A modular module system

XAVIER LEROY
INRIA Rocquencourt
B.P. 105, 78153 Le Chesnay, France
`Xavier.Leroy@inria.fr`

Abstract

A simple implementation of an SML-like module system is presented as a module parameterized by a base language and its type-checker. This implementation is useful both as a detailed tutorial on the Harper-Lillibridge-Leroy module system and its implementation, and as a constructive demonstration of the applicability of that module system to a wide range of programming languages.

1 Introduction

Modular programming can be done in any language, with sufficient discipline from the programmers (Parnas, 1972). 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 ML, classes in C++ and Java, ...

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 (MacQueen, 1986; Milner *et al.*, 1997), also used in Objective Caml (Leroy *et al.*, 1996). 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 attempts at transferring it to other languages are adaptations to Prolog (Sannella & Wallen, 1992) and to Signal (Nowak *et al.*, 1997) that did not receive much publicity. What if one wants SML-style modules in one's favorite language? Say, Fortran?

Recent work on the type-theoretic foundations of SML modules (Harper & Lillibridge, 1994; Leroy, 1994) 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, (Leroy, 1994) 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 twofold. The first purpose is to give a highly constructive proof of that claim: we present an implementation of a ML-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.

The second purpose of this paper is to give a tutorial introduction to the Harper-Lillibridge-Leroy presentation of the ML module system and to its implementation. To this end, most of the actual source code is shown, thus providing a reference implementation of the module system that complements 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 (Cardelli, 1987; Peyton-Jones, 1987; Weis & Leroy, 1999). For this purpose, the implementation presented in this paper has been made as simple as possible, but no simpler (to quote Einstein out of context).

The implementation presented in this paper is written in Objective Caml (Leroy *et al.*, 1996), an extension of the Caml dialect of ML (Weis & Leroy, 1999) with objects and a module system extremely close to the one that is described here. In the established tradition of meta-circular interpretors for Lisp, the code presented in this paper exemplifies the module language that it implements, in particular the systematic use of functors. We hope that, far from making this paper impenetrable to readers unfamiliar with the theory and practice of ML-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 (Wirsing, 1990). The algebraic approach is both stronger and weaker than the type-theoretic approach followed here: it supports equations, but not higher-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 (Guttag & Horning, 1993).

Cardelli (1998) 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. Other generic frameworks for linking and separate compilation with much the same characteristics as Cardelli's include (Flatt & Felleisen, 1998; Glew & Morrisett, 1999).

Mixins, originally introduced as a generalization of inheritance in object-oriented languages (Bracha, 1992), have been proposed as a generic module calculus by Ancona and Zucca (1998; 1999). Ancona and Zucca give algebraic and operational semantics for mixin modules that are largely independent of the underlying base language.

On the implementation side, the New Jersey ML implementation of the SML module system is described in (MacQueen, 1988) and its extension to higher-order functors in (Crégut & MacQueen, 1994). 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.

The New Jersey ML implementations follow the stamp-based static semantics for SML modules (Milner *et al.*, 1997; MacQueen & Tofte, 1994). This semantics is close to an actual implementation of a typechecker for the SML module system. In particular, the semantics represents the identities of types using stamps (unique names) just like actual SML implementations do. 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. Moreover, the static semantics is not completely algorithmic, in the sense that it allows both principal and non-principal typings, while an actual type-checker is expected to produce principal typings.

Another semantics for SML modules that is close to an actual implementation of a type-checker is that of Harper and Stone (1998). This semantics does not use stamps, but relies directly on a syntactic treatment of type abstraction similar to the one we use in this paper. However, the semantics does not lead directly to a type-checking algorithm for the same reasons as mentioned above in the case of (Milner *et al.*, 1997).

Cardelli's implementation of Quest (Cardelli, 1990) 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 can concentrate on subsections 2.4 and 2.7 only. 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. For reference, appendix A shows the typing rules for the module system implemented in this paper.

2 The modular module system

2.1 Identifiers

The first issue we have to solve is the status of names (of types, variables, and modules). In our module system, type and module names play an important role in deciding type compatibility (we will use name equivalence for abstract types). This requires different types to have different names, otherwise the soundness of the type system is compromised.

Some languages allow type names to be redefined arbitrarily; others prevent redefinition within the same block, but allow a declaration in an inner block to shadow a declaration with the same name in an outer enclosing block. In both cases, typing difficulties arise: 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`. To avoid these difficulties, typed calculi generally rely on renaming (α-conversion) of type names to ensure uniqueness of names within a typing context.

However, these renamings conflict with another feature of ML-like module systems: clients of modules refer to their components by name (e.g. `M.t` to refer to the `t` type component of module `M`). This implies that names of module components are fixed and must not be renamed lest external references become invalid.

