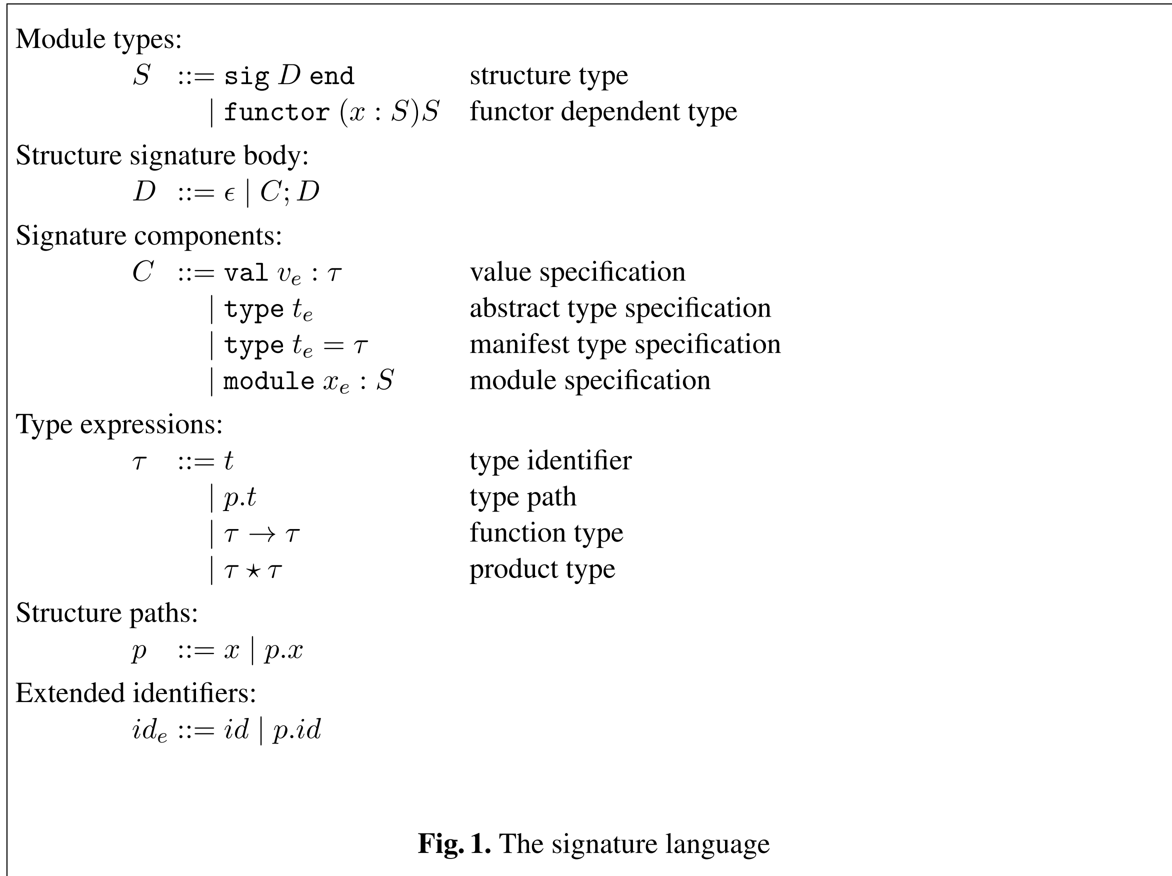

Fig. 1. The signature language

The structure path $m.x$ denotes the module component x of structure m. The path $p.x$ refers to the component x from an arbitrarily nested structure given by path p. Identifiers in structure paths must correspond to structure identifiers.

Type paths ($p.t$) and value paths ($p.v$) provide access to type and value identifiers from nested structures. Two occurrences of the path $p.t$ always correspond to the same type. On the other hand, manifest types can be equivalent even when coming from different type paths: they only have to be specified with (provably) equivalent type annotations.

