A Modular Module System

Xavier Leroy

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

A modular module system

Xavier Leroy

N ° 2866

Avril 1996

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

ISSN 0249-6399

A modular module system

Xavier Leroy

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Cristal

Rapport de recherche n ° 2866 — Avril 1996 — 32 pages

Abstract: A simple implementation of a SML-like module system is presented as a module parameterized by a base language and its type-checker. This demonstrates constructively the applicability of that module system to a wide range of programming languages.

Key-words: Module systems, type systems, functors, sharing constraints, Caml, SML, ML.

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Un système de modules modulaire

Résumé : Ce rapport présente une implémentation simple d'un système de modules à la SML, sous la forme d'un module paramétré par la description d'un langage de base et du typeur associé. Cette implémentation démontre de manière constructive que ce système de modules s'applique à une large classe de langages de programmation.

Mots-clé : Systèmes de modules, systèmes de types, foncteurs, contraintes de partage, Caml, SML, ML.

1 Introduction

Modular programming can be done in any language, with sufficient discipline from the programmers [21]. However, it is facilitated if the programming language provides constructs to express some aspects of the modular structure and check them automatically: implementations and interfaces in Modula, clusters in CLU, packages in Ada, structures and functors in SML, classes in C++, ...

Even though modular programming has little to do with the particulars of any programming language, each of the languages above puts forward its own design of a module system, without reusing directly an earlier module system — as if the design of a module system were so dependent on the base language that transferring a module system from one language to another were impossible. Consider for instance the module system of SML [16, 20]. This is one of the most powerful module systems proposed so far, particularly for its treatment of parameterized modules as *functors*, i.e. functions from modules to modules; the SML module system is actually a small functional language of its own that operates over modules. The only published attempt at transferring it to another language is an adaptation to Prolog that did not receive much publicity [25]. What if one wants SML-style modules in another language? Say, Fortran?

Recent work on the type-theoretic foundations of SML modules [9, 12] has led to a reformulation of the SML module system as a type system that uses mostly standard notions from type theory. On these presentations, it is apparent that the base language does not really matter, as long as its compile-time checks can be presented as a type system. In particular, [12] presents an SML-style module system built on top of a typed base language left mostly unspecified; even though core ML is used as the base language when the need arises to be more specific, it is claimed that "the module calculus makes few assumptions about the base language and should accommodate a variety of base languages".

The purpose of the present paper is to give a highly constructive proof of that claim: we present an implementation of a SML-style module system as a functor parameterized by the base language and its associated type-checking functions. This implementation gives sufficient conditions for an existing or future language to support SML-style modules: if it can be cast in the shape specified by the input interfaces of the functor, then it can easily be equipped with SML-style modules.

An unusual aspect of this paper is that most of the actual source code is shown. The purpose is twofold: first, show that the amount of code that needs to be written for adding SML-style modules to an existing compiler is much lower than expected; second, provide a reference implementation of the module system, complementing its type-theoretic description. The experience with Hindley-Milner typing shows that typing rules do not always tell the whole story, and a simple implementation may help in understanding all the issues involved [1, 22]. For these two purposes, the implementation presented in this paper has been simplified as much as possible, but no more (to quote Einstein out of context).

The implementation presented in this paper is written in Caml Special Light [14], an extension of the Caml dialect of ML [27] with a module system extremely close to the one that is described here. The code itself exemplifies the module language that it implements,

in particular, the systematic use of functors – in the established tradition of meta-circular interpretors for Lisp. We hope that, far from making this paper impenetrable to readers unfamiliar with the theory and practice of SML-style modules, this circularity will actually help them gain some understanding of both.

Related work

Algebraic specifications can be viewed as highly base language-independent languages for expressing module interfaces, with parameterized specifications playing the role of functor signatures [28]. The algebraic approach is both stronger and weaker than the type-theoretic approach followed here: it supports equations, but not higer-order functions. Our approach also provides a base language-independent framework for relating an implementation to its interface, while in the case of algebraic specification this operation is often left implicit, or performed through intermediate languages specialized for a particular base language [8].

Cardelli [4] gives a formal treatment of linking and separate compilation, which is also highly independent of the base language. The emphasis is on separate compilation rather than on module languages; in particular, functors are not considered.

On the implementation side, MacQueen describes the New Jersey ML implementation of the SML module system [17] and its extension to higher-order functors [5]. Both implementations are considerably more sophisticated than the implementation described in this paper, in particular because much attention is paid to reducing memory requirements through term sharing.

MacQueen's implementations follow the stamp-based static semantics for SML modules [20, 18], which in itself can be viewed as an abstract implementation of a typechecker for the SML module system. However, this stamp-based semantics is not presented in isolation from the base ML language; in particular, stamps are strongly tied with the generativity of `datatype` definitions in ML, but do not reflect directly more universal notions such as type abstraction.

Cardelli's implementation of Quest [3] inspired important parts of the present work, such as the central role played by paths and the distinction between identifiers and names.

Outline

The remainder of this paper is organized as follows. Section 2 presents the functors implementing the module system. The reader more interested in the applicability of the module system to many base languages than in the features and implementation of the module language itself may concentrate only on subsections 2.4 and 2.7. Two applications are outlined in section 3, with core-ML and mini-C as base languages. Section 4 briefly discusses compilation issues. Section 5 discusses some extensions, in particular to deal with generative type definitions. Concluding remarks follow in section 6.

2 The modular module system

2.1 Identifiers

Most languages allow type names and variable names to be redefined. This may cause difficulties when typechecking uses type names: assuming a type name `t` and a variable `x` of type `t`, redefining `t` to be a different type invalidates the typing hypothesis `x : t`. Hence we need to distinguish between the two types named `t`. To this end, we introduce a notion of identifiers distinct from names: each identifier has a name, but it also records the binding location of this name; hence, two identifiers with the same name but different binding locations are distinct identifiers [3]. The abstract type of identifiers has the following signature:

```
module type IDENT =
  sig
    type t
    val new: string -> t
    val name: t -> string
    val equal: t -> t -> bool
    type 'a tbl
    val emptytbl: 'a tbl
    val add: t -> 'a -> 'a tbl -> 'a tbl
    val find: t -> 'a tbl -> 'a
  end
module Ident : IDENT = ... (* omitted *)
```

`new` returns a fresh identifier with the name given as argument; `name` returns the name of the given identifier; `equal` checks the equality (same binding location) of two identifiers. The parameterized type `'a tbl` implements applicative dictionaries associating identifiers to data of type `'a`; `add` returns the given dictionary enriched with an (identifier, data) pair; `find` retrieves the data associated with an identifier, raising the `Not_found` exception if the identifier is unbound.