To solve this dilemma, we introduce a notion of identifiers distinct from names: each identifier has a name, but it also records the binding location of this name. Thus, we can have different type identifiers, bound at different locations, that have the same external name (Cardelli, 1990; Harper & Lillibridge, 1994; Leroy, 1994). Names in the program source are replaced by identifiers in the abstract syntax tree in accordance with the static scoping rules of the language. This can be performed either during parsing or as a separate “scoping” pass prior to type-checking. The abstract type of identifiers has the following signature:

```
module type IDENT =
  sig
    type t
    val create: 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
```

`create` 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 `create` operation, and `'a tbl` as association lists (any dictionary data structure could be used instead).

```
module Ident : IDENT =
  struct
    type t = {name: string; stamp: int}
    let currstamp = ref 0
    let create 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`. As mentioned in section 2.1, access to a module component is by name: the second argument of `Pdot` is a string, not an identifier; it would not make sense to put a full identifier there, since the access is generally not in the scope of the identifier binding. To avoid ambiguity, we require that all components of a module (at the same nesting level of modules) have distinct names. The same constraint is enforced on the signatures assigned to those modules. For instance, a module `M` cannot have two type components named `t`, because we would not know which type `M.t` refers to. However, sub-modules can still define components with the same name as components from an enclosing module, since the path notation distinguishes them. For instance, `M` can have a `t` type component and a `N` sub-module with another `t` type component; the former type `t` is referred to as `M.t`, and the latter as `M.N.t`.

Path equality naturally extends 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
```

`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 }          (* 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 :: k [= 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 :: k = 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 (Leroy, 1994). 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 is adequate for source languages that have a unique name space (e.g. a type and a value cannot have the same name); to handler multiple name spaces, separate tables can be used. 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 information. 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.kind -> 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
  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 equivalent to 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, `kind_match` e k_1 k_2 checks that the kind k_1 is a subkind of k_2 in the environment e. For most base languages, the kind structure is simple enough that `kind_match` reduces to kind equality. Finally, `deftype_equiv` e k t_1 t_2 checks that the definable types t_1 and t_2, viewed at kind k, are equivalent (identical modulo the type equalities induced by manifest type specifications contained in e). Again, for most base languages the extra kind argument k is not used, but with a rich enough kind system, the equivalence of definable types might depend on the kind with which they are considered.

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 `u` and the arity 0, it returns the type `u`; given the `t` and the arity 2, it returns the parameterized type `('a, 'b)` $\mapsto$ `('a, 'b) t`. This can be viewed as a form of eta-expansion on the type path.

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 *)
    let rec modtype_match env mty1 mty2 = ...         (* see section 2.9 *)
    let rec strengthen_modtype path mty = ...         (* see section 2.10 *)
```

We postpone the definition of the two 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 seen = function
        [] -> ()
      | Value_sig(id, vty) :: rem ->
          if List.mem (Ident.name id) seen
          then error "repeated value name";
          CT.check_valtype env vty;
          check_signature env (Ident.name id :: seen) rem
      | Type_sig(id, decl) :: rem ->
          if List.mem (Ident.name id) seen
          then error "repeated type name";
          CT.check_kind env decl.kind;
          begin match decl.manifest with
              None -> ()
            | Some typ ->
                if not (CT.kind_match env (CT.kind_deftype env typ)
                                          decl.kind)
                then error "kind mismatch in manifest type specification"
          end;
          check_signature (Env.add_type id decl env)
                          (Ident.name id :: seen) rem
      | Module_sig(id, mty) :: rem ->
          if List.mem (Ident.name id) seen
          then error "repeated module name";
          check_modtype env mty;
          check_signature (Env.add_module id mty env)
```

```
                    (Ident.name id :: seen) 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).

The extra parameter `seen` to `check_signature` is a list of component names already encountered; it is used to check that a given name does not appear twice in the signature.

```
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, (Longident path as 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;
          subst_modtype mty_res (Subst.add param path Subst.identity)
      | _ -> error "application of a non-functor")
  | Apply(funct, arg) ->
      error "application of a functor to a non-path"
  | Constraint(modl, mty) ->
      check_modtype env mty;
      modtype_match env (type_module env modl) mty;
      mty
and type_structure env seen = function
    [] -> []
  | stritem :: rem ->
      let (sigitem, seen') = type_definition env seen stritem in
      sigitem :: type_structure (Env.add_spec sigitem env) seen' rem
and type_definition env seen = function
    Value_str(id, term) ->
      if List.mem (Ident.name id) seen
      then error "repeated value name";
      (Value_sig(id, CT.type_term env term), Ident.name id :: seen)
  | Module_str(id, modl) ->
      if List.mem (Ident.name id) seen
      then error "repeated module name";
      (Module_sig(id, type_module env modl), Ident.name id :: seen)
  | Type_str(id, kind, typ) ->
      if List.mem (Ident.name id) seen
      then error "repeated type name";
      CT.check_kind env kind;
      if not (CT.kind_match env (CT.kind_deftype env typ) kind)
```

```
          then error "kind mismatch in type definition";
          (Type_sig(id, {kind = kind; manifest = Some typ}),
           Ident.name id :: seen)
  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 could compromise the type abstraction properties of the module system (Leroy, 1995; Courant, 1997a).) To avoid this difficulty, we simply reject all functor applications where the argument given to the functor is not a path. This requires users (or a preprocessor) to bind complex functor arguments to module names before applying the functors to the module names. In section 5.5, we shall return to this issue and propose less drastic restrictions.

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 proceeds 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 &&
  (match (decl1.manifest, decl2.manifest) with
    (_, None) -> true
  | (Some typ1, Some typ2) ->
     CT.deftype_equiv env decl2.kind typ1 typ2
  | (None, Some typ2) ->
      CT.deftype_equiv env decl2.kind
                 (CT.deftype_of_path (Pident id) decl1.kind) typ2)
```

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 and implements the same notion of type generativity as in SML (Leroy, 1996).

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                  (* integer constants *)
      | Floatconst of float              (* 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; see Web appendix *)
  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 -> ignore(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 reduces to type equivalence modulo the expansion of manifest types.