We extend identifiers to paths: v_e, t_e and x_e are extended identifiers for values, types and modules. Extended identifiers cannot appear in a programmer defined signature. They are used during equivalence checking to rename identifiers bound in flattened signatures. We justify their use, and show examples, later in section 3.2. The syntactic class id_e introduces extended identifiers for the class id ranging on value, types and module identifiers.

Modules *à la* SML can be seen as a set of modular constructs (the module language) built on top of a statically-typed language (the base language). To keep this paper simple, we restrict the set of type constructors of the base language to products and functions.

3 Defining module isomorphism

Isomorphism between two types can be formalized using *converters*, that is, functions to coerce any object of one type into an object of the other type, as long as there is no loss of information:

Definition 1 (Isomorphic types) *Two types A and B are isomorphic iff there exist conversion* functions $f : B \to A$ *and* $f' : A \to B$ *such that*

$$\forall M : A, \quad f(f'\ M) = M : A \text{ and } \forall N : B, \quad f'(f\ N) = N : B.$$

Analogously, we give a semantic definition of signature isomorphism, as follows:

Definition 2 (Isomorphic module signatures) *Two module signatures S and S' are isomorphic iff there exist conversion* functors $f : S \to S'$ *and* $f' : S' \to S$ *such that*

$$\forall s' : S', \quad f(f'\ s') = s' : S' \text{ and } \forall s : S, \quad f'(f\ s) = s : S.$$

Characterizing isomorphisms of signatures is more complex than for simple types: we have to deal not with one language but two, the base language and the module language. In the remainder of this section, we take as starting point the results for isomorphism on the base types [2] and try to apply it to modules.

3.1 A first naive approximation

A simple analogy can be made between the SML module language and a typed functional language: structures correspond to basic values, functors to functions and signatures to types. Moreover, as structures are sequences of component declarations including other modules, they can be seen as named products of modules, values and types [1].

Thus, it is tempting to adapt the known isomorphism rules to module types built with named products and arrows, and then compare signatures componentwise using the already known results on isomorphism for base types [3]. Here is an example of two signatures that can be shown isomorphic in this way:

```
sig
 val filter:
    ('a->bool)->'a list->'a list
 val list_it:
    ('a->'b->'a)->'a->'b list->'a
end
```

```
sig
  val filter:
       ('a list)*('a->bool)->'a list
  val list_it:
      ('a*'b -> 'b)->'a list->'b->'b
end
```

Unfortunately, this simple technique only discovers few isomorphisms: essentially those on signatures having identical skeletons, including the same number and names of components (modulo some limited permutations) and without type declarations.

3.2 Equivalence via signature transformations

Our approach of equivalence relies on a simple idea already used for base language types: we consider that two signatures are equivalent if they can be transformed (preserving equivalence) into componentwise isomorphic signatures.

It is easy to associate to each one of these transformations a corresponding pair of conversion functors establishing an isomorphism. In this section we discuss informally the transformations and equivalences over signatures we aim to capture in a formal framework.

Names of components Few programmers choose exactly the same identifiers for module components with identical semantics. During equivalence verification we consider possible mismatch of names between components with isomorphic types.

Interaction between base and module types Structures in the module language can be seen as named products of simple types. When we do not impose the same names on type-isomorphic components, we can freely mix products from the module language (i.e, sequences of components) and products from the base language. The signatures below are semantically equivalent:

```
sig  val a: int*bool  end
sig  val a1: int; val a2: bool  end
```

A similar equivalence arises for type declarations. These two equivalences lead to the following transformations: we decompose value specifications, and manifest type specifications, that are annotated by a product type, into as many value or type specifications as there are components in the product type.

Structural equivalence Structures declare components (values, types, or modules), which can be organized in many different ways. We consider two kinds of equivalences between structurally different signatures. First, components with isomorphic types can be declared in different orders; second, they can be declared at different levels of sub-structure nesting. Thus, during equivalence checking we flatten all inner structures in signatures before comparing them componentwise.