Here is a sample implementation of `IDENT`, representing identifiers as pairs of a name and an integer stamp incremented at each `new` operation, and `'a tbl` as association lists (balanced binary trees would be better).

```
module Ident : IDENT =
  struct
    type t = {name: string; stamp: int}
    let currstamp = ref 0
    let new s = currstamp := !currstamp + 1; {name = s; stamp = !currstamp}
    let name id = id.name
    let equal id1 id2 = (id1.stamp = id2.stamp)
    type 'a tbl = (t * 'a) list
    let emptytbl = []
    let add id data tbl = (id, data) :: tbl
```

```
    let rec find id1 = function
        [] -> raise Not_found
      | (id2, data) :: rem -> if equal id1 id2 then data else find id1 rem
  end
```

2.2 Access paths

We refer to named types, values (variables), and modules either by identifier (if we are in the scope of their binding) or via the dot notation, e.g. `M.x` to refer to the `x` component of module `M`. The data type `path` represent both kinds of references:

```
type path =
    Pident of Ident.t                   (* identifier *)
  | Pdot of path * string               (* access to a module component *)
```

Since modules can be nested, paths may be arbitrarily long, e.g. `M.N.P.x`, which reads ((`M.N`).`P`).`x`. Notice that access to a module component is by name (the second argument of `Pdot` is a string) and not by identifier. The reason is that the access is not in the scope of the identifier binding. To avoid ambiguity, we require that all components of a module (at the same nesting level) have distinct names. Path equality extends naturally identifier equality:

```
let rec path_equal p1 p2 =
  match (p1, p2) with
    (Pident id1, Pident id2) -> Ident.equal id1 id2
  | (Pdot(r1, field1), Pdot(r2, field2)) -> path_equal r1 r2 & field1 = field2
  | (_, _) -> false
```

2.3 Substitutions

For typechecking modules, we will need to substitute paths for identifiers. Substitutions are defined by the following signature:

```
module type SUBST =
  sig
    type t
    val identity: t
    val add: Ident.t -> path -> t -> t
    val path: path -> t -> path
  end
module Subst : SUBST = ... (* omitted *)
```

`Subst.add` i p σ extends the substitution σ by $[i \leftarrow p]$. `Subst.path` p σ applies σ to the path p. Here is a sample implementation of `SUBST`, where substitutions are represented as dictionaries from identifiers to paths (type `path Ident.tbl`).

```
module Subst : SUBST =
  struct
```

```
    type t = path Ident.tbl
    let identity = Ident.emptytbl
    let add = Ident.add
    let rec path p sub =
      match p with
        Pident id -> (try Ident.find id sub with Not_found -> p)
      | Pdot(root, field) -> Pdot(path root sub, field)
  end
```

2.4 Abstract syntax for the base language

The abstract syntax for the base language is provided as an implementation of the following signature:

```
module type CORE_SYNTAX =
  sig
    type term
    type val_type
    type def_type
    type kind
    val subst_valtype: val_type -> Subst.t -> val_type
    val subst_deftype: def_type -> Subst.t -> def_type
    val subst_kind: kind -> Subst.t -> kind
  end
```

The type `term` is the abstract syntax tree for definitions of value names: a value expression in a functional language, a variable declaration or procedure definition in a procedural language, or a set of clauses defining a predicate in a logic language. The type `val_type` represents type expressions for these terms; `def_type` represents the type expressions that can be bound to a type name. In many languages, `val_type` and `def_type` are identical, but ML, for instance, has type schemes for `val_type`, but type constructors (type expressions possibly parameterized by other types) for `def_type`. Finally, the type `kind` describes the various kinds that a `def_type` may have. Many languages have only one kind of definable types; in ML, the kind of a `def_type` is the arity of a type constructor.

2.5 Abstract syntax for the module language

Given the syntax for a core language (a module of signature `CORE_SYNTAX`), we build the abstract syntax structure for the module language specified below. The core language syntax is re-exported as a substructure `Core` of the module language syntax, in order to record the core language on top of which the module language is built; the remainder of the signature refers to the core language a.s.t. types as components of the `Core` substructure.

```
module type MOD_SYNTAX =
  sig
    module Core: CORE_SYNTAX            (* the core syntax we started with *)
```

```
  type type_decl =
    { kind: Core.kind;
      manifest: Core.def_type option } (* either abstract or manifest *)
  type mod_type =
      Signature of signature                            (* sig ... end *)
    | Functor_type of Ident.t * mod_type * mod_type     (* functor(X: mty) mty *)
  and signature = specification list
  and specification =
      Value_sig of Ident.t * Core.val_type              (* val x: ty *)
    | Type_sig of Ident.t * type_decl                   (* type t [= ty] *)
    | Module_sig of Ident.t * mod_type                  (* module X: mty *)
  type mod_term =
      Longident of path                                 (* X or X.Y.Z *)
    | Structure of structure                            (* struct ... end *)
    | Functor of Ident.t * mod_type * mod_term          (* functor (X: mty) mod *)
    | Apply of mod_term * mod_term                      (* mod1(mod2) *)
    | Constraint of mod_term * mod_type                 (* (mod : mty) *)
  and structure = definition list
  and definition =
      Value_str of Ident.t * Core.term                  (* let x = expr *)
    | Type_str of Ident.t * Core.kind * Core.def_type   (* type t = ty *)
    | Module_str of Ident.t * mod_term                  (* module X = mod *)
  val subst_typedecl: type_decl -> Subst.t -> type_decl
  val subst_modtype: mod_type -> Subst.t -> mod_type
end
```

Module terms (type `mod_term`) denote either structures or functors. Structures are sequences of definitions: of a value identifier equal to a core term, of a type identifier equal to a definable core type, or of a (sub-)module identifier equal to a module term. Functors are parameterized module terms, i.e. functions from module terms to module terms; a module type is explicitly given for the parameter. Other module terms are module identifiers and access paths (`Longident`), referring to module terms bound elsewhere; applications of a functor to a module (`Apply`); and restriction of a module term by a module type (`Constraint`).

Module types are either signatures or functor types. Functor types are dependent function types: they consist of a module type for the argument, a module type for the result, and a name for the argument, which may appear in the result type. A signature describes the interface of a structure, as a sequence of type specifications for identifiers bound in the structure. Value specifications are of the form "this value identifier has that value type"; module specifications, "this module identifier has that module type". Type specifications consist of a kind and an optional definable type revealing the implementation of the type; the type identifier is said to be *manifest* if its implementation is shown in the specification, and *abstract* otherwise. Manifest types play an important role for recording type equalities, propagating them through functors, and express so-called sharing constraints between functor arguments [12]. Not all components of a structure need to be specified in a matching

signature: identifiers not mentioned in the signature are hidden and remain local to the structure.

The functor that takes an implementation of `CORE_SYNTAX` and returns the corresponding implementation of `MOD_SYNTAX` is trivial:

```
module Mod_syntax(Core_syntax: CORE_SYNTAX) =
  struct
    module Core = Core_syntax
    type type_decl = ...        (* as in the signature MOD_SYNTAX *)
    type mod_type = ...
    type mod_term = ...

    let subst_typedecl decl sub =
      { kind = Core.subst_kind decl.kind sub;
        manifest = match decl.manifest with
                     None -> None
                   | Some dty -> Some(Core.subst_deftype dty sub) }
    let rec subst_modtype mty sub =
      match mty with
        Signature sg -> Signature(List.map (subst_sig_item sub) sg)
      | Functor_type(id, mty1, mty2) ->
          Functor_type(id, subst_modtype mty1 sub, subst_modtype mty2 sub)
    and subst_sig_item sub = function
        Value_sig(id, vty) -> Value_sig(id, Core.subst_valtype vty sub)
      | Type_sig(id, decl) -> Type_sig(id, subst_typedecl decl sub)
      | Module_sig(id, mty) -> Module_sig(id, subst_modtype mty sub)
  end
```

The substitution functions are simple morphisms over declarations and module types, calling the substitution functions from `Core_syntax` to deal with core-language types and kinds. They assume that identifiers are bound at most once, so that name captures cannot occur.

