On the design and implementation of Modular Explicits

Samuel Vivien* Didier Rémy† Thomas Refis‡ Gabriel Scherer§

Draft version 0.5 of July 2, 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.

Introduction

The name *modular explicits* is coined from *modular implicits* an extension proposed by White, Bour, and Yallop (2014) to provide OCaml with a mechanism similar to Haskell type classes but based on, and compatible with, the module system of OCaml. This builds on two ideas: implementing type classes as implicit arguments in Scala proposed by Oliveira, Moors, and Odersky (2010), but emulating type classes at the module-level rather then at the core-level as proposed by Dreyer, Harper, Chakravarty, and Keller (2007), hence, using implicit module arguments rather than implicit core-level values, which has many benefits as explained by White, Bour, and Yallop (2014). After synthesis, implicit module arguments should become explicit modules, and the program be, at least internally, usual OCaml code. However, this elaboration would be quite involved and verbose in the current version of OCaml to the point of being hardly readable. The reason is that modular implicits require a tight interaction between the core language and the module language that is only possible via first-class modules, which were not originally present in the OCaml module language, designed for programming in the large, and are still limited.

Modular explicits are an extension of the OCaml module system that facilitates this interaction. They do not strictly increase expressiveness, but they considerably increase conciseness and, in particular, reduce the amount of type annotations and boilerplate code that the user should otherwise write. While modular explicits are also designed to be used as the target of the elaboration of modular implicits, they are actually a standalone proposal, with its own interesting programming patterns, independently of the possibility of leaving some module arguments

*OCamlPro and École Normale Supérieure, PSL Research University, France
†Inria Paris, France
‡Contributed while at Inria Paris and Tarides, France
§Inria Paris, France

implicits. In order to help manipulate first-class functors, they give the illusion of abstraction over module arguments in the core language. For this purpose, a new construct called *module-dependent functions*, and its counterpart, module-dependent applications, are added to expressions of the core language—they actually connect expressions of the core and module languages.

Implementation

At the time of writing, there is an implementation of modular explicits by Vivien (2024a) that is almost ready to be submitted to the OCaml compiler[1][2]. This document should also serve as a reference specification for this implementation. Examples marked in blue below have been processed automatically. When printed, their output appears in green. Programs that do not typechecked are marked in red. These may fail to typecheck or use some extension not yet available in the prototype. Other examples left in gray have not been processed.

Plan

We start with a quick overview of modular explicits (§1). We then give a more detailed description of modular explicits with motivating examples (§2), a formal presentation (§3), and implement details (§4). We end with discussions (§6) on modular explicits and further extensions.

1 Overview

1.1 The limit of first-class modules

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** $M : S$). A first-class module 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[3]. 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

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

or equivalently, moving the annotation to the binding site,

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

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

$$\lambda(\textbf{module } \mathcal{X} : S). a[\mathcal{X}]$$

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

[1] An experimental version of the compiler with modular explicits can be installed with `opam`, following instructions give at `https://github.com/samsa1/modular-`σ_2`compiler-`σ_2`variants`.

[2] A previous implementation had been proposed by Ryan (2019).

[3] By writing $a[\mathcal{X}]$ we mean an expression a in which $\mathcal{X}$ may occur.

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., $a[\mathcal{X}]$. Therefore, $\mathcal{X}.t$ cannot 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.

Example 1: fails with OCaml version 5.2.0

```
module type Type = sig type t end
let id (module A : Type) (x : A.t) = x
```

```
Error: This pattern matches values of type "A.t"
       but a pattern was expected which matches values of type "'a"
       The type constructor "A.t" would escape its scope
```

1.2 A tentative work around

When we write the function $\lambda(\textbf{module}\ \mathcal{X} : S).\, a$, we do not really mean a function of type $(\exists\alpha.\, (\textbf{module}\ S[\alpha])) \to \tau[\alpha]$ but rather a polymorphic function of type $\forall\alpha.\, (\textbf{module}\ S[\alpha]) \to \tau[\alpha]$, that is:

$$\lambda(\textbf{type}\ \alpha).\, \lambda(\textbf{module}\ \mathcal{X} : S[\alpha]).\, a[\mathcal{X}]$$

where $S[\alpha]$ stands for S **with type** $t = \alpha$. Hence, a function that accepts as argument any module compatible with the signature S, i.e., with a type component t that might itself be an abstract type 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 that could not appear in the type of the right-hand side, avoids the scope escaping problem.

```
module type Type = sig type t end
let id (type a) (module A : Type with type t = a) (x : A.t) = x
```

```
val id : (module A : Type with type t = 'a) → A.t → A.t = <fun>
```

Notice however that this amounts to re-encoding the convenient module-level type abstraction mechanism into the core language, somewhat along the lines proposed by Blaudeau, Rémy, and Radanne (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, possibly introduced in submodules.