```
let rec valtype_match env ty1 ty2 =
  match (ty1, ty2) with
    (Void, Void) -> true
  | (Int, Int) -> true
  | (Float, Float) -> true
  | (Function(args1, res1), Function(args2, res2)) ->
      List.length args1 = List.length args2 &&
      List.for_all2 (valtype_match env) args1 args2 &&
      valtype_match env res1 res2
  | (Typename path1, Typename path2) ->
      path_equal path1 path2 ||
      begin match (CEnv.find_type path1 env,
                   CEnv.find_type path2 env) with
        ({manifest = Some def}, _) -> valtype_match env def ty2
      | (_, {manifest = Some def}) -> valtype_match env ty1 def
      | ({manifest = None}, {manifest = None}) -> false
      end
  | (Typename path1, _) ->
      begin match CEnv.find_type path1 env with
        {manifest = Some def} -> valtype_match env def ty2
      | {manifest = None} -> false
      end
  | (_, Typename path2) ->
      begin match CEnv.find_type path2 env with
        {manifest = Some def} -> valtype_match env ty1 def
      | {manifest = None} -> false
      end
  | (_, _) -> false
let deftype_equiv env kind t1 t2 = valtype_match env t1 t2
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; see Web appendix *)
let rec check_statement env type_function_result stmt =
  ...                         (* omitted; see Web appendix *)
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)
```

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. The kind of a definable type is an integer representing its arity (number of type parameters).

```
module ML =
  struct
    type term =
        Constant of int                       (* integer constants *)
      | Longident of path                     (* id or mod.mod...id *)
      | Function of Ident.t * term            (* fun 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  (* constructed type *)
    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;          (* 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 (Rémy, 1992; Weis & Leroy, 1999). 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)
```

The implementation of the type-checking functions (module `MLTyping`) is given in the Web appendix to this paper. We omit it here because it is mostly standard (Weis & Leroy, 1999). 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. `type_term` performs standard Hindley-Milner type reconstruction, then generalizes the type inferred and checks that the resulting type scheme is closed[1]. `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.

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 (MacQueen, 1988).

Structures are naturally represented as records (tuples) of values and substructures, obtained by erasing all type fields. Access to structure fields is either by name (similar to a method lookup in an Smalltalk 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 (Peyton-Jones & Launchbury, 1991; Leroy, 1992). This translation of functors into functions is relatively easy if the target language (the compiler's intermediate language) is untyped or weakly typed, but becomes much more difficult if the target language is strongly typed, like the typed intermediate language of (Tarditi *et al.*, 1996). Harper and Stone (1998) develop a type-preserving translation of functors into a typed intermediate language as part of their semantics for SML-97.

Alternatively, the functor body can be recompiled for each application, specializing the functor body for the actual argument of the application. This is how generics in Ada or templates in C++ are traditionally compiled. The fact that only a finite number of functor specializations need to be compiled is guaranteed by the

[1] The closedness check is not needed for pure ML, where all type variables free in the inferred type can always be generalized, but is required if generalization is restricted to syntactic values, as proposed in (Wright, 1995) to deal with the imperative features of full ML. Leaving non-generalized type variables free in the type schemes for value definitions, letting them be unified at points of use, raises delicate type soundness issues that are discussed in (Russo, 1998), sections 8.2 and 8.3.

"phase distinction" result (Harper *et al.*, 1990): the module language is strongly normalizing if core language terms are not reduced.

The static interpretation of SML modules proposed by Elsman (1999) goes one step further: not only functors are specialized at each application, but structures are also completely eliminated at compile-time, by replacing references to structure components by direct references to their definitions. Elsman also shows an incremental recompilation framework that avoids recompiling a functor specialization if neither the functor nor its argument have changed.

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 (Cardelli, 1989); classes and class types in Objective Caml (Leroy *et al.*, 1996); propositions and possibly proofs in specification languages (Sannella & Tarlecki, 1991); macro definitions in C and Lisp (Curtis & Rauen, 1990). 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 locally applying the `Env` functor inside the `Mod_typing` 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. For instance, 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. Finally, 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 = ...
```

This is exactly how many ML and Haskell type-checkers, as well as the SML definition (Milner *et al.*, 1997), represent datatypes during type-checking, although it is rarely, if ever, formulated explicitly in terms of kinds. 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-export the type `M.t` along with its constructors `A` and `B`, while keeping the compatibility between `M.t` and `N.t`. Writing `(M: sig type t = M.t end)` would preserve the type compatibility but fail to include the constructors `A` and `B` as components of `N`, while `(M: sig type t = A | B end)` would leave the constructors apparent in `N`, but make a new type `N.t` incompatible with `M.t`.

5.3 Manifest constants, inline functions, and macros

In the 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 : int = 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 (Curtis & Rauen, 1990; Mauny & de Rauglaudre, 1994) is to compile syntactic extensions separately, before compiling the remainder of the code.