Nevertheless, flattening of structure signatures must be performed carefully in order to preserve type dependencies over type identifiers bound within them. In the signature below, the type of `b` depends on the type `t` bound in the substructure `x`.

```
sig
  module x : sig type t = int; val a: t end
  type t = bool; val b: x.t
end
```

To flatten the inner module `x`, a natural solution is to lift all its declarations:

```
sig  type t = int;  val a: t;  type t = bool; val b: x.t  end
```

However, the path `x.t` in the type of `b` does not correspond any more to a bound type in this signature. To be correct, this transformation must also rename all references outside `x` of names bound within `x`. Yet this renaming can produce name clashes, and even change typing. Consider below the result of renaming the type name `x.t` into `t`: the type of `b` changes from `int` into `bool`. This simplistic transformation is erroneous:

```
sig  type t = int; val a: t; type t = bool; val b: t  end
```

To avoid this problem, we extend the syntax of identifiers to paths, and while lifting declarations from `x`, we rename every occurrence (even bounding ones) of any component `u` bound within `x` into the path `x.u`. The signature above is transformed into:

```
sig   type x.t = int; val x.a: x.t;  type t = bool; val b: x.t  end
```

Notice that this renaming remains local to `x`: the specification `type t = int` is modified into `type x.t = int`, and thus, any reference to `t` outside `x` remains (correct and) unchanged.

Curried functors In the base type language the following equivalence holds: $s_1 \to s_2 \to s \quad \equiv \quad s_1 \times s_2 \to s$. Analogously, we consider that functor signatures are equivalent if their (curried) arguments put together (as in products) are componentwise isomorphic. For instance, consider the functors `f` and `g` below:

```
module f :  functor(x: S1) functor(y: S2) S
module g :  functor(x: S3) S
```

We consider that they are equivalent if transforming functor `f` into

```
module f ' : functor (z: sig module x:S1; module y: S2 end) S
```

ends up in a functor `f'` such that its argument `z` has a signature componentwise isomorphic to `S`. This amounts to perform a transformation on curried functors analogous to that used for simple types: put in products all arguments of a functor. In module types, this corresponds to using a single argument having a substructure for each one of the functor arguments. Combined with flattening, this transformation can be used to check equivalence in presence of curried functors.

Equivalence of sharing specifications Type sharing specifications are used in functor signatures to express compatibility between type components of different arguments. Each functor below specifies sharing on two types `t` and `w` within theirs arguments.

```
signature S1 = sig type t=int end; signature S2 = sig type w=int end
module f : functor(x: S1) functor (y: sig type w = x.t end) S
module g : functor(y: S2) functor (x: sig type t = y.w end) S
```

We consider two functors equivalent if, when they specify sharing, the sharing specified is exactly the same on types equivalent across the functors. In this example, beacause `x.t` and `y.w` in `f` and `y.w` and `x.t` in `g` are all specified as `int`, they can be considered as equivalent across functors. Moreover, both functors specify exactly the same sharing, namely, the equivalence of the type components `x.t` and `y.w`.

Sharing is specified using manifest type specifications, and manifest type specifications induce an equivalence relation on type identifiers and congruences on type expressions. Thus, sharing specifications do not need a particular transformation during equivalence checking: they are simply taken in account as manifest type specifications in signatures. Thus, the two functor above can be proved equivalent after transforming them using the usual rules on functor equivalence (in particular those on curried funtors).