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 occurs 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. Currently, we need to encapsulate σ, either using semi-explicit polymorphism (Garrigue and Rémy, 1999) via objects or records polymorphic fields or using a module-level wrapper. For instance, the following code fails to typecheck because types `int` and `bool` are incompatible.

```
let app (f : (module Type with type t = 'a) → 'a → 'a) =
     (f (module Int) 3, f (module Bool) true)

Error: This expression has type "(module Type with type t = Bool.t)"
       but an expression was expected of type "(module Type with type t = Int.t)"
       Type "Bool.t" = "bool" is not compatible with type "Int.t" = "int"
```

We may enforce polymorphism with an object wrapper:

```
let app (f : <m : 'a. (module Type with type t = 'a) → 'a → 'a >) =
     (f#m (module Int) 3, f#m (module Bool) true)
```

or a module wrapper:

```
module type Fwrapper = sig val f : (module Type with type t = 'a) → 'a → 'a end
let app (module F : Fwrapper) = (F.f (module Int) 3, F.f (module Bool) true)
```

Both solution typechecks because we only loose polymorphism when accessing inside the wrapper, which is here done at each call-site.

In fact, this encoding of abstract types with universal types, which may be a work around in some cases, is no longer possible when t is a higher-kinded abstract type, such as List.t, since OCaml core-level type variables cannot be of higher-rank. For example, there is no way to fix the example below as we did before, as the `(type a)` construct cannot introduce a higher-kinded type.

```
module type Type1 = sig type 'a t end

let id (module T : Type1) (x : 'a T.t) = x

Error: This pattern matches values of type 'a T.t
       but a pattern was expected which matches values of type 'b
       The type constructor T.t would escape its scope
```

1.3 Module-dependent functions as first-class functors

An alternative solution, known by White, Bour, and Yallop (2014), is to view and encode $\lambda(\textbf{module}\ \mathcal{X} : S).\, a$ altogether as a first-class functor

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

whose signature S', equal to $\mathcal{X}.t \to \textbf{sig val}\ \mathit{value} : \tau[\mathcal{X}.t]\ \textbf{end}$, should be *named* and *explicitly* given[4] to build the first-class value or when passing it to another function. For our example of polymorphic identity over a parametrized type `Type1`, this gives:

```
module type ID = functor (A : Type1) → sig val v : 'b A.t → 'b A.t end
module Id : ID = functor (A : Type1) → struct let v (x : 'b A.t) = x end
let id = (module Id : ID)
```

This encoding can cope with higher-order abstract types and seems to cover all use cases.

However, it requires quite a few module-type manipulations, which 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.

[4] 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.

```
module type S = sig
  type _ t
  val f : 'a → 'a t
end

module List = struct
  type 'a t = 'a list
  let f x : _ t = [x]
end
```

Example 2: Shared definitions

```
let hof x = fun (module M : S) → M.f x
:  'a -> (module M : S) -> 'a M.t

let hoapp = hof 3 (module List)
```

Example 3: With a module dependent function

```
module type HOF = sig
  type t
  module F (M : S) :
    sig val value : t M.t end
end

let hof (type a) (x : a) =
  let module Hof = struct
    type t = a
    module F (M : S) =
      struct let value = M.f x end
  end in
  (module Hof : HOF with type t = a)
:  'a -> (module HOF with type t = 'a)

let hoapp =
  let module Hof = (val (hof 3)) in
  let module H = Hof.F(List) in H.value
```

Example 4: With a first-class functor

Figure 1: Comparing module-dependent functions and first-class functors

We give a more complete example in Figure 1 to simultaneously show how the encoding works with higher-order abstract types and how it quickly becomes unpractical at large scale. Comparing the one-line definition of `hof` in example 3 and the dozen lines of its corresponding implementation on example 4 speaks by itself. By comparison, the conciseness and clarity of the direct use of module-dependent functions is striking.

1.4 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 (**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-depend arrow (**module** $\mathcal{X} : S) \to \tau$ and the usual arrow whose domain is a module type (**module** $S) \to \tau$ is syntactically small, but technically significant: $\mathcal{X} : S$ acts as a binder for $\mathcal{X}$ with a scope limited to the codomain type τ. Hence, (**module** $\mathcal{X} : S) \to \tau$ behaves as a first-class polymorphic type[5] while (**module** $S) \to \tau$ behaves as a simple type.

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

The typing of modular explicits is sketched in §3 for an idealized subset of OCaml that we call SmallOCaml . As a result of the overlapping of the two kinds of arrows, there is also an overlapping of the typing rules, which is actually the main typechecking issue.