2.6 The environment structure

Type-checking for the base language necessitates type information for module identifiers, in order to type module accesses such as `M.x`. Before specifying the base-language typechecker, we therefore need to develop an environment structure that records type information for value, type and module identifiers, and answers queries such as "what is the type of the value `M.x`?".

```
module type ENV =
  sig
    module Mod: MOD_SYNTAX
    type t
    val empty: t
    val add_value: Ident.t -> Mod.Core.val_type -> t -> t
```

```
    val add_type: Ident.t -> Mod.type_decl -> t -> t
    val add_module: Ident.t -> Mod.mod_type -> t -> t
    val add_spec: Mod.specification -> t -> t
    val add_signature: Mod.signature -> t -> t
    val find_value: path -> t -> Mod.Core.val_type
    val find_type: path -> t -> Mod.type_decl
    val find_module: path -> t -> Mod.mod_type
  end
```

Environments are handled in a purely applicative way, without side-effects: each `add` operation leaves the original environment unchanged and returns a fresh environment enriched with the given binding. `add_value` records the value type of a value identifier; `add_type`, the declaration of a type identifier; `add_module`, the module type of a module identifier. `add_spec` records one of the three kinds of bindings described by the given specification; `add_signature` records in turn all specifications of the given signature.

Below is a simple implementation of environments, parameterized by an A.S.T. structure for modules.

```
module Env(Mod_syntax: MOD_SYNTAX) =
  struct
    module Mod = Mod_syntax
    type binding =
        Value of Mod.Core.val_type
      | Type of Mod.type_decl
      | Module of Mod.mod_type
    type t = binding Ident.tbl
    let empty = Ident.emptytbl
```

For simplicity, all three kinds of bindings are stored in the same table. This does not preclude the language from having several name spaces (e.g. a type and a value can have the same name): names from different spaces are bound at different locations, and therefore receive distinct identifiers. The `add` functions are straightforward:

```
    let add_value id vty env = Ident.add id (Value vty) env
    let add_type id decl env = Ident.add id (Type decl) env
    let add_module id mty env = Ident.add id (Module mty) env
    let add_spec item env =
      match item with
        Mod.Value_sig(id, vty) -> add_value id vty env
      | Mod.Type_sig(id, decl) -> add_type id decl env
      | Mod.Module_sig(id, mty) -> add_module id mty env
    let add_signature = List.fold_right add_spec
```

The `find` functions returns the typing information associated with a path in an environment. If the input path is just an identifier, then a simple lookup in the environment suffices. If the path is a dot access, e.g. `M.x`, the signature of `M` is looked up in the environment, then scanned to find its `x` field and the associated type informations. Moreover, some substitutions are

required to preserve the dependencies between signature components. Assume for instance that the module `M` has the following signature:

```
M : sig type t  val x: t end
```

Then, the type of the value `M.x` is not `t` as indicated in the signature (that `t` becomes unbound once lifted out of the signature), but `M.t`. More generally, in the type of a component of a signature, all identifiers bound earlier in the signature must be prefixed by the path leading to the signature. This substitution can either be performed each time a path is looked up, or, more efficiently, be computed in advance when a module identifier with a signature type is introduced in the environment. Below is a naive implementation where the substitution is computed and applied at path lookup time.

```
let rec find path env =
  match path with
    Pident id ->
      Ident.find id env
  | Pdot(root, field) ->
      match find_module root env with
        Mod.Signature sg -> find_field root field Subst.identity sg
      | _ -> error "structure expected in dot access"
and find_field p field subst = function
    [] -> error "no such field in structure"
  | Mod.Value_sig(id, vty) :: rem ->
      if Ident.name id = field
      then Value(Mod.Core.subst_valtype vty subst)
      else find_field p field subst rem
  | Mod.Type_sig(id, decl) :: rem ->
      if Ident.name id = field
      then Type(Mod.subst_typedecl decl subst)
      else find_field p field (Subst.add id (Pdot(p, Ident.name id)) subst) rem
  | Mod.Module_sig(id, mty) :: rem ->
      if Ident.name id = field
      then Module(Mod.subst_modtype mty subst)
      else find_field p field (Subst.add id (Pdot(p, Ident.name id)) subst) rem
and find_value path env =
  match find path env with Value vty -> vty | _ -> error "value field expected"
and find_type path env =
  match find path env with Type decl -> decl | _ -> error "type field expected"
and find_module path env =
  match find path env with Module mty -> mty | _ -> error "module field expected"
end
```

As the reader may have noticed, error handling is extremely simplified in this paper: we assume given an `error` function that prints a message and aborts. Similarly, `Not_found` exceptions raised by `Ident.find` are not handled. A better implementation would use exceptions to gather more context before printing the error.

2.7 Type-checking the base language

The type-checker for the base language must implement the following signature:

```
module type CORE_TYPING =
  sig
    module Core: CORE_SYNTAX
    module Env: ENV with module Mod.Core = Core
(* Typing functions *)
    val type_term: Env.t -> Core.term -> Core.val_type
    val kind_deftype: Env.t -> Core.def_type -> Core.kind
    val check_valtype: Env.t -> Core.val_type -> unit
    val check_kind: Env.t -> Core.kind -> unit
(* Type matching functions *)
    val valtype_match: Env.t -> Core.val_type -> Core.val_type -> bool
    val deftype_equiv: Env.t -> Core.def_type -> Core.def_type -> bool
    val kind_match: Env.t -> Core.kind -> Core.kind -> bool
    val deftype_of_path: path -> Core.kind -> Core.def_type
(* Elimination of dependencies *)
    val nondep_valtype: Env.t -> Ident.t -> Core.val_type -> Core.val_type
    val nondep_deftype: Env.t -> Ident.t -> Core.def_type -> Core.def_type
    val nondep_kind: Env.t -> Ident.t -> Core.kind -> Core.kind
  end
```

The `Core` and `Env` components record the a.s.t. types and the environment structure over which the type-checker is built. Of course, the environment structure must be compatible with the a.s.t. structure: in SML parlance, some of their type components must share. In our system, this is expressed by the notation `ENV with module Mod.Core = Core`, which is actually syntactic sugar for the following signature that enriches `ENV` with type equalities over its `Mod.Core` component:

```
    sig
      module Mod: sig
        module Core: sig
          type term = Core.term
          type val_type = Core.val_type
          type def_type = Core.def_type
          type kind = Core.kind
          (* remainder of CORE_SYNTAX unchanged *)
        end
        (* remainder of MOD_SYNTAX unchanged *)
      end
      (* remainder of ENV unchanged *)
    end
```

The main typing function is `type_term`, which takes a term and an environment, and returns the principal type of the term in that environment (principal w.r.t. the `valtype_match`

ordering on value types). Depending on the base language, this function implements type inference (propagate types from the declarations of variables and function parameters) or ML-style type reconstruction (guess the types of function parameters as well). For simplicity, all typing functions are assumed to print a message and abort on error.

Three auxiliary functions `kind_deftype`, `check_valtype` and `check_kind` check the well-formedness of type and kind expressions in an environment, in particular that all type paths are bound and all kind constraints are met. In addition, `kind_deftype` infers and returns the kind of the given definable type.