4 An axiomatization of signature isomorphism

We formalize the ideas presented above, by presenting a set of equivalence rules for *checking* signature isomorphism. The main goal of these rules is to say whether a set of equivalences E between two signatures S_1 and S_2 can be used to establish an isomorphism between them. This will be written:

$$E \vdash S1 \quad \equiv \quad S2$$

Module signatures can contain specifications for type components that can be referred to later on in the signature, and this type information may be necessary to determine equivalences for the remainder of the signature. Thus, our equivalence rules check whether two signatures are isomorphic *both* with respect to an *equivalence type context* E, which records equalities on type identifiers declared in the two signatures checked for isomorphisms, *and* with respect to two local contexts L and R, containing each the type equivalence information local to the left or right signature. Our rule system uses the following equivalence judgements:

$E, L, R \vdash S \quad \equiv \quad S'$	signature iso	$E \vdash \tau \; \mathtt{iso}_{base} \; \sigma$	base type iso
$E, L, R \vdash \tau \; \mathtt{iso} \; \tau'$	cross base type iso	$A \vdash \tau \approx \tau'$	local base type equiv

We classify these judgements into two kinds: those stating equivalence between types of two different signatures (either between type components, or between whole signatures), and those stating equivalence between (base language) types local to one of the compared signatures. We call the former *cross equivalences* and the latter *local equivalences*. Equivalence contexts are distinguished in a similar way: the equivalence context E is a cross context, while L (for left) and R (for right) are local contexts. The context A above stands for any of these local contexts.

Both cross and local equivalences are necessary to determine isomorphism between signatures, as shown in the example below:

```
signature S = sig  type t = int; val a: t * t  end
signature S' = sig  type w = int; val p : int * w  end
```

In order to prove `S` equivalent to `S'`, we must show a cross equivalence between the type identifiers `t` and `w`, and between the types `t * t` and `int * w`. The former is immediate, but for the latter we need to use type information local to each signature, namely, that `t` is specified as `int` in `S`, and `w` as `int` in `S'`. Thus, to show equivalence of components `a:t*t` and `p:int*w`, the cross context E contains $\{\mathtt{type\ t} = \mathtt{w}\}$, and the local contexts are $L = (\mathtt{type\ t} = \mathtt{int})$ and $R = (\mathtt{type\ w} = \mathtt{int})$.

In the following rules, we need an auxiliary normalisation function `norm` associating to each signature S a signature S' (its normal form), as detailed in figure 7. Finally, we will write $E \vdash S1 \quad \equiv \quad S2$ for $E, \emptyset, \emptyset \vdash S1 \quad \equiv \quad S2$.

4.1 The isomorphism rules

We give a formal axiomatic presentation of our rules for *checking* signature isomorphism. For this, we assume given a cross context E, containing already the appropriate cross equations to check the equivalence. We then use the various transformations discussed above to test equivalence with respect to E. Due to the symmetry of equivalence judgments, we will often present only the rule for the left case, with a name ending with a -L suffix. We will also often talk generically about a rule, with no suffix, when meaning both the right and the left rule.

The rules in figure 2 compare componentwise two signatures, reordering and renaming components. Rule (Comp-fun) compares functor components. An analogous rule for structure components is subsumed by the rule (Flatten), that also captures flattening as presented in section 3.2.

Rules in figure 3 deal with interactions between base-language product types and products implicit in sequences of signature components. Whenever we can use base-language isomorphisms to show that a component has a product type, we split it in two components. The rules in figure 4 show how the isomorphisms for the base language can be lifted to isomorphisms of the types that appear in the module signatures (these types can indeed contain path expressions, that are not part of the base language).
Rules in figure 5 account for the equivalence relation on types induced by manifest type specifications local to each compared signature [6].

Notation 3 (Fully substituted form) *The rules in figure 5 can be seen as a* function $\phi(A, \tau) = \tau'$, *that given a type τ and an environment A, performs all possible substitution of identifiers from A in τ.*

Rules in figure 6 deal with functor equivalences. These rules use normalization, which is only needed to flatten functor arguments before checking equivalence of bodies. Rule (Functor-decompose) compares two functors by comparing argument signatures and result signatures. In this rule, E contains the cross type equations (for type identifiers coming from both functor arguments) needed to verify equivalence on functor bodies. Here, normalization of arguments guarantees, first, that the correct type paths are held in E; and second, that type declarations in arguments are flattened. These are necessary steps to apply rule (Eq-type-path) on functor arguments when used as hypothesis to derive equivalences of bodies. Rules (Functor-uncurry) deal with curried functors as explained in section 3.2.