The key property is that when several typing rules applies, they always agree. In fact, we should state it in contraposed form: the typing rules have been designed to prevent the

[5]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 F^ω by (Blaudeau, Rémy, and Radanne, 2024) and the specific positions of quantifiers in this interpretation.

overlapping when the rule would not agree, which crucially relies on the fact that the domain of module-dependent arrows should always be *known* for the type to be treated as a module-dependent arrow, which we have so far left informal.

Indeed, to formalize this concept, we must reveal a detail that we have temporarily hidden 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 from types that are not yet known and should always be treated as non-dependent arrows. Node variables are of a special kind and incompatible with usual type variables. This mechanism is also necessary to ensure that type inference does not take a decision depending on the order in which typing and unification constraints are solved, and thus to ensure principal types. Interestingly, the way we distinguish *known* from *guessed* types of module-dependent-functions is quite close to a recent proposal for adding *semi-explicit polymorphic parameters* by White (2023), which itself is similar to the mechanism for typing semi-implicit first-class polymorphism in OCaml introduced by Garrigue and Rémy (1999). More details will be given in §3.2.

1.5 Compatibility with first-class modules

Module-dependent functions are only typing artifacts: their runtime representation is the same as 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 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.

Still, module-dependent functions can only accept immediate module expressions as arguments, whose values can therefore not be chosen dynamically (see §3.2). Indeed, immediate module expressions cannot be stored in data structures, the result of a conditional expression, or more generally the result of a computation and first-class modules must be used instead in such cases.

2 Motivating examples

This section presents several programming patterns enabled by modular explicits, namely type classes §2.1, emulation of higher-rank §2.2 and higher-kinded §2.3 types, encoding of F^{ω} §2.5, and fine-grained polymorphism §2.6.

2.1 Type classes with explicit applications

We briefly illustrate the use of module-dependent functions for the encoding of type classes, i.e., what they have first been designed for, but using explicit module abstractions and applications. We only give a simple, textbook example. Advanced examples can be found in the original paper on modular implicits by White, Bour, and Yallop (2014) and a recent, more systematic exploration of their potential by Reader and Vlasits (2024); Reader et al. (2024).

We start by defining the *class* of types whose inhabitants can be compared for equality, along with a generic `equal` function (sometimes also called a *method*) which when given an instance `Eq` and two values of the supporting type, will check whether they are equal or not.

```
module type Eq = sig type t val equal : t → t → bool end

let equal (module X : Eq) x y = X.equal x y
```

We may now define two instances of `Eq`, one for type `int` and one for type `bool`.

```
module Eq_int = struct type t = int let equal x y = Int.equal x y end

module Eq_bool = struct
  type t = bool
  let equal x y = Bool.equal x y
end
```

(We could also have taken the `Int` and `Bool` modules of the standard library directly, which are already instances of `Eq`, instead of defining `Eq_int` and `Eq_bool`.) We may then call the `equal` method as in the following example:

```
let _ = equal (module Eq_int) 1 2 || equal (module Eq_bool) true true
```

Given an instance `X` of the `Eq` class, we may also construct an instance of `Eq` for lists of elements of type `X.t` using the generic `equal` function defined above, i.e., by calling the equal method of the `Eq` class:

```
module Eq_list (X : Eq) = struct
  module rec FX : Eq with type t = X.t list = struct
    type t = X.t list
    let equal (x : t) y  = match x, y with
      | [], [] → true
      | hx::tx, hy::ty → equal (module X) hx hy && equal (module FX) tx ty
      | _ → false
  end
  include FX
end

let _ = equal (module Eq_list(Eq_int)) [1] [1;2]
```

This is more involved than the Haskell version, where the instance is implicitly recursive. Here. we must be explicit about recursion (which is by default in Haskell), and we need to use recursive modules. *Inheritance* is implemented by embedding a submodule of the inherited type:

```
module type Ord = sig
  type t
  module Eq : Eq with type t = t
  val lt : t → t → bool
end

let lt (module X : Ord) x y = X.lt x y
```

When building instances, we simply define the submodule as an alias for the inherited class:

```
module Ord_int = struct
  type t = int
  module Eq = Eq_int
  let lt x y = x - y < 0
end
```

The reason for *embedding* an alias instead of just *including* the parent is due to *coherence* considerations in the context of *modular implicits* and is explained by White et al. (2014). While these considerations do not apply in the context of *modular explicits*, we chose to remain consistent with their encoding.