5.4 Mutually recursive modules

Like the ML module system, the module system presented here requires that a module refers only to previously defined modules, thus preventing recursive or mutually recursive module definitions. Such recursive modules occur naturally when recursive definitions of types and function are spread across different modules. For instance, one cannot define structures for trees and forests as in the following pseudo-code:

```
module rec Tree =
  struct type 'a t = Leaf of 'a | Node of 'a Forest.t
         let size = function Leaf _ -> 1 | Node f -> Forest.size f
  end
and Forest =
  struct type 'a t = 'a Tree.t list
         let rec size = function [] -> 0
                               | t::l -> Tree.size t + size l
  end
```

Adding a `module rec` construct to our module system raises delicate typing and compilation issues; see (Crary *et al.*, 1999) for a discussion. Two different approaches to solving these issues have been proposed so far. Crary *et al.* (1999) rely on mutual recursion between signatures and on a special, "transparent" interpretation of ML datatypes during the type-checking of the mutual definitions. Duggan and Sourelis (1996) introduce mixin modules, which are structures containing deferred (not yet defined) components, and a special mixin composition operation to connect together the deferred and defined components of two structures. Mixin modules have been studied further by Ancona and Zucca (1998; 1999), and are also very close to Flatt and Felleisen's units (Flatt & Felleisen, 1998).

5.5 Functors applied to non-paths

The type-checker for the module language presented in section 2.8 rejects all functor applications $m_1(m_2)$ where m_2 is not syntactically a module path – in accordance with the typing rules of appendix A. The technical justification for this restriction is that our type algebra is not closed under substitution of arbitrary module expressions m_2 for module identifiers, but only under substitution of paths for identifiers. This restriction does not reduce the expressive power of the module language (as shown in (Leroy, 1996), a program can always be rewritten in a form where all functor arguments are paths), but still is a minor annoyance for programmers.

There are several common situations where the path restriction could be lifted without harm. First, if the functor m_1 has a non-dependent type $\mathtt{functor}(X : M_1)\ M_2$ where the formal parameter X does not occur in the result signature M_2, it is tempting to say that the application $m_1(m_2)$ has type M_2 regardless of whether m_2 is a path or not: the substitution of m_2 for X in M_2 always succeeds. More formally, we could try to replace the typing rule 5 for functor application by the following more lenient rule:

$$\frac{\begin{array}{c}E \vdash m_1 : \mathtt{functor}(X : M_1)\ M_2 \qquad E \vdash m_2 : M_2 \\ m_2 \text{ is a path or } X \text{ is not free in } M_2\end{array}}{E \vdash m_1(m_2) : M_2\{X \leftarrow m_2\}} \quad (5')$$

Perhaps surprisingly, this rule, combined with the subsumption rule 7, allows to type-check certain applications of functors with truly dependent types to arguments that are not paths. This was first noticed by Harper and Lillibridge (1994). Consider the following example, similar to that of section 2.8:

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

To type-check the application of `F`, we could first consider `F` with the type

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

which is a supertype of the "true" type of `F`, `functor(X: sig type t end) sig type t = X.t -> X.t end)`. The new type for `F` being non-dependent, rule 5′ applies and concludes that `A` has type `sig type t = int -> int end`. More intuitively, we took advantage of the fact that the `t` component of the actual argument is known (from the signature of the argument) to be `int`, and instead of replacing `X` by the argument in the result signature of the functor, we replaced `X.t` by `int`, obtaining the correct signature `sig type t = int -> int end` for `A`.

The problem with this approach is to choose the right non-dependent supertype of the functor type that permits the functor application. In order to obtain the principal (most precise) type for the functor application, we need to find a smallest X-free supertype of M_2 under the hypothesis $X : M$ (where M is the actual type of the argument m_2). This smallest non-dependent X-free supertype does not always exist (Lillibridge, 1997). In general, the set of non-dependent supertypes has several minimal elements. Thus, there is no hope of obtaining a type-checking algorithm that always returns principal types. Type-checking algorithms that do not always return principal types can still be useful in practice, but are less satisfactory in that they do not have clear specifications in the form of typing rules, and may fail in ways that are hard to understand for the programmer.

The Objective Caml type-checker implements one solution that is still incomplete with respect to rules 5′ and 7, but which is at least reasonably easy to understand for the programmer. Instead of trying to take an X-free supertype of the functor result type M_2, it just tries to take an X-free type type equivalent to M_2 under the hypothesis $X : M$. In the example above, `struct type t = int -> int end` is indeed an `X`-free type equivalent to `struct type t = X.t -> X.t end` under the hypothesis `X` : `sig type t = int end`. The good things about X-free equivalent types are that they do not lose typing information, and that they are easy to compute: just expand repeatedly type paths rooted at X that refer to manifest type components of M until either no reference to X remains (success) or we hit a path rooted at X referring to a type abstract in M (failure).