Semantic soundness We can now show that all our equivalence rules are semantically sound.

Proposition 4 Semantic soundness. *Assume that S, S' are module signatures such that $E \vdash S \quad \equiv \quad S'$. Then, there exist conversion functors $f : S \to S'$ and $f' : S' \to S$ such that:*

$$\forall s' : S', \quad f(f'\ s') = s' \textit{ and } \forall s : S, \quad f'(f\ s) = s.$$

Proof. To each of the equivalence rules one can associate a pair of conversion functors, as hinted in the informal discussion. Then one can build the conversion functors associated to a full proof by composing the simpler ones associated to each equivalence used in the derivation.

5 Deciding module isomorphism

From the definition in the previous section it is not really evident that, given a cross context E, the property $E \vdash S_1 \equiv S_2$ is decidable, as not all the rules transform a decision problem into a smaller one: notably, the Mix rules replace one component with *two* other ones, while splitting products.

Indeed, decidability depends in an essential way on the properties of isomorphisms of the base language. Consider the rule Mixtype-L: to prove $E, L, R \vdash S_1; \texttt{type}\ t = \tau; S \equiv S'$ one has first to establish $E, L; \texttt{type}\ t = t_1 * t_2, R \vdash S_1; \texttt{type}\ t_1 = \tau_1; \texttt{type}\ t_2 = \tau_2; S \equiv S'$ where $L \vdash \tau \approx \tau_1 * \tau_2$, i.e. $L \vdash \phi(L, \tau)\ \texttt{iso}_{base}\ \tau_1 * \tau_2$.

This rule reminds us, like all the Comp rules, that if isomorphism in the base language is not decidable, neither can be isomorphism at the module level.

But it also tells us that if we do not have enough information on τ_1 and τ_2, we are in the presence of a potential cause of nontermination: if the equivalence classes of the base language isomorphism are infinite (this is the case for second order isomorphisms, which are nevertheless decidable, as shown in [4]), or if either one of τ_1 or τ_2 is not strictly *smaller*, according to an appropriate measure, than τ (and this is the case in the presence of recursive types), then there may be an infinite number of derivations with the same root and the brute-force search for a legal derivation may not terminate. This fact highlights once more the tight relationship that exists between isomorphisms at the base language level, and isomorphisms of modules. If the base language isomorphisms are decidable in a sophisticated way that can cope with potentially infinite equivalence classes of isomorphic types, then the module isomorphisms will probably also be decidable, but by means of a specialized algorithm, which is not simply built on top of the existing decision procedure for the base language.

Nevertheless, here are some general necessary conditions for decidability.

Proposition 5 Decidability w.r.t. E. *Assume the following conditions:*

- *the equivalence classes of the base language isomorphism are finite (or pseudo-finite, i.e., finite up to alpha renaming of bound type variables as in ML and F), and*
- *there is a well founded measure m such that if $\tau \equiv \tau_1 * \tau_2$, then $m(\tau) > m(\tau_i)$, $i = 1, 2$.*

Then, signature isomorphism w.r.t. a given E is decidable.

Proof. (sketch) It is enough to show that the brute-force construction of all proof derivations, given E, terminates.

The first condition ensures that at any step only a finite number of subproblems is generated (the trick for handling pseudo-finiteness is to consider only isomorphic types with the same bound variables in the Mix rule, and only renaming of bound variables to already existing bound variables in the Comp rules). The second condition ensures that only a finite number of Mix rules can be used in a derivation (as each application of the rule decreases the overall measure of all types on which Mix can be applied). This is sufficient, as all other rules strictly decrease the size of the subproblems.