Finally, we can use `Ord` and the induced equality, as illustrated in the following example. The full type annotation on `search` is needed as it is a recursive definition:

```
let rec (search : (module X : Ord) → X.t → X.t list → X.t option) =
  fun (module X) x → function
  | [] → None
  | h::_ when equal (module X.Eq) x h → Some x
  | _::t → search (module X) x t

let _ = search (module Ord_int) 2 [1; 2; 3]
```

2.2 Higher-rank types

In the examples of type classes, module abstractions are primarily used to abstract over dictionaries. Typically, dictionaries are modules with a distinguished type component t representing the type at which the dictionary is implemented and a set of operations provided by the class at this type. (We also use functors building dictionaries from other subdictionaries.)

There is a surprisingly interesting subcase of module-dependent functions where *module-dependent* abstractions and applications are only used to emulate *type* abstractions and applications.

We can already use abstract types to enforce polymorphism. For instance, let us define a biased function that choose between two values of any given type:

```
let bias (type a) (x : a) (y : a) = if Random.int 3 > 0 then x else y
```

```
val bias : 'a → 'a → 'a = <fun>
```

This ensures that the function works for any type `a` and not just for some instance of `a`. Still, polymorphism is left implicit as if the type polymorphism had been inferred: Reusing the previous signature `Type` of modules just carrying a type field `t`, we may redefine `bias` as follows:

```
let bias (module A:Type) (x : A.t) (y : A.t) = if Random.bool() then x else y
```

```
val bias : (module A : Type) → A.t → A.t → A.t = <fun>
```

In particular, using `bias` at some particular type now requires the expected type as an explicit module argument.

```
module Tint = struct type t = int end
let my_int = bias (module Tint) 7 42
```

As a counterpart, we may now receive `either` as a polymorphic function as argument, for example to choose between heterogeneous pairs:

```
type choice =  (module A:Type) → A.t → A.t → A.t

let choose_pair (choose : choice)
    (module A:Type) (module B:Type) (x1, x2) (y1, y2) =
  choose (module A) x1 y1, choose (module B) x2 y2

module Tbool = struct type t = bool end
```

```
let choose_int_bool_pair choose = choose_pair choose (module Tint) (module Tbool)
```
```
val choose_int_bool_pair :
  choice → Tint.t * Tbool.t → Tint.t * Tbool.t → Tint.t * Tbool.t = <fun>
```

Notice that the definitions of modules `Tint` and `Tbool` cannot currently be inlined, even though they are just used to parameterize modules by types.

However, a recent proposal, coined *functor type arguments* by Vivien (2024b), allows type arguments to be used directly as parameters of functors and appear in paths. With this extension, we can then define a module wrapper:

```
module T(type a) = struct type t = a end
```

Then, `Tint` could be defined as the application `T(type int)`, and be inlined as a path. We can then write without prior definition but that of `T`:

```
let choose_int_bool_pair choose = choose_pair choose (module T(type int))
(module T(type bool))
```

See §6 for other extensions, independent of modular explicits, but useful for modular explicits.

While this example only uses rank-2 polymorphism, the method extends to more complex types at higher ranks. We illustrate this on the well-known encoding of the existential type $\exists\alpha.\,\tau$ as the polymorphic type $\forall\alpha.\,(\forall\beta.\,\tau \to \beta) \to \alpha$, but for a particular type τ. For instance, consider delaying a computation by pairing a function with its argument:

```
type ('a, 'b) delay = ('a → 'b) * 'a
```

Ideally, we wish to hide `'a` in `('a, 'b) delay` if we are just delaying the computation to eventually perform the application. That is, we wish to give it the existential type $\exists$ `'a. ('a, 'b) delay`, say `'b frozen` defined as:

```
type 'a frozen =
  (module A:Type) → ((module B:Type) → (B.t, 'a) delay → A.t) → A.t

let freeze (type a) (type b) (f : a → b) (x : a) : b frozen =
  fun (module A:Type) (body : (module B:Type) → (B.t, b) delay → A.t) →
    body (module T(type a)) (f, x)
```
```
val freeze : ('a → 'b) → 'a → 'b frozen = <fun>
```

Having hidden the parameter of the function, we may now build lists of heterogeneous frozen computations:

```
let two = [freeze ((=) 1)  0; freeze not false ]
```

The converse function is

```
let unfreeze (type b) (fx : b frozen) =
  fx (module (T(type b))) (fun (module A:Type) (f, x) → f x)
```
```
val unfreeze : 'b frozen → 'b = <fun>
```

We may then get the list of evaluated results as:

```
let two_defrost = List.map unfreeze two
```

Notice that we could also have defined

```
let unfreeze' (module B:Type) (fx : B.t frozen) =
  fx (module B) (fun (module A:Type) (f, x) → f x)
```

```
val unfreeze' : (module B:Type) → B.t frozen → B.t
```

remaining in the explicit encoding of polymorphism. This is even a bit shorter, but this now requires an explicit type argument when applied, as indicated by the inferred type.