This approach can easily be added to our modular implementation. The `CORE_TYPING` structure must provide three additional functions:

```
module type CORE_TYPING =
  sig ...
    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
```

`nondep_valtype` $e\ x\ t$ should return a value type t' equivalent to t in the environment e, and such that x does not occur in t'. It proceeds by repeated expansion of manifest type paths rooted at x, as outlined above, and raises the `Not_found` exception if no such type t' exists. `nondep_deftype` and `nondep_kind` behave similarly on definable types and on kinds, respectively. Then, in the structure returned by the `Mod_typing` functor, we add the following functions that similarly remove dependencies on a given identifier in module types and signatures:

```
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 manifest' =
            match decl.manifest with
              None -> None
            | Some ty -> Some(CT.nondep_deftype env param ty) in
          let decl' =
            {kind = CT.nondep_kind env param decl.kind;
             manifest = manifest'} in
          Type_sig(id, decl') :: rem'
      | Module_sig(id, mty) ->
          Module_sig(id, nondep_modtype env param mty) :: rem'
```

Then, the type-checking of functor applications becomes:

```
let rec type_module env = function
    ...
  | 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 application")
      | _ -> error "application of a non-functor")
```

5.6 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 (Leroy, 1995). Besides facilitating the type-checking of nested functor applications such as `G(F(A))`, this extension enhances the expressive power of higher-order functors (functors taking functors as arguments), making them "fully transparent" in the terminology of (MacQueen & Tofte, 1994). A complete discussion of full transparency and applicative functors is beyond the scope of this paper; see (Leroy, 1995). 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 difficult case of 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)
```

In the absence of support for mutual recursion between structures, we are forced to use inelegant encodings at the level of the core language. 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 Objective Caml 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 like 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 like type theory, it is largely independent of the evaluation paradigm (Cardelli, 1989): 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 interesting issues. Languages not based on classes, such as Modula-3 (Nelson, 1991), are easily accommodated: the object-oriented features related to evaluation only (e.g. method invocation) are orthogonal to the module system, and the necessary subtyping between object types fits our generic module system very well. Even partially abstract types (types specified as any subtype of a given type) are easily handled by introducing power kinds (Cardelli, 1988) at the kind level.

For class-based object-oriented languages, it is possible to treat classes as another sort of structure components, along with values, types and sub-modules. This is the approach followed in Objective Caml (Leroy *et al.*, 1996). However, classes and inheritance can also be used as code structuring devices, partially overlapping the mechanisms provided at the module level. The Moby design (Fisher & Reppy, 1999) attemps to reduce this overlap by using module-level signature constraints to express some of the visibility modifiers often found in the class mechanism. Vouillon (1998) tried to go further by unifying classes with structures and inheritance with some forms of functors. Yet another direction is to use mixin modules or similar linking calculi to encode both functors and inheritance (Ancona & Zucca, 1998; Bracha, 1992).

Another interesting direction for future work is the application of module systems to logical frameworks and proof checkers, an area where the need is growing for decomposing large proofs in smaller units (Courant, 1997b).

On the implementation side, doubts have been expressed on the ability of the system presented here to scale to a full compiler for a real language. The main problem is the inefficiency of the environment lookup operations, due to the number of substitutions that have to be performed on the types of structure components at each path lookup. However, it is easy to amortize the cost of those substitutions through the use of more sophisticated data structures to represent the typing environment. As a case in point, the type-checker for modules used in the Objective Caml system is very close to the implementation presented in this paper, except that environments are represented by balanced binary trees in which all substitutions on the types of structure components are performed at insertion time, rather than at lookup time. This simple optimization suffices to obtain good performances even on large source programs.

