First-class modules and composable signatures in Objective Caml 3.12

Alain Frisch* Jacques Garrigue†

1 Introduction

Notwithstanding its name, Objective Caml [3] has a full-fledged SML-style module system. Its applicative functors allow for flexible parameterization of components, and many libraries, including the preprocessor Camlp4, use them for their structure. More recent extensions include the addition of recursive modules [2], and private row types [1], which allow even more involved structuring.

While this module language is overall successful, as demonstrated by its active use, it has also some well-known weaknesses. Among them, one can mention its mostly static nature —despite the availability of local module definitions inside expressions, modules remained second class citizens— and the absence of compositionality of its signature language. In this presentation we show how these two weaknesses were alleviated in Objective Caml 3.12, and what are the concrete applications of these features.

2 First-class modules

Traditionnaly, modules systems in the ML family of languages are stratified in two well-separated sub languages: a base language which enjoys automatic type inference in the spirit of the Hindley-Milner type system (with extensions), and an explicitly typed module language built on top of the base language. While base types and values can of course appear in the module system, module values (structures and functors) cannot appear in base values and module types (signatures and functor types) cannot appear in base types.

Nevertheless, modules are not as static as one might think. First, the compilation scheme followed by Objective Caml represents modules values as regular runtime blocks: a structure is compiled in the same way as a record (by keeping only its dynamic components: values, sub-modules, exceptions), and functors are compiled into functions. Second, Objective Caml already allowed a module expression to be evaluated within a base expression, using the syntax `let` *M* = *module-expr* `in` *expr*.

What was missing was the ability to turn module values into base values, in order to put them in larger data structures, pass them to functions or return them, manipulate them with usual base language constructions (conditionals, pattern matching), and so on. We implemented a variant of Russo's proposal[5].

Typical uses of first-class modules are such as the choice of a concrete implementation for a module at runtime (depending for instance on command-line argument), or the encoding of existential types (packing together a type and some values and operations on this type into a value). Combining these, one can also use first-class modules for providing well-typed plug-ins.

The changes to the syntax are as follows:

expr ::= ...
| (`module` *module-expr* `:` *package-type*)
module-expr ::= ... | (`val` *expr* `:` *package-type*)
type ::= ... | (`module` *package-type*)
package-type ::= *modtype-path*
| *modtype-path* `with type` *t* = *type*
{`and type` *t* = *type*}*

The new kind of expression packs a module into a value of the base language (a packed module). Dually, the new kind of module expression opens a packed module into a real module. The type system allows to use this construction only in top-level structures and in local module expression, but not in the body of functors. Otherwise, the construction would be type-unsafe because of a bad interaction with applicative functors: a type path such as *F(M).t* could refer to several different types at runtime if first-class modules could be opened within the body of the functor *F*.

Package types are used to give a type to packed modules. They form a sub-category of module types, where only named module types (with constraints on ground types) are allowed. This restriction, compared to Russo's proposal, allowed for a minimal implementation on top of the existing Objective Caml type-checker. Two package types are deemed equal if their module types path are equal (path equality) and if they have the same constrainted types types (modulo permutation of constraints) with equal right-hand sides. Concerning unification and other type-related algorithms, a package type behaves similarly to an *n*-ary type constructor (where *n* is the number of type constraints). Type checking for the new forms of expressions and module expressions is easy because the package type is given explicitly. In many cases, the package type could be inferred, but care is needed in order not to break principality of type-checking. We leave this to future work.

Objective Caml assumes that type variables are local to the current expression or type declaration. For some uses of first-class modules, it is useful to refer to a surrounding type variable within a local structure. To support this situation (and others which are not related to first-class modules), we further extended the base language with an extra construction, *abstract type parameters*:

expr ::= ... | `fun (type` *t*`) ->` *expr*

The behavior of this construction is to make the type name *t* available within the body of *expr* as an abstract type which

*LexiFi
†Nagoya University Graduate School of Mathematics

cannot escape to the context. The type for the expression is obtained from the type of *expr* by replacing *t* with a fresh type variable. In effect, this new construction allows to name a type variable within the body of an expression and to ensure that it is kept generic locally. Contrary to what the syntax may suggest, it does not suspend the evaluation of the *expr* and no type is actually passed at runtime, but the ability to mix type parameters with other arguments allows for a natural formulation of first-class polymorphic functors.