In fact, there is inter-convertibility between the two variants with implicit and explicit polymorphism as shown below:

```
let _unfreeze' (module B : Type)  fx = unfreeze fx
let _unfreeze (type a) fx = unfreeze' (module T(type a))
```

Unfortunately, this only works for toplevel polymorphism, and only explicit polymorphism can be used for inner polymorphism.

2.3 Higher-kinded types

We have emulated first-class polymorphism with dependent functions by abstracting over modules of signature `Type`. We may as well abstract over modules whose type definition is a type operator.

```
module type Monad = sig
  type 'a m
  val bind : 'a m → ('a → 'b m) → 'b m
  val return : 'a → 'a m
end
```

Then, we may write functions such as:

```
let bind (module M : Monad) = M.bind
```

```
val bind : (module M : Monad) → 'a M.m → ('a → 'b M.m) → 'b M.m = <fun>
```

That is, functions using higher-kinded polymorphism.

Alternative representation of type operators

We may also encode type operators as functors taking a module of signature `Type` as argument:

```
module type Op = functor (A:Type) → Type
let twice (module F : Op) (f : (module A:Type) → F(A).t → A.t)
          (module A:Type) (x : F(F(A)).t) =
  f (module A) (f (module F(A)) x)
```

```
val twice :
  (module F : Op) → ((module A:Type) → A.t → F(A).t) →
    (module A:Type) → A.t → F(F(A)).t
```

```
module Tlist (A:Type) = T(type A.t list)
let hd_of_hd =
  let hd (module A:Type) : Tlist(A).t → A.t = List.hd in
  twice (module Tlist) hd
```

```
val hd_of_hd : (module A : Type) → Tlist(Tlist(A)).t → A.t = <fun>
```

We may turn explicit polymorphism into implicit polymorphism as seen earlier:

```
let hd_of_hd_ (type a) x =
  let module A = struct type t = a end in hd_of_hd (module A) x
```

```
val hd_of_hd_ : 'a list list → 'a = <fun>
```

The advantage of using functor abstraction to provide type operators is that the approach is more regular and transparently extend to higher kinds, e.g., operators taking operators as arguments.

Limitation While modular explicits brings type abstraction at higher kinds in OCaml, the encoding of types of higher kinds only exits at the module-level. In particular, we still do not have core-level kinds and cannot have parameters of a *type definition* of higher kinds. An independent extension of OCaml with primitive higher kinds would also be possible and could allow that.

2.4 Mixing higher-kinded types and higher-order types

Since modular explicits allows for both explicit higher kinds and higher-order types, we may freely combine them and so reach the power of F^ω.

To illustrate this, we generalize the encoding of existential types given above on the particular case of frozen computations. Thanks to abstraction over type operations, we may provide a general encoding of existential types into universal types in F^ω, i.e., one that is parameterized by an arbitrary type $\exists\alpha.\tau$, technically by a type operator F representing the type function $\lambda\alpha : \mathsf{Type}.\,\tau$.

Namely, the CPS encoding of the existential type $\exists\alpha.F(\alpha)$ is:

```
module Exists (F : Op) = struct
    type t = (module B:Type) → ((module A:Type) → F(A).t → B.t) → B.t
  end
```

The introduction of such an existential type is the `pack` function:

```
let pack (module F : Op) (module A:Type) (x : F(A).t) : Exists(F).t =
  fun (module B:Type) (body : (module A:Type) → F(A).t → B.t) →
    body (module A) x
```

The elimination or unpacking of an encoded existential value is then just its application to the continuation, which should be of type $\forall\beta.\,(\forall\alpha.\,F(\alpha) \to \beta)$, here encoded as the dependent type $(\textbf{module}\ A : \mathsf{Type}) \to F(A).t \to B.t$:

```
let unpack (module F : Op) (v : Exists(F).t) (module B:Type)
        (k : (module A:Type) → F(A).t → B.t) = (v (module B) k)
```

We may then recover the first-order implementation of frozen computations of §2.2 just using our higher-order `pack` and `unpack` functions:

```
module Frozen (B:Type) (A:Type) = struct type t = (A.t, B.t) delay end

let freeze (type a) (type b) (f : a → b) (x : a) =
  pack (module Frozen(T(type b))) (module T(type a)) (f, x)
```

```
let unfreeze (type a) x =
  unpack (module Frozen(T(type a))) x (module T(type a))
    (fun (module A:Type) (f, x) → f x)
```

The function `freeze` and `unfreeze` use explicit polymorphism:

```
val freeze :
  ('a → 'b) → 'a →
    (module B:Type) → ((module A:Type) → (A.t, 'b) delay → B.t) → B.t
val unfreeze :
  ((module B:Type) → ((module A:Type) → (A.t, 'a) delay → B.t) → B.t) → 'a
```

One would perhaps prefer to see the following signatures:

```
val freeze : ('a → 'b) → 'a → Exists(Frozen(T(type 'b))).t = freeze
val unfreeze : Exists(Frozen(T(type 'b))).t → 'b = unfreeze
```

Unfortunately, the free variable `'b` is not *currently* allowed in path `(type 'b)`, hence the signatures need to be expanded.

