On the design and implementation of Modular Explicits

Samuel Vivien, Didier Rémy

On the design and implementation of Modular Explicits

Samuel Vivien* Didier Rémy†

July 26, 2024

Abstract

We present and discuss the design and implementation of *modular explicits*, an extension of OCaml first-class modules with *module-dependent functions*, functions taking first-class modules as arguments. We show some difficulties with the present use of first-class modules and how modular explicits solve them in a simpler, more direct way. Modular explicits are fully compatible with, and can be presented as an extension of, first-class modules. Interestingly, both the formalization and the implementation reuse the mechanism designed to ensure principal types in the presence of semi-explicit first-class polymorphism and OCaml polymorphic methods. Modular explicits are also meant to be the underlying language in which *modular implicits*, i.e., module arguments left implicit from their signatures, should be elaborated.

First-class modules in OCaml

First-class modules, a feature present in OCaml since 3.12, allow a module M of signature S to be packed as a value m of type (**module** S) using the construct (**module** $M : S$), which can then be stored in data-structures or passed as argument to functions. A first-class module such as m, i.e., a value of *known* type (**module** S) can then be unpacked with the construct

$$\textbf{let module } \mathcal{X} = (\textbf{val } m) \textbf{ in } a[X]$$

This binds $\mathcal{X}$ to a module of type S that can be used in the expression a. If m were of an unknown type, we should write $(m : (\textbf{module } S))$ instead of m. This situation occurs in particular when x is a function argument, as in

$$\textbf{fun } x \to \textbf{let module } \mathcal{X} = (\textbf{val } (x : (\textbf{module } S))) \textbf{ in } a[X] \qquad (\mathbf{1})$$

or equivalently, moving the annotation to the binding site,

$$\textbf{fun } x : (\textbf{module } S) \to \textbf{let module } \mathcal{X} = (\textbf{val } x) \textbf{ in } a[X]$$

Here, the first-class module is immediately unpacked as a module $\mathcal{X}$ which can then be used in $a[\mathcal{X}]$. OCaml offers a concise abbreviation for this is common pattern,

$$\textbf{fun } (\textbf{module } \mathcal{X} : S) \to a[\mathcal{X}]$$

giving the (useful) illusion that the function directly receives a module as argument.

*OCamlPro and École Normale Supérieure PSL, France
†Inria Paris Research Centre, France

Unfortunately, there is some limitation to this pattern when the signature S contains an abstract type component, say **type** t. Looking at the equivalent form (1), it is clear that the first-class module type (**module** S) of x contains an abstract type t, which when unpacked becomes the abstract type $\mathcal{X}.t$ with a scope limited to the let-binding body, i.e., $a[\mathcal{X}]$. That is, $\mathcal{X}.t$ should not occur free in the type of $a[\mathcal{X}]$, which is also the type of the let-binding, as it would otherwise escape from its scope.

A possible work around?

When we write **fun** (**module** $\mathcal{X} : S$) $\to a$ in such a case, we rather mean a polymorphic function of type $\forall\alpha.\,(\textbf{module}\ S\ \textbf{with type}\ t = \alpha) \to \tau[\alpha]$, say σ, That is,

$$\textbf{fun}\ (\textbf{type}\ \alpha) \to \textbf{fun}\ (\textbf{module}\ \mathcal{X} : S\ \textbf{with type}\ t = \alpha) \to a[\mathcal{X}]$$

which accepts as argument any module compatible with the signature S, i.e., with a type component t that might be abstract or any concrete type, much as during the application of a functor **functor** $(\mathcal{X} : S) \to M'$. Using a universal binder that extends to the whole arrow type, instead of an existential binder limited to the right-hand side, avoids the scope escaping problem.

Notice however that this amounts to re-encoding the convenient module-level type abstraction mechanism into the core language, somewhat along the lines of Blaudeau et al. (2024), hence loosing the convenient and concise path-based approach of OCaml modules. In particular, it does not scale well when S has multiple abstract types.

Besides, we still have to deal with first-class polymorphism when passing such a function, say f, to another higher-order function, say *happy*, that will apply f several times to module arguments of different types:

$$\textbf{let}\ happy\ (f : \sigma) = \ \ldots\ f(\textbf{module}\ M_1)\ \ldots\ f(\textbf{module}\ M_2)\ \ldots$$

a pattern that often happens in some use cases of modular explicits, such as the emulation of overloading.