The three predicates `valtype_match`, `deftype_equiv` and `kind_match` are used when checking an implementation against a specification, e.g. a structure against a signature. In a language with subtyping, `valtype_match` $e\ t_1\ t_2$ checks that the type t_1 is a subtype of t_2 in the environment e; in a language with ML-style polymorphism, that t_1 is a type schema more general than t_2; in a language with coercions, that t_1 can be coerced into t_2. Similarly, `deftype_equiv` $e\ t_1\ t_2$ checks that the definable types t_1 and t_2 are equivalent (identical modulo the type equalities induced by manifest type specifications contained in e).

The functions `nondep_valtype`, `nondep_deftype` and `nondep_kind` are used to eliminate all occurrences of a given identifier in the given type or kind (see section 2.11).

Finally, `deftype_of_path` transforms a type path and its kind into the corresponding definable type. For instance, in the case of ML, given the path `t` and the arity 2, it returns the parameterized type `('a, 'b)` $\mapsto$ `('a, 'b) t`.

2.8 Type-checking the module language

The type-checker for the module language has the following interface:

```
module type MOD_TYPING =
  sig
    module Mod: MOD_SYNTAX
    module Env: ENV with module Mod = Mod
    val type_module: Env.t -> Mod.mod_term -> Mod.mod_type
    val type_definition: Env.t -> Mod.definition -> Mod.specification
  end
```

The main entry point is `type_module`, which infers and returns the type of a module term. The intended usage for a separate compiler is to parse a whole implementation file as a module term, then pass it to `type_module`. If an interface file is also given, `type_module` should be applied to the constrained term $(m : M)$, where m is the implementation (a module term) and M the interface (a module type). The alternate entry point `type_definition` is intended for interactive use: the toplevel loop reads a definition, infers its specification, and prints the outcome.

The implementation of the type-checker is parameterized by an A.S.T. structure, an environment structure, and a type-checker for the core language, all three operating on compatible types:

```
module Mod_typing
    (TheMod: MOD_SYNTAX)
    (TheEnv: ENV with module Mod = TheMod)
    (CT: CORE_TYPING with module Core = TheMod.Core and module Env = TheEnv) =
  struct
    module Mod = TheMod
    module Env = TheEnv
    open Mod          (* Allows to omit the 'Mod.' prefix -- saves on typing *)
    let rec modtype_match env mty1 mty2 = ...   (* see section 2.9 *)
    let rec strengthen_modtype path mty = ...   (* see section 2.10 *)
    let nondep_modtype env id mty = ...         (* see section 2.11 *)
```

We postpone the definition of the three auxiliary functions above to the following sections. The `check_modtype` function below checks the well-formedness of a user-supplied module type — in particular, that no identifier is used before being bound.

```
    let rec check_modtype env = function
        Signature sg -> check_signature env sg
      | Functor_type(param, arg, res) ->
          check_modtype env arg; check_modtype (Env.add_module param arg env) res
    and check_signature env = function
        [] -> ()
      | Value_sig(id, vty) :: rem ->
          CT.check_valtype env vty; check_signature env rem
      | Type_sig(id, decl) :: rem ->
          CT.check_kind env decl.kind;
          begin match decl.manifest with
              None -> ()
            | Some typ -> CT.kind_deftype env typ; ()
          end;
          check_signature (Env.add_type id decl env) rem
      | Module_sig(id, mty) :: rem ->
          check_modtype env mty; check_signature (Env.add_module id mty env) rem
```

After checking a type specification or module specification in a signature, we add it to the environment before checking the remainder of the signature, since subsequent signature elements may refer to the type or module just checked. No such dependency occurs for value specifications. Similarly, the result type of a functor may depend on its parameter (the type of the `Mod_typing` functor itself is an example).

```
    let rec type_module env = function
        Longident path ->
          strengthen_modtype path (Env.find_module path env)
      | Structure str ->
          Signature(type_structure env str)
      | Functor(param, mty, body) ->
          check_modtype env mty;
```

```
        Functor_type(param, mty, type_module (Env.add_module param mty env) body)
    | Apply(funct, arg) ->
        (match type_module env funct with
          Functor_type(param, mty_param, mty_res) ->
            let mty_arg = type_module env arg in
            modtype_match env mty_arg mty_param;
            (match arg with
              Longident path ->
                subst_modtype mty_res (Subst.add param path Subst.identity)
            | _ ->
                try
                  nondep_modtype (Env.add_module param mty_arg env) param mty_res
                with Not_found ->
                  error "cannot eliminate dependency in functor application")
        | _ -> error "application of a non-functor")
    | Constraint(modl, mty) ->
        check_modtype env mty;
        modtype_match env (type_module env modl) mty;
        mty
  and type_structure env = function
      [] -> []
    | stritem :: rem ->
        let sigitem = type_definition env stritem in
        let sigrem = type_structure (Env.add_spec sigitem env) rem in
        sigitem :: sigrem
  and type_definition env = function
      Value_str(id, term) -> Value_sig(id, CT.type_term env term)
    | Module_str(id, modl) -> Module_sig(id, type_module env modl)
    | Type_str(id, kind, typ) ->
        if CT.kind_match env (CT.kind_deftype env typ) kind
        then Type_sig(id, {kind = kind; manifest = Some typ})
        else error "kind mismatch in type definition"
end
```

A reference to a module identifier or module component of a structure (`Longident`) is typed by a lookup in the environment, followed by a "strengthening" operation (`strengthen_modtype`) that turns abstract type specifications into specifications of types manifestly equal to themselves. Strengthening ensures that the identities of abstract types are preserved; this is detailed in section 2.10.

In the case of a structure, each definition is typed, then entered in the environment before typing the remainder of the structure, which can depend on the definition. Type definitions are assigned manifest signatures, which reveal their implementations; the type can be abstracted later, if desired, using a module constraint.

The typing of functor definitions is straightforward. For functor applications, we type the functor and its argument, then check that the type of the argument matches the type of the functor parameter. That is, the argument must provide at least all the components required

by the functor, with types at least as general. Matching between module types is detailed in section 2.9. Determining the result type of the application raises a subtle difficulty: since functor types are dependent, the result type of the functor can refer to the parameter name; according to the standard elimination rule for dependent function types, the parameter name must therefore be replaced by the actual argument to obtain the type of the application. If the actual argument is a path, this causes no difficulties, because we can always substitute a path for a module identifier anywhere in the module language. But if the argument is not a path, then the substitution is not always possible. Consider:

```
module F = functor(X: sig type t end) struct type t = X.t end
module A = F(struct type t = int end)
```

The result type of `F` is `sig type t = X.t end`, and attempting to replace `X` by `struct type t = int end` in this type creates an ill-formed module access `(struct type t = int end).t`. (Recall that accesses to structure components are restricted to module paths; lifting this restriction would compromise the type abstraction properties of the module system [13].) However, not all functor applications to non-paths run into this problem: everything works fine if the functor type is non-dependent; more subtly, there are also cases where the dependency can be eliminated using manifest type information from the type of the argument. The function `nondep_modtype`, explained below in section 2.11, attempts to eliminate the dependency in the functor result type; only if it fails is an error generated.