2.5 Light-weight encoding of F^ω

The illustrative encoding of existential types into modular explicits actually works for the full system F^ω, which we demonstrate in this section. The definition of F^ω is left implicit. We write e, σ and κ for terms, types, and types of F^ω, so as to distinguish them from expression a and types τ of the core language and M and S of the signature language. However, we reuse type variables x and α of the core language for F^ω, which are mapped to expression variables and module variables in the translation, respectively.

More precisely, the translation of terms of F^ω into modular explicits is defined in Figure 2. You may ignore the gray lines at first. The encoding uses the functor T that takes a type as argument as a direct parameter, as in the example above.

The target of the encoding, modular explicits, is stratified between core-level expressions and module-level expressions, but module-level expressions are only used to encode types, using functors but only to encode higher-order types. Therefore, functors and their applications could be fully erased during compilation (assuming that polymorphism is restricted to values) as no runtime value depends on them.

In particular, a source expression e of F^ω of type σ (which is itself always of the base kind $\star$) is translated to an expression of core-level type $[\![\sigma]\!]^\flat$. More precisely, a variable, a function, or an application of type τ is translated to a core-level expression; by contrast, a type abstraction $\Lambda\alpha : \kappa.\, e$ or a type application $e\ \tau$ is translated to a dependent-abstraction $\lambda\alpha : [\![\kappa]\!].\, [\![e]\!]$ or a dependent application $[\![e]\!]$ (**module** $[\![\tau]\!]$) with the same computational content as $[\![e]\!]$—since modules used in the encoding never manipulate anything else but type components. Notice that $[\![\sigma]\!]$ is a module-level expression of signature $[\![\kappa]\!]$.

The translation of types is slightly more involved because of the stratification of modular explicits: a type σ of kind κ in F^ω can be translated to a module-level expression $[\![\sigma]\!]$ of kind $[\![\kappa]\!]$ and also to a core-level type $[\![\sigma]\!]^\flat$—but only when κ is the kind $\star$. Besides, there are coercions between the two levels: we may project a module expression M of signature Type into a core-level type $M.t$ and, conversely, a core-level type $[\![\sigma]\!]^\flat$ may need to be lifted to a module-level type $T($**type** $[\![\sigma]\!]^\flat)$ of signature Type. We slightly abuse the notation in Figure 2 where the lines $[\![\sigma : \star]\!]$ and $[\![\sigma : \star]\!]^\flat$ are default cases that should only be used when σ is of kind $\star$ and the preceding cases do not apply. This should avoid building types expressions of the form $T($**type** $[\![\sigma]\!].t)$ or $T($**type** $[\![\sigma]\!]^\flat).t.$ which would be correct but could be reduced to $[\![\sigma]\!]$ and $[\![\sigma]\!]^\flat$ respectively.