Acknowledgements

The author is grateful to Claudio Russo, Philip Wadler and the anonymous referees for their comments and helpful suggestions for improving the exposition of this paper.

References

Ancona, Davide, & Zucca, Elena. (1998). A theory of mixin modules: Basic and derived operators. *Mathematical structures in computer science*, **8**(4), 401–446.

Ancona, Davide, & Zucca, Elena. (1999). A primitive calculus for module systems. *Pages 62–79 of:* Nadathur, Gopalan (ed), *PPDP'99 - international conference on principles and practice of declarative programming*. Lecture Notes in Computer Science, vol. 1702. Springer-Verlag.

Bracha, Gilad. (1992). *The programming language Jigsaw: Mixins, modularity and multiple inheritance*. Ph.D. thesis, University of Utah.

Cardelli, Luca. (1987). Basic polymorphic typechecking. *Science of computer programming*, **8**(2), 147–172.

Cardelli, Luca. (1988). Structural subtyping and the notion of power type. *Pages 70–79 of: 15th symposium Principles of Programming Languages*. ACM Press.

Cardelli, Luca. (1989). Typeful programming. *Pages 431–507 of:* Neuhold, E. J., & Paul, M. (eds), *Formal description of programming concepts*. Springer-Verlag.

Cardelli, Luca. (1990). *The Quest implementation*. Software and documentation available on `ftp://gatekeeper.dec.com/pub/DEC/Quest`.

Cardelli, Luca. (1998). Program fragments, linking, and modularization. *Pages 266–277 of: 24th symposium Principles of Programming Languages*. ACM Press.

Courant, Judicaël. (1997a). An applicative module calculus. *Pages 622–636 of:* Bidoit, M., & Dauchet, M. (eds), *TAPSOFT '97: Theory and practice of software development*. Lecture Notes in Computer Science, vol. 1214. Springer-Verlag.

Courant, Judicaël. (1997b). A module calculus for Pure Type Systems. *Pages 112 – 128 of: Typed lambda calculi and applications 97*. Lecture Notes in Computer Science, vol. 1210. Springer-Verlag.

Crary, Karl, Harper, Robert, & Puri, Sidd. (1999). What is a recursive module? *Pages 50–63 of: Programming Language Design and Implementation 1999*. ACM Press.

Crégut, Pierre, & MacQueen, David B. (1994). An implementation of higher-order functors. *Pages 13–21 of: Proc. 1994 workshop on ML and its applications*. Research report 2265, INRIA.

Curtis, P., & Rauen, J. (1990). A module system for Scheme. *Pages 13–19 of: Lisp and Functional Programming 1990*. ACM Press.

Duggan, Dominic, & Sourelis, Constantinos. (1996). Mixin modules. *Pages 262–273 of: International Conference on Functional Programming 96*. ACM Press.

Elsman, Martin. (1999). *Program modules, separate compilation, and intermodule optimisation*. Ph.D. thesis, Department of Computer Science, University of Copenhagen.

Fisher, Kathleen, & Reppy, John H. (1999). The design of a class mechanism for Moby. *Pages 37–49 of: Programming Language Design and Implementation 1999*. ACM Press.

Flatt, Matthew, & Felleisen, Matthias. (1998). Units: cool modules for HOT languages. *Pages 236–248 of: Programming Language Design and Implementation 1998*. ACM Press.

Glew, Neal, & Morrisett, Greg. (1999). Type-safe linking and modular assembly language. *Pages 250–261 of: 26th symposium Principles of Programming Languages*. ACM Press.

Guttag, John V., & Horning, James J. (1993). *Larch: languages and tools for formal specification*. Springer-Verlag.

Harper, Robert, & Lillibridge, Mark. (1994). A type-theoretic approach to higher-order modules with sharing. *Pages 123–137 of: 21st symposium Principles of Programming Languages*. ACM Press.

Harper, Robert, & Stone, Chris. (1998). A type-theoretic interpretation of Standard ML. Plotkin, Gordon, Stirling, Colin, & Tofte, Mads (eds), *Robin Milner Festschrifft*. MIT Press. A preliminary version is available as technical report CMU-CS-97-147, Carnegie Mellon University.

Harper, Robert, Mitchell, John C., & Moggi, Eugenio. (1990). Higher-order modules and the phase distinction. *Pages 341–354 of: 17th symposium Principles of Programming Languages*. ACM Press.

Leroy, Xavier. (1992). Unboxed objects and polymorphic typing. *Pages 177–188 of: 19th symposium Principles of Programming Languages*. ACM Press.

Leroy, Xavier. (1994). Manifest types, modules, and separate compilation. *Pages 109–122 of: 21st symposium Principles of Programming Languages*. ACM Press.

Leroy, Xavier. (1995). Applicative functors and fully transparent higher-order modules. *Pages 142–153 of: 22nd symposium Principles of Programming Languages*. ACM Press.

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