2.9 Matching between module types

A module type M matches a module type N if any module m satisfying the specification M also satisfies N. This allows several degrees of flexibility. If M and N are signatures, then M may specify more components than N; components common to both signatures may be specified more tightly in M than in N (e.g. N specifies a type `t` abstract and M manifest). If M and N are functor types, then M's result type can be more precise than N's, or M's argument type can be less precise (accepting more arguments) than N's. All in all, module type matching resembles subtyping in a functional language with records, with some extra complications due to the dependencies in functor types and signatures.

```
let rec modtype_match env mty1 mty2 =
  match (mty1, mty2) with
    (Signature sig1, Signature sig2) ->
      let (paired_components, subst) = pair_signature_components sig1 sig2 in
      let ext_env = Env.add_signature sig1 env in
      List.iter (specification_match ext_env subst) paired_components
  | (Functor_type(param1, arg1, res1), Functor_type(param2, arg2, res2)) ->
      let subst = Subst.add param1 (Pident param2) Subst.identity in
      let res1' = Mod.subst_modtype res1 subst in
      modtype_match env arg2 arg1;
      modtype_match (Env.add_module param2 arg2 env) res1' res2
  | (_, _) ->
      error "module type mismatch"
```

As outlined above, matching between functor types is contravariant in the argument types. Since the result types may depend on the parameters, we need to identify the two parameter identifiers. For matching the result types, we assign the parameter the more precise of the two argument types, allowing more type equalities to be derived about components of the parameter.

Matching between signatures proceed in several steps. First, the signature components are paired: to each component of `sig2`, we associate the component of `sig1` with same name and class. This pass also builds a substitution that equates the identifiers of the paired components, so that these identifiers are considered equal when matching specifications of components that depend on these identifiers.

```
and pair_signature_components sig1 sig2 =
  match sig2 with
    [] -> ([], Subst.identity)
  | item2 :: rem2 ->
      let rec find_matching_component = function
          [] -> error "unmatched signature component"
        | item1 :: rem1 ->
            match (item1, item2) with
              (Value_sig(id1, _), Value_sig(id2, _))
              when Ident.name id1 = Ident.name id2 -> (id1, id2, item1)
            | (Type_sig(id1, _), Type_sig(id2, _))
              when Ident.name id1 = Ident.name id2 -> (id1, id2, item1)
            | (Module_sig(id1, _), Module_sig(id2, _))
              when Ident.name id1 = Ident.name id2 -> (id1, id2, item1)
            | _ -> find_matching_component rem1 in
      let (id1, id2, item1) = find_matching_component sig1 in
      let (pairs, subst) = pair_signature_components sig1 rem2 in
      ((item1, item2) :: pairs, Subst.add id2 (Pident id1) subst)
```

After pairing, all components of the richer signature `sig1` are added to the typing environment; this allows matching of specifications to take advantage of all type equalities specified in `sig1`. Finally, the specifications of paired components are matched pairwise.

```
and specification_match env subst = function
    (Value_sig(_, vty1), Value_sig(_, vty2)) ->
      if not (CT.valtype_match env vty1 (Core.subst_valtype vty2 subst))
      then error "value components do not match"
  | (Type_sig(id, decl1), Type_sig(_, decl2)) ->
      if not (typedecl_match env id decl1 (Mod.subst_typedecl decl2 subst))
      then error "type components do not match"
  | (Module_sig(_, mty1), Module_sig(_, mty2)) ->
      modtype_match env mty1 (Mod.subst_modtype mty2 subst)
and typedecl_match env id decl1 decl2 =
  CT.kind_match env decl1.kind decl2.kind &
  begin match (decl1.manifest, decl2.manifest) with
```

```
  (_, None) -> true
| (Some typ1, Some typ2) -> CT.deftype_equiv env typ1 typ2
| (None, Some typ2) ->
    CT.deftype_equiv env (CT.deftype_of_path (Pident id) decl1.kind) typ2
end
```

Matching pairs of specifications is straightforward: value specifications match if their value types satisfy the `valtype_match` predicate provided by the core language type-checker. Module specifications match if their module types do. For type specifications, the kinds should obviously agree. No additional condition is required if the second type is specified abstract. If it is specified manifestly equal to some definable type d, then the first type must either be specified manifestly equal to a type equivalent to d, or specified abstract but provably equivalent to d in the current context.

The following ML example illustrates all cases of type specification matching:

M = `sig type 'a t  type u = int  type v = u  type w  type z = w end`
N = `sig type 'a t  type v = int  type z  type w = z end`

The two `t` specifications match because both are abstract with the same kind (arity 1). The `v=u` specification in M matches the `v=int` specification in N because `u` is equivalent to `int` in the environment enriched by M's components. The abstract type `z` in N is matched because `z` is manifest with the right kind (arity 0) in M. Finally, the `w=z` specification in N is matched by the `w` component of M, despite it being abstract, because `w` and `z` are equivalent in the enriched environment.

2.10 Strengthening of module types

Consider a module path p with a signature containing an abstract type `t`:

p : `sig type t ... end`

What makes p.`t` abstract is that, since the signature contains no type equality over `t`, p.`t` is incompatible with any other type except itself. However, the identity of p.`t` must be preserved, in particular across rebindings. Assume for instance that p is bound to a module identifier `m`:

`module m =` p

If we assign `m` the same signature as p, `sig type t ... end`, then `m.t` and p.`t` are different types. The identity of the abstract type p.`t` was lost. The correct signature for `m` that preserves p.`t`'s identity is:

`m : sig type t =` p`.t  ... end`

Fortunately, this signature is a perfectly legal signature for p itself: an abstract type `t` component of a path p is always manifestly equal to itself, p.`t`. The following function `strengthen_modtype` replaces all abstract type specifications in a module type by the corresponding manifest types rooted at the given path:

```
let rec strengthen_modtype path mty =
  match mty with
    Signature sg -> Signature(List.map (strengthen_spec path) sg)
  | Functor_type(_, _, _) -> mty
and strengthen_spec path item =
  match item with
    Value_sig(id, vty) -> item
  | Type_sig(id, decl) ->
      let m = match decl.manifest with
          None -> Some(CT.deftype_of_path (Pdot(path, Ident.name id)) decl.kind)
        | Some ty -> Some ty in
      Type_sig(id, {kind = decl.kind; manifest = m})
  | Module_sig(id, mty) ->
      Module_sig(id, strengthen_modtype (Pdot(path, Ident.name id)) mty)
```

In `type_module`, this strengthening operation is performed systematically on a module path each time it is referenced. It can be shown that this ensures inference of minimal module types [12] and implements the same notion of type generativity as in SML [15].