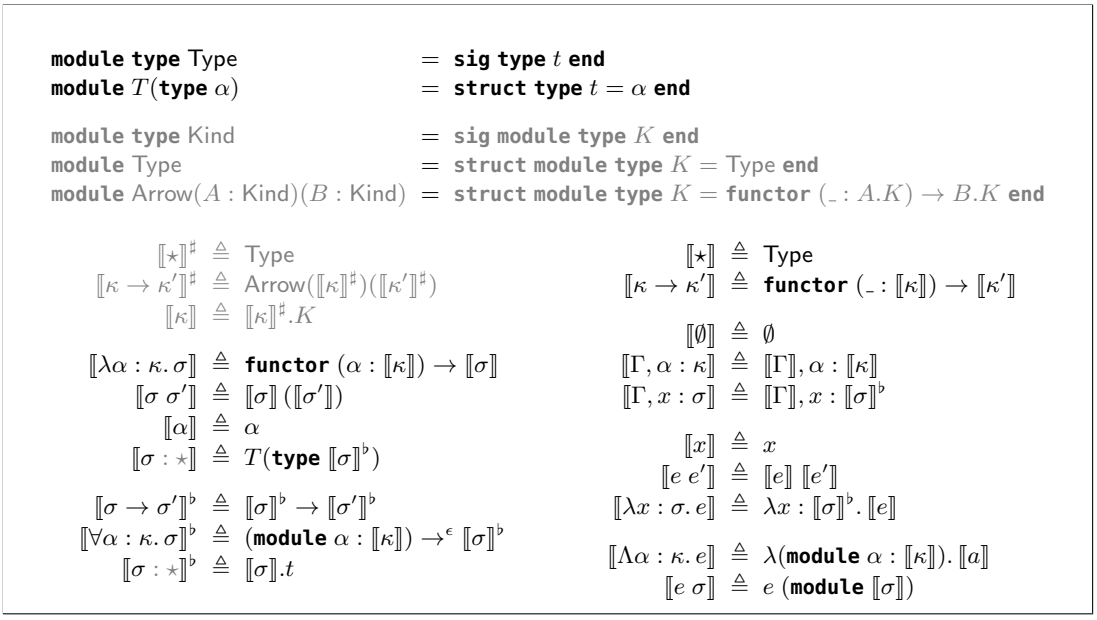

Figure 2: Encoding of F^ω with modular explicits.

Syntactic restrictions When ignoring the gray, the translation is slightly incorrect as it encode kinds with signature that are not named paths. Indeed $[\![\kappa \to \kappa']\!]$ is a functor expression. As a result, the encoding requires an extension of the current proposal as discussed in §6.2 or some post-processing pass to extrude those definitions using **let module** $\mathcal{X} = M$ **in** e expressions. In fact, with such an extension, we could as well have inlined $T(\textbf{type}\ [\![\sigma]\!]^\flat)$ as **struct type** $t = [\![\sigma]\!]$ **end** in $[\![\sigma]\!]$.

In practice, we just use a few higher-order kinds, the most common being $\star \to \star$, which could then be predefined as we did for `Op` in §2.4.

Interestingly, there is a slight modification of the encoding that fixes the problem. This is the gray code. Instead of translating a kind directly to its signature $[\![\kappa]\!]$, we lift it to a module $[\![\kappa]\!]^\flat$ containing a module type definition equal to **module type** $K = [\![\kappa]\!]$. This way we can use a functor Arrow to build the lifted signature of $[\![\kappa \to \kappa']\!]^\flat$ from $[\![\kappa]\!]^\flat$ and $[\![\kappa']\!]^\flat$. Then we recover $[\![\kappa]\!]$ as the projection $[\![\tau]\!]^\flat.K$. The rest of the encoding is unchanged.

For example, the signature `Op` of §2.4 could have be defined as $\mathsf{Arrow}(\mathsf{Type})(\mathsf{Type}).K$, which is a path, and therefore could have been inlined.

Core-level polymorphism So far, the target of the translation of F^ω does not use core-level polymorphism. However, we may easily enrich the source language to do so by distinguishing two subsets of variables α and β where β would be restricted to type abstraction of let-bound expressions and be translated into implicit core-level type polymorphism, as follows:

$$e ::= \ldots \mid \textbf{let}\ x = \Lambda\bar{\beta}.\, e\ \textbf{in}\ e \mid e\ \tau$$

$$[\![\textbf{let}\ x = \Lambda\bar{\beta}.\, e\ \textbf{in}\ e']\!] \triangleq \textbf{let}\ x = \Lambda(\textbf{type}\ \bar{\beta}).\, [\![e]\!]\ \textbf{in}\ [\![e']\!] \qquad [\![e\ \tau]\!] \triangleq [\![e]\!] \qquad [\![\beta]\!]^\flat \triangleq \beta$$

Core-level type constants and type operators We may also add base types such as **int** and base type operators such as **list** _, with the following translation:

$$[\![\mathbf{int}]\!]^\flat = \mathbf{int} \qquad\qquad [\![\mathbf{list}\ \tau]\!]^\flat = \mathbf{list}\ [\![\tau]\!]^\flat$$

Then, $[\![\lambda\alpha : \star.\mathbf{list}\ \alpha]\!]$ would be translated to **functor** $(\alpha : \mathsf{Type}) \to T(\mathbf{type}\ \mathbf{list}\ \mathbf{int})$ which we may bind to some module **List**. If we similarly bind $[\![\mathbf{int}]\!]$ to **Int**, we could then translate $[\![\mathbf{list}\ \mathbf{int}]\!]$ either to $\mathbf{List}([\![\mathbf{int}]\!])$ or directly to $T(\mathbf{list}\ \mathbf{int})$, as both forms are equivalent.

Soundness of the encoding The encoding preserves convertibility: that is, $\sigma \equiv \sigma'$ implies $[\![\sigma]\!] \equiv [\![\sigma']\!]$. It also preserves typability. If $\Gamma \vdash a : \sigma$, then $[\![\Gamma]\!] \vdash [\![a]\!] : [\![\sigma]\!]$.