Leroy, Xavier, Vouillon, Jérôme, Doligez, Damien, *et al.* . (1996). *The Objective Caml system*. Software and documentation available on the Web, `http://caml.inria.fr/ocaml/`.

Lillibridge, Mark. (1997). *Translucent sums: a foundation for higher-order module systems*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

MacQueen, David B. (1986). Modules for Standard ML. Harper, Robert, MacQueen, David B., & Milner, Robin (eds), *Standard ML*. University of Edinburgh, technical report ECS LFCS 86-2.

MacQueen, David B. (1988). The implementation of Standard ML modules. *Pages 212–223 of: Lisp and Functional Programming 1988*. ACM Press.

MacQueen, David B., & Tofte, Mads. (1994). A semantics for higher-order functors. *Pages 409–423 of:* Sannella, D. (ed), *Programming languages and systems – ESOP '94*. Lecture Notes in Computer Science, vol. 788. Springer-Verlag.

Mauny, Michel, & de Rauglaudre, Daniel. (1994). A complete and realistic implementation of quotations in ML. *Pages 70–78 of: Proc. 1994 workshop on ML and its applications*. Research report 2265, INRIA.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The definition of Standard ML (revised)*. The MIT Press.

Nelson, Greg (ed). (1991). *Systems programming in Modula-3*. Prentice-Hall.

Nowak, David, Talpin, Jean-Pierre, Gautier, Thierry, & Le Guernic, Paul. (1997). An ML-like module system for the synchronous language Signal. *Pages 1244–1253 of: European conference on parallel processing (Euro-Par'97)*. Lecture Notes in Computer Science, no. 1300. Springer-Verlag.

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

Peyton-Jones, Simon L. (1987). *The implementation of functional programming languages*. Prentice-Hall.

Peyton-Jones, Simon L., & Launchbury, John. (1991). Unboxed values as first-class citizens in a non-strict functional language. *Pages 636–666 of: Functional programming languages and computer architecture 1991*. Lecture Notes in Computer Science, vol. 523. Springer-Verlag.

Rémy, Didier. (1992). *Extending ML type system with a sorted equational theory*. Research report 1766. INRIA.

Russo, Claudio V. (1998). *Types for modules*. Ph.D. thesis, LFCS, University of Edinburgh.

Sannella, D. T., & Wallen, L. A. (1992). A calculus for the construction of modular Prolog programs. *Journal of logic programming*, **12**, 147–177.

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

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., & Lee, P. (1996). TIL: a type-directed optimizing compiler for ML. *Pages 181–192 of: Programming Language Design and Implementation 1996.* ACM Press.

Vouillon, Jérôme. (1998). Using modules as classes. *Informal proceedings of the FOOL'5 workshop.*

Weis, Pierre, & Leroy, Xavier. (1999). *Le langage Caml.* Dunod.

Wirsing, Martin. (1990). Algebraic specifications. *Pages 675–788 of:* van Leeuwen, Jan (ed), *Handbook of theoretical computer science, volume B.* The MIT Press/Elsevier.

Wright, Andrew K. (1995). Simple imperative polymorphism. *Lisp and symbolic computation*, **8**(4), 343–356.

A Typing rules for the module system

Notations:

Module expressions:	$m ::= p \mid \texttt{struct}\ s\ \texttt{end} \mid (m : M)$
	$\mid \texttt{functor}(X_i : M)\, m \mid m(p)$
Structures:	$s ::= \varepsilon \mid c;\ s$
Structure components:	$c ::= \texttt{val}\ v_i = e \mid \texttt{type}\ t_i :: \kappa = \tau_d \mid \texttt{module}\ X_i = m$
Module types:	$M ::= \texttt{sig}\ S\ \texttt{end} \mid \texttt{functor}(X_i : M)\, M'$
Signatures:	$S ::= \varepsilon \mid C;\ S$
Signature components:	$C ::= \texttt{val}\ v_i : \tau_v \mid \texttt{type}\ t_i :: \kappa \mid \texttt{type}\ t_i :: \kappa = \tau_d$
	$\mid \texttt{module}\ X_i : M$
Typing environments:	$E ::= \varepsilon \mid E;\ C$
Core expressions:	$e ::= v_i \mid p.v \mid \ldots$
Core value types:	$\tau_v ::= \ldots$
Core definable types:	$\tau_d ::= \ldots$
Core kinds:	$\kappa ::= \ldots$
Access paths:	$p ::= X_i \mid p.X$

We write Dom(S) and Dom(E) for the set of identifiers bound in the structure S or the environment E. Identifiers v_i, t_i, X_i are bound by `struct`, `sig` and `functor` constructs, and can be alpha-converted provided their name part v, t, X do not change. We assume given the following typing judgements for the core language:

$E \vdash e : \tau_v$	expression e has value type τ_v
$E \vdash \tau_d :: \kappa$	definable type τ_d has kind κ
$E \vdash \kappa\ \texttt{wf}$	kind κ is well-formed
$E \vdash \tau_v\ \texttt{wf}$	value type τ_v is well-formed
$E \vdash \tau_v <: \tau_v'$	value type τ_v is a subtype of τ_v'
$E \vdash \kappa <: \kappa'$	kind κ is a subkind of κ'
$E \vdash \tau_d \approx \tau_d' :: \kappa$	definable types τ_d and τ_d' are equivalent at kind κ

Typing of modules $E \vdash m : M$ **and structures** $E \vdash s : S$

$$\frac{E = E_1;\ \texttt{module}\ X_i : M;\ E_2}{E \vdash X_i : M} \ (1)$$