2.11 Elimination of dependencies in functor types

As mentioned in section 2.8, a difficulty arises when applying a functor with a truly dependent type to a module expression that is not a path. Continuing the example given above,

```
module F = functor(X: sig type t end) struct type t = X.t end
module A = F(struct type t = int end)
```

we cannot just replace `X` by the actual argument in the signature of the functor result (`sig type t = X.t end`). However, we can take advantage of the fact that the `t` component of the actual argument is known (from the signature of the argument) to be `int`: instead of replacing `X` by the argument, we will replace `X.t` by `int`, obtaining the correct signature `sig type t = int end` for `A`. The formal justification for this trick [9] is to notice that the functor type for `F`,

```
functor(X: sig type t end) sig type t = X.t end
```

is a subtype of

```
functor(X: sig type t = int end) sig type t = int end
```

therefore the former can be weakened to the latter just before typing the application. The latter type being non-dependent, no problem arises from the argument not being a path. Algorithmically, what we are doing is: given the result type `sig type t = X.t end` of the functor, find an equivalent type that does not depend on the parameter `X`, under the assumption that `X` has type `sig type t = int end`, the actual type of the argument. The functions `nondep_valtype`, `nondep_deftype` and `nondep_kind` provided by the core language type-checker perform the same operation on value types, definable types, and kinds.

```
let rec nondep_modtype env param = function
    Signature sg -> Signature(nondep_signature env param sg)
  | Functor_type(id, arg, res) ->
      Functor_type(id, nondep_modtype env param arg,
                       nondep_modtype (Env.add_module id arg env) param res)
and nondep_signature env param = function
    [] -> []
  | item :: rem ->
      let rem' = nondep_signature (Env.add_spec item env) param rem in
      match item with
        Value_sig(id, vty) ->
          Value_sig(id, CT.nondep_valtype env param vty) :: rem'
      | Type_sig(id, decl) ->
          let decl' =
            {kind = CT.nondep_kind env param decl.kind;
             manifest = match decl.manifest with
                          None -> None
                        | Some ty -> Some(CT.nondep_deftype env param ty)} in
          Type_sig(id, decl') :: rem'
      | Module_sig(id, mty) ->
          Module_sig(id, nondep_modtype env param mty) :: rem'
```

We assume that the `CT.nondep` functions raise the `Not_found` exception if they cannot eliminate the dependency on the parameter (for instance, we are trying to eliminate `X` in `X.t` under the assumption `X : sig type t end`). This exception goes through `nondep_modtype` transparently and is caught in `type_module`, causing an error to be reported.

One can go further in trying to eliminate dependencies: in covariant position, a manifest type specification `type t =` τ where the parameter cannot be eliminated from τ can be turned into an abstract type `type t`; a value specification `val x:` τ can be removed from a signature. This corresponds to taking a minimal non-dependent supertype of the functor result type, instead of a non-dependent equivalent type as in the `nondep_modtype` function above. By taking non-dependent supertypes, more programs are well-typed, but the inferred types are often confusing, because information is lost without warning when moving to a supertype. The approach followed in this paper (signal an error if some information would be lost but eliminate silently the dependency otherwise) seems preferable in practice.

3 Applications

This section outlines two applications of the generic module system presented above to two simplified base languages: core C and mini-ML.

3.1 Core C

The first base language considered is a small subset of the C language, hopefully representative of many conventional imperative languages. The abstract syntax is:

```
module C =
  struct
    type ctype =
        Void | Int | Float | Pointer of ctype | Function of ctype list * ctype
      | Typename of path
    type expr =
        Intconst of int | Floatconst of float   (* constants *)
      | Variable of path                        (* var or mod.mod...var *)
      | Apply of expr * expr list               (* function call *)
      | Assign of expr * expr                   (* var = expr *)
      | Unary_op of string * expr               (* *expr, !expr, etc *)
      | Binary_op of string * expr * expr       (* expr + expr, etc *)
      | Cast of expr * ctype                    (* (type)expr *)
    type statement =
        Expr of expr                            (* expr; *)
      | If of expr * statement * statement      (* if (cond) stmt; else stmt; *)
      | For of expr * expr * expr * statement   (* for (init; cond; step) stmt; *)
      | Return of expr                          (* return expr; *)
      | Block of (Ident.t * ctype) list * statement list (* { decls; stmts; } *)
    type term =
        Var_decl of ctype
      | Fun_def of (Ident.t * ctype) list * ctype * statement
    type val_type = ctype
    type def_type = ctype
    type kind = unit
    (* Substitution functions omitted *)
  end
```

Type expressions are quite simple: there is no distinction between value types and definable types, and there is only one kind of definable types. Applying the `Mod_syntax` and `Env` functors to `C` produces an environment structure suitable for writing the core-C typechecker:

```
module CMod = Mod_syntax(C)
module CEnv = Env(CMod)

module CTyping =
  struct
    module Core = C
    module Env = CEnv
    open CMod
    open C
    let rec check_valtype env = function
```

```
      Typename path -> CEnv.find_type path env; ()
    | Pointer ty -> check_valtype env ty
    | Function(args, res) -> List.iter (check_valtype env) args; check_valtype env res
    | _ -> ()
let kind_deftype = check_valtype
let check_kind env k = ()
let deftype_of_path path kind = Typename path
```

Type matching takes into account the fact that integers can be coerced into floats and vice-versa. Function types are invariant in their argument types.

```
let rec type_match env flexible ty1 ty2 =
  match (ty1, ty2) with
    (Void, Void) -> true
  | (Int, Int) -> true
  | (Float, Float) -> true
  | (Int, Float) -> flexible
  | (Float, Int) -> flexible
  | (Pointer t1, Pointer t2) -> type_match env flexible t1 t2
  | (Function(args1, res1), Function(args2, res2)) ->
      List.length args1 = List.length args2 &
      List.for_all2 (type_match env false) args1 args2 &
      type_match env flexible res1 res2
  | (Typename path1, Typename path2) ->
      path_equal path1 path2 or
      begin match (CEnv.find_type path1 env, CEnv.find_type path2 env) with
        ({manifest = Some def}, _) -> type_match env flexible def ty2
      | (_, {manifest = Some def}) -> type_match env flexible ty1 def
      | ({manifest = None}, {manifest = None}) -> false
      end
  | (Typename path1, _) ->
      begin match CEnv.find_type path1 env with
        {manifest = Some def} -> type_match env flexible def ty2
      | {manifest = None} -> false
      end
  | (_, Typename path2) ->
      begin match CEnv.find_type path2 env with
        {manifest = Some def} -> type_match env flexible ty1 def
      | {manifest = None} -> false
      end
  | (_, _) -> false
let deftype_equiv env ty1 ty2 = type_match env false ty1 ty2
let valtype_match env ty1 ty2 = type_match env true ty1 ty2
let kind_match env k1 k2 = true
```

Each time a type path is encountered that does not match trivially the other type, we look it up in the environment and resume matching with its definition if it is manifest; if it is abstract, then by definition it is not compatible with the other type and we return `false`.

```
    let rec type_expr env expr = ...            (* omitted; straightforward *)
    let rec check_statement env stmt = ...      (* omitted; straightforward *)
    let type_term env = function
        Var_decl ty ->
          check_valtype env ty; ty
      | Fun_def(params, ty_res, body) ->
          check_valtype env ty_res;
          check_statement (add_variables env params) ty_res body;
          Function(List.map snd params, ty_res)
```

Finally, removing the dependency of a type expression on an identifier is performed by expanding repeatedly type paths rooted at that identifier:

```
    let rec is_rooted_at id = function
        Pident id' -> Ident.equal id id'
      | Pdot(p, s) -> is_rooted_at id p
    let nondep_kind env id kind = ()
    let rec nondep_valtype env id = function
        Typename path ->
          if is_rooted_at id path then begin
            match CEnv.find_type path env with
              {manifest = None} -> raise Not_found (* cannot remove dependency *)
            | {manifest = Some ty} -> nondep_valtype env id ty
          end else
            Typename path
      | Pointer ty -> Pointer(nondep_valtype env id ty)
      | Function(args, res) ->
          Function (List.map (nondep_valtype env id) args, nondep_valtype env id res)
      | ty -> ty
    let nondep_deftype = nondep_valtype
  end
```