Combining first-class modules, locally abstract types, and polymorphic recursion (also freshly available in Objective Caml 3.12), it is also possible to encode GADTs, by passing arround a dynamic representation of type equations. Such an encoding was already possible using objects with polymorphic methods, but the encoding based on first-class modules is both more powerful (one may quantify on type *constructors*) and more intuitive (thanks to existential types).

3 Composing signatures

A recurrent gripe with Objective Caml signatures has been their lack of compositionality. Namely, suppose that we have the following signatures *Printable* and *Comparable*:

```
module type Printable =
  sig type t  val print : t -> unit end
module type Comparable =
  sig type t  val compare : t -> t -> int end
```

How can we compose them in a signature *PrintableComparable*, with a single type and two functions? While there is an `include` construct for signatures, it will not merge identical types, resulting in the following error:

```
module type PrintableComparable = sig
  include Printable
  include Comparable with type t = t
end
Error: Multiple definition of the type name t.
       Names must be unique in a given
       structure or signature.
```

Eventhough we told explicitly that the two `t`'s are identical (here the right hand `t` refers to the `t` imported from *Printable*), we have an error about repeated definitions.

Ramsey et al. [4] already gave concrete examples of features lacking when writing signatures, and proposed a full-fledged signature language to replace the existing one. Their language offers some solutions to the merging problem, but it seemed too large for inclusion in Objective Caml. Instead, we chose to identify the most important missing patterns (including the above one), and support them with a minimal number of constructs.

Along with the above merging of signatures, other commonly missed features were the ability to rename components in a signature, or to remove them form the signature. Since all these behaviours are linked with the concepts of substitution and removal, we finally came up with a new construct, *destructive substitution*, which extends the `with` constraints of module types, by additionally removing the definitions from the signature after substituting them. The syntax for the new constraints is:

mod-constraint ::= ...
| `type` *[type-parameters] typeconstr* `:=` *typexpr*
| `module` *module-name* `:=` *extended-module-path*

We can see its effect in the following example:

```
module type ComparableInt =
  Comparable with type t := int
module type ComparableInt =
  sig val compare : int -> int -> int end
```

As a result, our merging problem can be solved by using the constraint `with type t := t`. One can also rename type or module components by adding the same definition and destructively substituting it, or remove them by destructively substituting with an existing definition.

The current implementation restricts the substituted type or module to be at the toplevel of the target signature, and the substitute to be a path. The first restriction seems difficult to avoid, since internal references to a module enclosing a removed component could cause problems. The second one is just due to our choice of using path substitution for simplicity; in general, it seems all right to substitute with an arbitrary type expression, and we might do so in the future.

While this new construction combines several features in one, its real intuition is that it provides a functional view of signatures. Namely, it allows to see a signature as a function from some of its type and module components to its other components. `Comparable with type t := ...` can be seen as equivalent to the following functor.

```
module Comparable(X:sig type t end) = struct
  module type S =
    sig val comparable : X.t -> X.t -> int end
end
```

Thanks to destructive substitution we can convert from the signature form to the functor form. This is useful, as the signature form is more compact and intuitive, but the functor form is more flexible.

References

[1] J. Garrigue. Private rows: abstracting the unnamed. In *Proc. Asian Symposium on Programming Languages and Systems*, volume 4279 of *Springer LNCS*, Sydney, Nov. 2006.

[2] X. Leroy. A proposal for recursive modules in Objective Caml. Available from `http://caml.inria.fr/about/papers.en.html`, May 2003.

[3] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.11, Documentation and user's manual*. Projet Gallium, INRIA, Nov. 2008.

[4] N. Ramsey, K. Fisher, and P. Govereau. An expressive language of signatures. In *Proc. International Conference on Functional Programming*, 2005.

[5] C. V. Russo. First-class structures for Standard ML. In *Proc. European Symposium on Programming*, volume 1782 of *Springer LNCS*, pages 336–350, Mar. 2000.