Remark. The conditions for termination are not as restrictive as they might seem: for example all theories of isomorphisms found in [3] satisfy them.

Now that we know how to *check* whether a given set of cross equivalences E is enough to derive the equivalence of the two signatures, what about *finding* all such E's?

Proposition 6 Decidability. *If signature isomorphism w.r.t. E is decidable, then so is the problem of finding all E such that $E \vdash S_1 \equiv S_2$*

Proof. The equations in E that are relevant for signature equivalence are only those involving the possible paths in the two given signatures. Hence we can use decidability of signature isomorphism checking w.r.t. a given E to build an algorithm for isomorphism checking: just try all the relevant E's.

It is possible to give a more refined algorithm to find all possible valid E for a given specific base language.

6 Conclusions and future work

We have presented a notion of module isomorphism that parallels the semantic notion of type isomorphism. This notion is at the same time conceptually simpler and technically richer than previous proposals (for example [18]).

We showed that, when dealing with module interconvertibility, one can no longer separate the module language from the base language, as long as the base language has types such as products or records, and that this fact can have important consequences on the decidability of the module isomorphisms: this is quite reasonable, and nevertheless essentially new.

We also showed how, using Leroy's formalism, sharing and higher order modules can fit nicely and naturally in our framework: the system presented here does not have the usual restrictions to first order modules, and does not sweep sharing (one of the most important features in this kind of modules) under the carpet. Nevertheless, like Leroy's calculus [6], our system does not account for the full transparency problem of higher-order functors. A solution would be to adapt our equivalence system to Leroy's applicative functors semantics [8], but this is still under investigation. Proceeding from here, we can now address some further problems, both on the practical and theoretical side:

Module matching up to isomorphism There are two orthogonal problems when reusing code from a library: one is the different representations of the same functionality (captured by our notion of isomorphism); the other is how to combine several pieces from the library to build the functionality that we need. While our work focuses on the first problem, there has been work dealing with the second: [14], for example, studies a language with higher-order parametric module signatures, and shows how to translate a library of higher order modules into a logic program. The execution of this program generates all possible compositions of modules that result in a module containing at least all components specified in the query. This approach, based on an extension of signature matching does not account for the problem of equivalence of representations, which is central to our work, nor for type sharing in signatures. Nevertheless, it would be interesting to combine the two approaches: this would require an axiomatization of subsumption up to isomorphism.

Semantic soundness and completeness An important feature of previous work on isomorphisms of types is the existence of a soundness and completeness proof for the isomorphism inference system (see [3] for a survey): this guarantees that only relevant information is retrieved, and that nothing is missed during a search. While semantic soundness for our set of rules can be shown easily using the conversion functors associated to each inference rule, semantic completeness is a much more difficult task: even for the base language types for the systems presented in [3], the proof relies on a very complex analysis of a special class of typed λ-terms. The most promising approach seems to try to exploit the analogy between module types and base language types to recover the completeness result for modules from completeness on base language type isomorphisms.

Acknowledgements

The authors would like to thank Mathias Felleisen and Xavier Leroy for pleasurable discussions and careful reading of the paper.

References

1. María Virginia Aponte. Extending records typing to type parametric modules with sharing. In *20th symposium on Principles of Programming Languages*, 1993.
2. 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.
3. Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995. ISBN-0-8176-3763-X.
4. Roberto Di Cosmo. Second order isomorphic types. A proof theoretic study on second order λ-calculus with surjective pairing and terminal object. *Information and Computation*, pages 176–201, June 1995.