Unfortunately, this is not allowed in OCaml when σ is polymorphic as is the case here. OCaml offers some but still poor support for first-class polymorphism. We need to encapsulate σ, either using semi-explicit polymorphism as proposed by Garrigue and Rémy (1999) via objects or records polymorphic fields—or using some module-level wrapper. Examples of such encodings are given by Vivien et al. (2024).

In fact, the encoding of abstract types with universal types, which may be a work around in some cases, is not possible in OCaml when t is a higher-order abstract type, such as List.t, since OCaml core-level type variables cannot be of higher-rank.

A correct, but unpractical encoding

An alternative solution is to view and encode **fun** (**module** $\mathcal{X} : S$) $\to a$ altogether as a first-class functor

$$(\textbf{module functor}\ (\mathcal{X} : S) \to \textbf{struct}\ value = a\ \textbf{end} : S')$$

whose signature S', equal to $\mathcal{X}.t \to \textbf{sig val}\ value : \tau[\mathcal{X}.t]\ \textbf{end}$, should be *named* and *explicitly* given[1] to build the first-class value (or when passing it to a function such as *happy*). An example of this encoding is given by Vivien et al. (2024).

[1] One may sometimes work around using the **module type of** construct to derive S' from the inferred type of the functor before packing the functor.

This encoding can also cope with higher-order abstract types. Actually, it seems to cover all use cases, showing that modular explicits do not provide additional expressiveness—if we ignore conciseness and the amount of type annotations. Indeed, it requires quite a few module-types manipulations, which may quickly become cumbersome, as module types cannot be passed directly to functors in OCaml but only when embedded in structures—which must themselves be given a signature. Hence, this encoding is quite unpractical at large scale. As shown by Vivien et al. (2024), a one-line definition with modular explicits may expand to a dozen-line implementation with lots of boiler-plate code in plain OCaml.

Presentation of modular explicits

Modular explicits are a solution to the limitation of first-class modules, which they extend by introducing a new type construction in the language, the module-dependent arrow $(\textbf{module}\ \mathcal{X} : S) \to \tau$ where $\mathcal{X}$ may occur free in τ (as the origin of a path leading to an abstract type such as $\mathcal{X}.P.t$). The difference between the module-dependent arrow $(\textbf{module}\ \mathcal{X} : S) \to \tau$ and the usual arrow $(\textbf{module}\ S) \to \tau$ whose domain is a module type is syntactically small, but technically significant: $\mathcal{X} : S$ acts as a binder for $\mathcal{X}$ with some limited scope, as $\mathcal{X}$ may appear in (and only in) the codomain type τ. Hence, $(\textbf{module}\ \mathcal{X} : S) \to \tau$ behaves as a first-class polymorphic type[2] while $S \to \tau$ behaves as a simple type.

Still, when $\mathcal{X}$ does not appear in τ, the two forms mean the same and the implementation will actually turn the module-dependent arrow type into a usual arrow type whose domain is a first-class module type.

The typing of modular explicits is sketched by Vivien et al. (2024) using an excerpt of OCaml. As a result of the overlapping of the two kinds of arrows, there is also an overlapping of the typing rules. However, the typing rules have been designed to always agree when they overlap, which relies on the fact that the domain of module-dependent arrow should always be *known* for the type to be treated as a module-dependent arrow. To formalize this concept, we must reveal a detailed that we have hidden so far for sake of simplicity. Arrow types $(\textbf{module}\ \mathcal{X} : S) \to^{\epsilon} \tau$ and $\tau_1 \to^{\epsilon} \tau_2$ both carry an additional parameter ϵ, called a node-variable, that allows to distinguish between types that are known (when ϵ is generalizable in the current context) and can be treated as dependent arrows and types that are not yet known and should always be treated as non-dependent arrow types. Node variables are of a special kind and incompatible with usual type variables. This mechanism, introduced in OCaml for typechecking polymorphic methods and record fields and formalized by Garrigue and Rémy (1999), ensures that type inference does not take a decision depending on the order in which type inference and unification constraints are solved, so that inferred types are principal. More details are given by Vivien et al. (2024).

Interestingly, the way we distinguish *known* from *guessed* types of module-dependent-functions is also similar to a recent proposal by White (2023) for adding *semi-explicit polymorphic parameters*.