Voilà, the type-checker for a modular C:

```
module CModTyping = Mod_typing(CMod)(CEnv)(CTyping)
```

3.2 Mini ML

The application to ML as base language is not that different from the application to C. The main change is that value types and definable types are distinct in ML: value types are type schemes, while definable types are parameterized simple types. Definable types have a kind: their arities.

```
module ML =
  struct
    type term =
        Constant of int
```

```
      | Longident of path                        (* id or mod.mod...id *)
      | Function of Ident.t * term              (* function id -> expr *)
      | Apply of term * term                    (* expr(expr) *)
      | Let of Ident.t * term * term            (* let id = expr in expr *)
    type simple_type =
        Var of type_variable                    (* 'a, 'b *)
      | Typeconstr of path * simple_type list   (* typeconstr application *)
    and type_variable =
      { mutable repres: simple_type option;     (* representative, for union-find *)
        mutable level: int }                    (* binding level, for generalization *)
    type val_type =
      { quantif: type_variable list;            (* universally quantified variables *)
        body: simple_type }                     (* body of type scheme *)
    type def_type =
      { params: type_variable list;             (* list of parameters *)
        defbody: simple_type }                  (* body of type definition *)
    type kind = { arity: int }
    (* Substitution functions omitted *)
  end
module MLMod = Mod_syntax(ML)
module MLEnv = Env(MLMod)
```

For type reconstruction, we maintain incrementally the binding level of type variables, which allows generalization without scanning the typing environment for free type variables [24]. Scanning the type environment is costly, and moreover is not supported by the environment structure returned by the `Env` functor: we would have to use a custom environment structure, or manipulate a local environment (for `Function`- and `Let`-bound identifiers) in addition to the global environment (for module-level bindings).

```
module MLTyping = struct ... end
module MLModTyping = Mod_typing(MLMod)(MLEnv)(MLTyping)
```

We omit the implementation of the type-checking functions (module `MLTyping`), which is mostly standard [27, chapter 17]. Unification of two types whose type constructors are not equal paths looks up the paths in the environment and expands them if they are manifest types, very much like the `CTyping.type_match` does. `type_term` performs standard Hindley-Milner type reconstruction, then generalizes the type inferred. `kind_deftype` checks that the given parameterized type is closed and returns its arity. `valtype_match` is subsumption between type schemes, modulo expansion of manifest types as in unification. The `nondep_` functions proceed essentially as in `CTyping.nondep_type`.

4 Compilation

We have concentrated so far on the problem of type-checking the module language. We now sketch briefly its compilation, which is mostly standard and builds on the type information gathered during module typing [17].

Structures are naturally represented as records (tuples) of values and sub-structures, obtained by erasing all type fields. Access to structure fields is either by name (similar to a method lookup in an object) or, more efficiently, at fixed offsets determined at compile-time from the signature of the structure. In the latter case, constraining a structure to a less precise signature involves reconstructing the record to match the new signature (coercive subtyping). To this end, the `modtype_match` function should return a coercion term recording the matching operation (e.g. the mapping of components from the more precise signature to the less precise signature). These coercions introduce no run-time inefficiencies, since they occur only at link time or program initialization time, but never inside loops or recursive functions.

If the compiler supports first-class functions (closures), functors can be translated to functions from structure representations to structure representations and compiled only once. A functor that takes abstract type components in its argument becomes a polymorphic function; this imposes the same constraints on data representations as in polymorphically-typed languages [23, 11]. Alternatively, the functor body can be recompiled for each application, as is often done for generics in Ada or templates in C++. The fact that only a finite number of functor specializations need to be compiled is guaranteed by the "phase distinction" result [10]: the module language is strongly normalizing if core language terms are not reduced.

5 Extensions

5.1 Beyond values and types

We have assumed so far that the base language has only two classes of things that can be defined and put inside structures: values and types. Some languages need more classes of definitions: kind definitions in languages with a rich kind system [2]; propositions and possibly proofs in specification languages [26]; macro definitions in C and Lisp [6]. For these languages, the `Mod_syntax`, `Env` and `Mod_typing` functors need to be reworked: the extra classes of definitions should be added to the `definition` type, their type specifications to the `specification` type, `add` and `find` functions to the environment structure, and finally matching rules for the new classes of specifications to the `specification_match` function.

Other language features do not correspond to new classes of definitions, but simply to subdivisions of the general classes of values and types: in Pascal and Modula, values are subdivided into constants and variables; in ML, type definitions are either datatypes or type abbreviations, and values are either `let`-bound identifiers, datatype constructors, or exception constructors. In this situation, our generic module system need not be modified: it suffices to reflect the subdivision in the `val_type` and `def_type` types of the base language description, e.g.

```
type val_type = Variable of ... | Constant of ...
```

Finally, some type definitions may also define values at the same time: typically, a class definition in a typed object-oriented language defines both a type of objects and a set of

methods; in ML, a datatype definition or an exception definition introduces constructors that can be later used as values. This is easily handled in our framework by defining a custom environment structure whose `add_type` function records the associated value definitions in the value name space. The `Mod_syntax` and `Mod_typing` functors need not be changed. This illustrates the interest of parameterizing `Mod_typing` by the environment structure, instead of hard-wireing internally the use of the `Env` functor.

5.2 Generative type definitions

Throughout this work, we have compared types by structure, except for type paths specified abstractly, which are compared by name. This makes type definitions non generative; only type abstraction is generative — more precisely, the only operation that generates new types is constraining a structure by a signature specifying an abstract type. Some languages have type definitions that generate new types, yet do not abstract the concrete representations of the types:

- In C, `struct` types are compared by name, thus each `struct` definition generates a new type, yet the record fields can be accessed directly.
- In ML, datatype definitions also generate new types, compared by name rather than by structure during unification, yet the constructors allow direct construction and inspection of values of that type.
- The definitions `is new` t in Ada and `BRANDED REF` t in Modula-3 create a type different from t, but which can be coerced to and from t.

The correct way to treat these definitions in our framework is to record their structure (e.g. list of record fields or datatype constructors, with their types) in the `kind` field of their definition, leaving the `manifest` field equal to `None`. This way, the types are compared by name (no type equalities are known for them), but their structure is remembered and can be consulted to check a record access or a type coercion, or to record the datatype constructors as values. For instance, in the case of ML, kinds record not only the arity of the type constructor, but also whether it comes with associated constructors:

```
type kind = { arity: int; description: type_description }
and type_description = Plain | Datatype of constructor list
and constructor = ...
```

Since having associated constructors and being manifestly equal to another type are independent properties in this approach, a type specification can combine both, as in

```
module M = struct ... type t = A | B ... end
module N = (M: sig type t = M.t = A | B end)
```

This is useful to re-exports the type `M.t` along with its constructors `A` and `B`, while keeping the compatibility between `M` and `N`. Writing `(M: sig type t = M.t end)` would preserve the type compatibility but hide the constructors, while `(M: sig type t = A | B end)` would leave the constructors apparent, but make a new type `t` incompatible with `M.t`.

5.3 Manifest constants, inline functions, and macros

In a context of separate compilation, the interface of a module is supposed to provide all the information needed to compile clients of this module. Some base-language features complicate this goal. For instance, if a module exports a macro definition, then the actual definition of this macro (and not just a guarantee of its existence) is needed to compile client modules. If a module defines a value as a constant, compilers could generate better code for the clients if they knew the actual value of the constant and not just its type. Similarly, if a function is defined as expandable (inline), then its actual definition must be available to the clients for inline expansion to take place.