5. Robert Harper, Robin Milner, and Mads Tofte. A type discipline for program modules. In *Theory and Practice of Programming Languages*, volume 250 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.
6. Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symposium on Principles of Programming Languages*, pages 109–122. ACM Press, January 1994.
7. Xavier Leroy. A syntactic approach to type generativity and sharing (extended abstract). In *Record of the 1994 ACM-SIGPLAN Workshop on ML and its Applications*, pages 1–12. INRIA, June 1994.
8. Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd symposium on Principles of Programming Languages*. ACM Press, January 1995.
9. Xavier Leroy. The Caml Special Light system. Technical report, INRIA, Roquencourt, Le Chesnay Cedex 78153, France, 1995. Available as `ftp://ftp.inria.fr/lang/caml-light/csl*`.
10. Xavier Leroy. The Objective Caml reference manual. Technical report, INRIA, Roquencourt, Le Chesnay Cedex 78153, France, 1996. Available as `ftp://ftp.inria.fr/lang/caml-light/ocaml*`.
11. David MacQueen. Modules for standard ML. In *ACM Symposium on Lisp and Functional Programming*, 1984.
12. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
13. Departement of Defense DoD. *Ada reference manual*. 1983.
14. Patrick Parot. Automatisation d'une bibliothèque de modules. In *Journées Francophones des Langages Applicatifs*, pages 75–98, 1995.
15. Mikael Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation*. PhD thesis, University of Göteborg, Göteborg, Sweden, 1990.
16. Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.
17. Pierre Weis and Xavier Leroy. *Le langage Caml*. InterÉditions, 1993.
18. 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.

(Paren)

$$\frac{E, L, R \vdash D \quad \equiv \quad D'}{E, L, R \vdash \texttt{sig}\ D\ \texttt{end} \quad \equiv \quad \texttt{sig}\ D'\ \texttt{end}}$$

(Comp-value)

$$\frac{E, L, R \vdash \tau\ \texttt{iso}\ \tau' \qquad E, L, R \vdash D_1; D_2 \quad \equiv \quad D_1'; D_2'}{E, L, R \vdash D_1; \texttt{val}\ a : \tau; D_2 \quad \equiv \quad D_1'; \texttt{val}\ b : \tau'; D_2'}$$

(Comp-type)

$$\frac{\begin{array}{c} E, L, R \vdash \tau\ \texttt{iso}\ \tau' \qquad E \supseteq \{\texttt{type}\ t = r\} \\ E, L; \texttt{type}\ t = \tau, R; \texttt{type}\ r = \tau' \vdash D_1; D_2 \quad \equiv \quad D_1'; D_2' \end{array}}{E, L, R \vdash D_1; \texttt{type}\ t = \tau; D_2 \quad \equiv \quad D_1'; \texttt{type}\ r = \tau'; D_2'}$$

(Comp-abs-type)

$$\frac{E, L, R \vdash D_1; D_2 \quad \equiv \quad D_1'; D_2' \qquad E \supseteq \{\texttt{type}\ t = r\}}{E, L, R \vdash D_1; \texttt{type}\ t; D_2 \quad \equiv \quad D_1'; \texttt{type}\ r; D_2'}$$

(Comp-fun)

$$\frac{\begin{array}{c} E, L, R \vdash F \quad \equiv \quad F' \qquad F, F' \text{ functor signatures} \\ E, L, R \vdash D_1; D_2 \quad \equiv \quad D_1'; D_2' \end{array}}{E, L, R \vdash D_1; \texttt{module}\ f : F; D_2 \quad \equiv \quad D_1'; \texttt{module}\ f' : F'; D_2'}$$

(Flatten-L)

$$\frac{E, L, R \vdash D_1; D\{n \leftarrow x.n \mid n \in BV(D)\}; D_2 \quad \equiv \quad S}{E, L, R \vdash D_1; \texttt{module}\ x : \texttt{sig}\ D\ \texttt{end}; D_2 \quad \equiv \quad S}$$

Fig. 2. Structure compatibility rules

(Mixvalue-L)

$$\frac{E, L, R \vdash D_1; \texttt{val}\ a_1 : \tau_1; \texttt{val}\ a_2 : \tau_2; D_2 \quad \equiv \quad S \qquad L \vdash \tau \approx \tau_1 * \tau_2}{E, L, R \vdash D_1; \texttt{val}\ a : \tau; D_2 \quad \equiv \quad S}$$