Compatibility with first-class modules

Module-dependent functions are only typing artifacts: their runtime representations are the same as that of functions taking first-class modules as arguments. This allows code-free coercions from the type of the former to the type of the latter. In the implementation, the tricky part

[2] Module-dependent arrow types are actually no more than first-class polymorphic types and in no way a general form of dependent types. This agrees with the interpretation of modules in F^{ω} by Blaudeau et al. (2024) and the specific positions of quantifiers in this interpretation.

is the unification of dependent and non-dependent arrow types, which is possible under certain conditions. Namely, $(\textbf{module}\ \mathcal{X} : S_1) \to \tau_1$ and $(\textbf{module}\ S_2) \to \tau_2$ are unifiable if and only if:

- inlining $\mathcal{X}$ in τ_1 leads to (via type equivalence) a type $\tau_1{}'$ that does not contain a reference to $\mathcal{X}$ and
- $(\textbf{module}\ S_1) \to \tau_1{}'$ and $(\textbf{module}\ S_2) \to \tau_2$ are unifiable.

Summary

Modular explicits are a small extension to first-class modules as they do not actually increase expressiveness, but an extension that considerably improves conciseness thanks to a better interaction between the core and module levels, making module-level first-class functors become core-level module-dependent functions, and thus enabling some programming with modules directly at the core level without boiler-plate encodings.

Although types of module-dependent functions and functions over first-class modules are two different constructions that are typed differently, their overlapping is made mostly transparent to the user, who only sees one kind of arrow that can be treated as a module-dependent arrow type when its domain is *known* and the context *allows* it. Interestingly, the implementation reuses the OCaml existing trick to keep track of principal types and smoothly move from module-dependent arrow types to non-dependent arrow types when needed.

Although modular explicits have been originally designed as the language in which *modular implicits* proposed by White et al. (2014) will be elaborated, many examples of modular explicits need not implicit arguments to be usable—or need not them at all. Hence, modular explicits are useful for themselves and should make their way to the compiler as soon as possible. At the time of writing, there is an implementation of modular explicits by Vivien (2024) that is under review before merging it in the OCaml compiler[3]. Vivien et al. (2024) give a formal presentation of modular explicits that should also serve as a reference specification for this implementation. They also provide a wide range of motivating examples for modular explicits.

Acknowledgments

We would like to thank Vincent Laviron, the Flambda team, and all the other people at OCamlPro for hosting the internship that lead to this implementation. We also thank Leo White for his advices and code reviewing, but also all the other people who have contributed the reviewing process of the implementation. Some ideas emerged during previous work on modular explicits with Thomas Refis. We also wish to acknowledge the work of Matthew Ryan on his own implementation of modular explicits[4], which lead to various discussions on github that gave an insight on their implementation. Reader and Vlasits (2024) wrote a significant amount of examples (Reader et al., 2024) for *modular implicits* that can be used with *modular explicits* when doing the elaboration by hand. They also conclude with a list of desirable features that should be implemented to reach a satisfying status for *modular implicits*.

References

C. Blaudeau, D. Rémy, and G. Radanne. Fulfilling ocaml modules with transparency. In *Proceedings of the 2024 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '24, New York, NY, USA, 2024. ACM.

[3] An experimental version of the compiler with modular explicits can be installed with opam following instructions at https://github.com/samsa1/modular-compiler-variants.

[4] Available at https://github.com/ocaml/ocaml/pull/9187

J. Garrigue and D. Rémy. Extending ML with semi-explicit higher-order polymorphism. *Information and Computation*, 155(1/2):134–169, 1999. URL http://www.springerlink.com/content/m303472288241339/. A preliminary version appeared in TACS'97.

P. Reader and D. Vlasits. Modular implicits internship report, 2024. URL https://github.com/modular-implicits.github.io/report.pdf.

P. Reader, D. Vlasits, L. White, and J. Yallop. A repository of modular implicits packages, 2024. URL https://github.com/modular-implicits/modular-implicits-opam.

S. Vivien. Modular explicits, June 2024. URL https://github.com/samsa1/modular-compiler-variantshttps://github.com/samsa1/modular-compiler-variants. Available as an OCaml variant on github.

S. Vivien, D. Rémy, T. Refis, and G. Scherer. On the design and implementation of modular explicits. Draft, July 2024. URL https://gallium.inria.fr/~remy/ocamod/modular-explicits.pdf.

L. White. Semi-explicit polymorphic parameters. Presentation at the Higher-order, Typed, Inferred, Strict: ML Family workshops, sep 2023.

L. White, F. Bour, and J. Yallop. Modular implicits. In O. Kiselyov and J. Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, volume 198 of *EPTCS*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2. URL https://doi.org/10.4204/EPTCS.198.2.