There are two ways to address this problem. One is to enrich the language of module signatures to allow "manifest values", analogous to manifest types: the signature specifies not only the type of the value, but also its actual definition. For instance, the following signature

```
module M :
  sig
    val c = 10
    val f : int -> int = fun x -> x+1
  end
```

allows in-line expansion of the function `f` and of the constant `c` in all users of `M`. This approach raises several technical issues. First, signature matching requires a suitable notion of equivalence between manifest values. Equivalence is straightforward between constants, but not between in-line functions or macro definitions; some decidable approximation must be agreed upon. Second, checking the well-formedness of signatures requires that the manifest values are well-typed in the context of the signature. This prevents exporting in-line functions that refer to non-exported functions or variables in the same structure, or that take advantage of the particular implementation of a type exported abstractly.

One may object that function inlining and constant propagation are purely compiler issues and should not pollute the module system. From this alternate viewpoint, manifest values have nothing to do in the interface of a module, viewed as its type specification; they are just additional information for cross-module optimizations. This information should be recorded and propagated separately by the compiler, possibly in persistent storage to support separate compilation. This alternate approach is especially adequate if the extra information affects only the efficiency of the generated code, but not its semantics: if inlining information for an external function is not available at the time this function is used, a standard function call can always be generated. On the other hand, this approach is probably inadequate for macros and other syntactic extensions, whose definition must be available at the time they are used. The solution adopted in [6, 19] is to compile syntactic extensions separately, before compiling the remainder of the code.

5.4 Applicative functors

An interesting extension of the module calculus is to allow simple functor applications in paths, e.g. `F(A).t` where `F` is a functor identifier and `A` a structure identifier is a valid type expression [13]. In particular, this extension enhances the expressive power of higher-order functors (functors taking functors as arguments), making them "fully transparent" in the terminology of [18]. A complete discussion of full transparency and applicative functors is beyond the scope of this paper; see [13]. Here, we will only discuss their impact on the generic module implementation.

Allowing functor applications in paths raises a difficulty in the implementation of the `Env` environment structure. Recall that the environment structure should answer queries such as "what is the type of this path?". It does so by looking up the bindings of identifiers in the current environment (if the path is an identifier), possibly followed by accesses to signature fields (if the path is a projection `M.x`). If the path can also be a functor application `F(A)`, the environment structure must also check the type-correctness of the application of `F` to `A`, before deriving the type of `F(A)` from the result type of `F`. Type-checking a functor application requires matching a module type against another — as per the `modtype_match` function in the `Mod_typing` functor (see section 2.9). Unfortunately, the `modtype_match` function assumes given an already-built environment structure.

Applicative functors therefore introduce a nasty mutual recursion between the `Env` and `Mod_typing` functors; either needs to be parameterized by the result of applying the other, as in the following pseudo-code:

```
module rec MLEnv = Env(MLMod)(MLModTyping)
       and MLTyping = struct ... end
       and MLModTyping = Mod_typing(MLMod)(MLEnv)(MLTyping)
```

Unfortunately, modules defined by mutual recursion are not supported in SML nor in the module language presented here, mostly because they raise serious compilation problems. (The typing issues are not completely sorted out either, but see [7] for a proposal.) The usual trick for reducing mutual recursion to simple recursion at the level of values (parameterize all functions in `Env` and `MLTyping` by the `modtype_match` function) does not work very well here, as it pollutes the base-language implementation with module-level operations. The Caml Special Light implementation uses a reference to a dummy matching function in the environment structure; this reference is updated later by the correct `modtype_match` function.

6 Conclusions

We have presented a reference implementation of a module system with functors and multiple views of modules, and demonstrated its versatility and independence with respect to the base language. The requirements put on the base language are fairly weak, and many existing languages — not just typed λ-calculi — appear to fit in the framework presented here.

Just as type theory in general, our module system is biased towards structural equivalence between types, but generative type definitions can also be handled with little extra effort. Again just as type theory. it is largely independent of the evaluation paradigm [2]: we have used imperative and functional languages as examples, but there is no reason why logic, reactive, or dataflow languages could not be accommodated, once equipped with a type system. Object-oriented languages raise a subtle issue: the object-oriented features related to evaluation only (e.g. method invocation) are orthogonal to the module system, but most object-oriented languages also provide code structuring features (classes, inheritance, ...) that overlap with a module system. An interesting direction for future work is to understand and reduce this overlap by unifying classes with structures and inheritance with some forms of functors. Another interesting direction is to see if the module system applies as well to proof checkers, an area where the need is growing for decomposing large proofs in smaller units.

References

[1] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

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

[3] Luca Cardelli. The Quest implementation. Software and documentation available on `ftp://gatekeeper.dec.com/pub/DEC/Quest`, 1990.

[4] Luca Cardelli. Program fragments, linking, and modularization. Unpublished draft, 1993.

[5] Pierre Crégut and David B. MacQueen. An implementation of higher-order functors. In *Proc. 1994 Workshop on ML and its applications*, pages 13–21. Research report 2265, INRIA, 1994.

[6] P. Curtis and J. Rauen. A module system for Scheme. In *Lisp and Functional Programming 1990*, pages 13–19. ACM Press, 1990.

[7] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *International Conference on Functional Programming 96*. ACM Press, 1996. To appear.

[8] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag, 1993.

[9] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[10] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th symposium Principles of Programming Languages*, pages 341–354. ACM Press, 1990.

[11] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th symposium Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

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

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

[14] Xavier Leroy. The Caml Special Light system. Software and documentation available on the Web, `http://pauillac.inria.fr/csl/`, 1995.

[15] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 1996. To appear. An extended abstract appeared in *Proc. 1994 Workshop on ML and its applications*, research report 2265, INRIA, pages 1–12.

[16] David B. MacQueen. Modules for Standard ML. In Robert Harper, David B. MacQueen, and Robin Milner, editors, *Standard ML*. University of Edinburgh, technical report ECS LFCS 86-2, 1986.

[17] David B. MacQueen. The implementation of Standard ML modules. In *Lisp and Functional Programming 1988*, pages 212–223. ACM Press, 1988.

[18] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In D. Sannella, editor, *Programming languages and systems – ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

[19] Michel Mauny and Daniel de Rauglaudre. A complete and realistic implementation of quotations in ML. In *Proc. 1994 Workshop on ML and its applications*, pages 70–78. Research report 2265, INRIA, 1994.

[20] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.

[21] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[22] Simon L. Peyton-Jones. *The implementation of functional programming languages*. Prentice-Hall, 1987.

[23] Simon L. Peyton-Jones and John Launchbury. Unboxed values as first-class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, 1991.

[24] Didier Rémy. Extending ML type system with a sorted equational theory. Research report 1766, INRIA, 1992.

[25] D. T. Sannella and L. A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, 12:147–177, 1992.

[26] Donald Sannella and Andrzej Tarlecki. Extended ML: past, present and future. Technical report ECS-LFCS-91-138, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[27] Pierre Weis and Xavier Leroy. *Le langage Caml*. InterÉditions, 1993.

[28] Martin Wirsing. Algebraic specifications. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, pages 675–788. The MIT Press/Elsevier, 1990.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399