(Mixtype-L)

$$\frac{\begin{array}{c} E, L; \texttt{type}\ t = t_1 * t_2, R \vdash D_1; \texttt{type}\ t_1 = \tau_1; \texttt{type}\ t_2 = \tau_2; D_2 \quad \equiv \quad S \\ L \vdash \tau \approx \tau_1 * \tau_2 \end{array}}{E, L, R \vdash D_1; \texttt{type}\ t = \tau; D_2 \quad \equiv \quad S}$$

Fig. 3. Mix products rules

(Iso-E)

$$E \cup \{\texttt{type}\ t = w\} \vdash t\ \texttt{iso}_{base}\ w$$

(Iso-ELR)

$$\frac{L \vdash \tau \approx \tau' \qquad R \vdash \sigma \approx \sigma' \qquad E \vdash \tau'\ \texttt{iso}_{base}\ \sigma'}{E, L, R \vdash \tau\ \texttt{iso}\ \sigma}$$

Fig. 4. Base language isomorphisms

(Eq-type)

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

(Eq-type-path)

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

(Subst)

$$\frac{A \vdash \tau \approx \tau' \qquad A \vdash C[\tau'] \approx \sigma}{A \vdash C[\tau] \approx \sigma}$$

Fig. 5. Type equivalence rules

(Functor-decompose)

$$\frac{\begin{array}{c} N_1 = \texttt{norm}(S_1) \qquad N_2 = \texttt{norm}(S_2) \\ E, L, R \vdash \texttt{module}\ x : N_1 \quad \equiv \quad \texttt{module}\ y : N_2 \\ E, L; \texttt{module}\ x : N_1, R; \texttt{module}\ y : N_2 \vdash S_1' \quad \equiv \quad S_2' \end{array}}{E, L, R \vdash \texttt{functor}(x : S_1)S_1' \quad \equiv \quad \texttt{functor}(y : S_2)S_2'}$$

(Functor-uncurry-L)

$$\frac{\begin{array}{l} E, L, R \vdash\ \texttt{functor}(z : \texttt{sig}\ \texttt{module}\ x : S_1; \texttt{module}\ y : S_2\ \texttt{end}) \\ \qquad\qquad S_3\{x.n \leftarrow z.x.n \mid n \in BV(x)\}\{y.n \leftarrow z.y.n \mid n \in BV(y)\} \qquad \equiv S \end{array}}{E, L, R \vdash \texttt{functor}(x : S_1)\texttt{functor}(y : S_2)S_3 \quad \equiv \quad S}$$

Fig. 6. Functor equivalence rules

A The normalization rules

We present here the definition of the norm function that associates to a module signature its *normal form* (essentially flattening signatures using (Flatten)).

$$
\begin{aligned}
\mathtt{norm}(\mathbf{val}\ a : \tau; D) &\longrightarrow \mathbf{val}\ a : \tau; \mathtt{norm}(D) \\
\mathtt{norm}(\mathbf{type}\ t; D) &\longrightarrow \mathbf{type}\ t; \mathtt{norm}(D) \\
\mathtt{norm}(\mathbf{type}\ t = \tau; D) &\longrightarrow \mathbf{type}\ t = \tau; \mathtt{norm}(D) \\
\mathtt{norm}(\mathbf{module}\ x : \mathbf{sig}\ D\ \mathbf{end}; D') &\longrightarrow \mathtt{norm}(D\{n \leftarrow x.n \mid n \in BV(D)\}; D') \\
\mathtt{norm}(\mathbf{module}\ f : F; D) &\longrightarrow \mathbf{module}\ f : F; \mathtt{norm}(D) \\
&\qquad \text{if } F \text{ is a functor signature}
\end{aligned}
$$

Fig. 7. The rewriting rules for normalization

This article was processed using the LaTeX macro package with LLNCS style