$$\frac{E \vdash p : \texttt{sig}\ S_1;\ \texttt{module}\ X_i : M;\ S_2\ \texttt{end}}{E \vdash p.X : M\{z_j \leftarrow p.z \mid z_j \in \mathrm{Dom}(S_1)\}} \ (2)$$

$$\frac{E \vdash s : S \qquad \text{components of } s \text{ have distinct names}}{E \vdash (\texttt{struct}\ s\ \texttt{end}) : (\texttt{sig}\ S\ \texttt{end})} \ (3)$$

$$\frac{E \vdash M'\ \texttt{wf} \qquad X_i \notin \mathrm{Dom}(E) \qquad E;\ \texttt{module}\ X_i : M' \vdash m : M}{E \vdash (\texttt{functor}(X_i : M')\, m) : (\texttt{functor}(X_i : M')\, M)} \ (4)$$

$$\frac{E \vdash m : \texttt{functor}(X_i : M')\, M \qquad E \vdash p : M'}{E \vdash m(p) : M\{X_i \leftarrow p\}} \ (5) \qquad \frac{E \vdash M\ \texttt{wf} \qquad E \vdash m : M}{E \vdash (m : M) : M} \ (6)$$

$$\frac{E \vdash M\ \texttt{wf} \qquad E \vdash M' <: M \qquad E \vdash m : M'}{E \vdash m : M} \ (7) \qquad \frac{E \vdash p : M}{E \vdash p : M/p} \ (8)$$

$$E \vdash \varepsilon : \varepsilon \ (9) \qquad \frac{E \vdash e : \tau_v \qquad v_i \notin \mathrm{Dom}(E) \qquad E;\ \texttt{val}\ v_i : \tau_v \vdash s : S}{E \vdash (\texttt{val}\ v_i = e;\ s) : (\texttt{val}\ v_i : \tau_v;\ S)} \ (10)$$

$$\frac{E \vdash \kappa\ \texttt{wf} \qquad E \vdash \tau_d :: \kappa \qquad t_i \notin \mathrm{Dom}(E) \qquad E;\ \texttt{type}\ t_i :: \kappa = \tau_d \vdash s : S}{E \vdash (\texttt{type}\ t_i :: \kappa = \tau_d;\ s) : (\texttt{type}\ t_i :: \kappa = \tau_d;\ S)} \ (11)$$

$$\frac{E \vdash m : M \qquad E;\ \texttt{module}\ X_i : M \vdash s : S}{E \vdash (\texttt{module}\ X_i = m;\ s) : (\texttt{module}\ X_i : M;\ S)} \ (12)$$

Well-formedness of module types $E \vdash M\ \texttt{wf}$ **and signatures** $E \vdash S\ \texttt{wf}$

$$\frac{E \vdash S\ \texttt{wf} \qquad \text{components of } S \text{ have distinct names}}{E \vdash (\texttt{sig}\ S\ \texttt{end})\ \texttt{wf}} \ (13)$$

$$\frac{E \vdash M_1\ \texttt{wf} \qquad E;\ \texttt{module}\ X_i : M_1 \vdash M_2\ \texttt{wf}}{E \vdash (\texttt{functor}(X_i : M_1)\, M_2)\ \texttt{wf}} \ (14) \qquad E \vdash \varepsilon\ \texttt{wf} \ (15)$$

$$\frac{E \vdash \tau_v\ \texttt{wf} \qquad E \vdash S\ \texttt{wf}}{E \vdash (\texttt{val}\ v_i : \tau_v;\ S)\ \texttt{wf}} \ (16) \qquad \frac{E \vdash \kappa\ \texttt{wf} \qquad E;\ \texttt{type}\ t_i :: \kappa \vdash S\ \texttt{wf}}{E \vdash (\texttt{type}\ t_i :: \kappa;\ S)\ \texttt{wf}} \ (17)$$

$$\frac{E \vdash \kappa\ \texttt{wf} \qquad E \vdash \tau_d :: \kappa \qquad E;\ \texttt{type}\ t_i :: \kappa = \tau_d \vdash S\ \texttt{wf}}{E \vdash (\texttt{type}\ t_i :: \kappa = \tau_d;\ S)\ \texttt{wf}} \ (18)$$

$$\frac{E \vdash M\ \texttt{wf} \qquad E;\ \texttt{module}\ X_i : M \vdash S\ \texttt{wf}}{E \vdash (\texttt{module}\ X_i : M;\ S)\ \texttt{wf}} \ (19)$$

Subtyping between module types $E \vdash M <: M'$ and between signature components $E \vdash C <: C'$

$$\frac{\sigma : \{1 \ldots m\} \to \{1 \ldots n\} \qquad \mathrm{Dom}(C_1;\ \ldots;\ C_n) \cap \mathrm{Dom}(E) = \emptyset \qquad E;\ C_1;\ \ldots;\ C_n \vdash C_{\sigma(i)} <: C'_i \ \text{ for } i = 1 \ldots m}{E \vdash (\mathtt{sig}\ C_1;\ \ldots;\ C_n;\ \varepsilon\ \mathtt{end}) <: (\mathtt{sig}\ C'_1;\ \ldots;\ C'_m;\ \varepsilon\ \mathtt{end})} \quad (20)$$

$$\frac{E \vdash M'_1 <: M_1 \qquad E;\ \mathtt{module}\ Y_j : M_1 \vdash M_2\{X_i \leftarrow Y_j\} <: M'_2}{E \vdash (\mathtt{functor}(X_i : M_1)\, M_2) <: (\mathtt{functor}(Y_j : M'_1)\, M'_2)} \quad (21)$$

$$\frac{E \vdash \tau_v <: \tau'_v}{E \vdash (\mathtt{val}\ v_i : \tau_v) <: (\mathtt{val}\ v_i : \tau'_v)} \quad (22)$$

$$\frac{E \vdash M <: M'}{E \vdash (\mathtt{module}\ X_i : M) <: (\mathtt{module}\ X_i : M')} \quad (23)$$

$$\frac{E \vdash \kappa <: \kappa'}{E \vdash (\mathtt{type}\ t_i :: \kappa) <: (\mathtt{type}\ t_i :: \kappa')} \quad (24)$$

$$\frac{E \vdash \kappa <: \kappa'}{E \vdash (\mathtt{type}\ t_i :: \kappa = \tau_d) <: (\mathtt{type}\ t_i :: \kappa')} \quad (25)$$

$$\frac{E \vdash \kappa <: \kappa' \qquad E \vdash \tau_d \approx \tau'_d :: \kappa'}{E \vdash (\mathtt{type}\ t_i :: \kappa = \tau_d) <: (\mathtt{type}\ t_i :: \kappa' = \tau'_d)} \quad (26)$$

$$\frac{E \vdash \kappa <: \kappa' \qquad E \vdash t_i \approx \tau_d :: \kappa'}{E \vdash (\mathtt{type}\ t_i :: \kappa) <: (\mathtt{type}\ t_i :: \kappa' = \tau_d)} \quad (27)$$

Strengthening of module types M/p and signatures S/p

$$\begin{aligned}
(\mathtt{sig}\ S\ \mathtt{end})/p &= \mathtt{sig}\ S/p\ \mathtt{end} \\
(\mathtt{functor}(X_i : M_1)\, M_2)/p &= \mathtt{functor}(X_i : M_1)\, M_2 \\
\varepsilon/p &= \varepsilon \\
(\mathtt{val}\ v_i : \tau_v;\ S)/p &= \mathtt{val}\ v_i : \tau_v;\ S/p \\
(\mathtt{type}\ t_i :: \kappa;\ S)/p &= \mathtt{type}\ t_i :: \kappa = p.t;\ S/p \\
(\mathtt{type}\ t_i :: \kappa = \tau;\ S)/p &= \mathtt{type}\ t_i :: \kappa = \tau;\ S/p \\
(\mathtt{module}\ X_i : M;\ S)/p &= \mathtt{module}\ X_i : M/p.X;\ S/p
\end{aligned}$$