Beyond $\mathbf{F}^\omega$ The encoding we have shown here only uses modules for carrying type information. Of course, modules may simultaneously carry values, as in most examples with modules. Modules may also carry module type definitions.

2.6 Fine-grain polymorphism

So far, we have looked at examples where modular explicits allow to explicitly quantify over some type, and over some types equipped with specific operations. But we can also use them to quantify over types that match specific properties. For instance types whose values are known to be represented as `int`'s at runtime:

```
module type Immediate = sig type t [immediate] end
let f (module I:Immediate) (i : I.t) = ((* something necessarily dirty *))
```

This is yet another (heavy) way to attach information to the type of parameters than the one proposed in the OCaml pull request *Allow explicit binders for type variables* #10437, but one that scales to higher-kinded types. For example, quantitation over injective types can be done as follows:

```
type (_, _) eq = Refl : ('a, 'a) eq
module type Injective = sig type !'a t end
let strip (module A : Injective) (type a) (type b) =
   fun (Refl : (a A.t, b A.t) eq) → (Refl : (a, b) eq)
```

3 Formal presentation

We formalize modular explicits in a small subset of OCaml called SmallOCaml , focusing on the typing of core-language expressions, including first-class modules, but taking typing of modules for granted.

3.1 SmallOCaml

The syntax of (a mini subset of) OCaml is given in Figure 3. The only extension to first-class module types is the introduction of a new form of arrow type $(\mathbf{module}\ X : Q) \to^\epsilon \tau$, the module-dependent arrow type and functions to construct and use them.

Syntactic productions in gray correspond to the extension to explicit functor type arguments and can be ignore at first. There are discussed in at the end of this section and are not part of the typing rules for sake of conciseness (they do not raise technical issues).

$$
\begin{array}{rcll}
P & ::= & \mathcal{X} \mid P.\mathcal{X} \mid P(P) \mid P(\texttt{type}\ \pi) & \text{Path} \\
\pi & ::= & t \mid P.t & \\
\tau & ::= & \alpha \mid \rho \mid \tau \to^{\epsilon} \tau \mid (\texttt{module}\ \mathcal{X} : Q) \to^{\epsilon} \tau \mid \ldots & \\
 & \mid & (\texttt{module}\ (\texttt{type}\ t)) \to^{\epsilon} \tau & \text{Types} \\
\gamma & ::= & \alpha \mid \epsilon & \text{Type \& node variables} \\
\sigma & ::= & \forall \bar{\gamma}.\, \tau & \text{Type schemes} \\
Q & ::= & T \mid P.T \mid Q\ \texttt{with type}\ t = \tau & \text{Named signatures} \\
S & ::= & Q \mid \texttt{sig}\ \bar{D}\ \texttt{end} \mid \texttt{functor}\ (\mathcal{X} : T) \to S & \\
 & \mid & \texttt{functor}\ (\texttt{type}\ t) \to S & \text{Signature} \\
D & ::= & \texttt{val}\ \ell : \sigma \mid \texttt{type}\ t \mid \texttt{type}\ t = \tau & \\
 & \mid & \texttt{module}\ \mathcal{X} : S \mid \texttt{module type}\ T = S & \text{Specification} \\
a & ::= & x \mid P.\ell \mid \lambda x.\, a \mid a\ a \mid \texttt{let}\ x = a\ \texttt{in}\ a \mid \lambda(x : \tau).\, a & \text{Core expressions} \\
 & \mid & \texttt{let module}\ \mathcal{X} = M\ \texttt{in}\ a \mid (\texttt{module}\ M : Q) & \text{First-class modules} \\
 & \mid & \lambda(\texttt{module}\ \mathcal{X} : Q).\, a \mid a\ (\texttt{module}\ N) & \\
 & \mid & \lambda(\texttt{module}\ (\texttt{type}\ t)).\, a & \text{Modular explicits} \\
M & ::= & P \mid \texttt{struct}\ \bar{d}\ \texttt{end} \mid (M : S) & \\
 & \mid & M(N) \mid \texttt{functor}\ (\mathcal{X} : S) \to M & \\
 & \mid & \texttt{functor}\ (\texttt{type}\ t) \to M & \text{Module} \\
N & ::= & M \mid (\texttt{type}\ \pi) & \\
d & ::= & \texttt{val}\ \ell = a \mid \texttt{type}\ t \mid \texttt{type}\ t = \tau & \\
 & \mid & \texttt{module}\ \mathcal{X} = M \mid \texttt{module type}\ T = S & \text{Declaration}
\end{array}
$$

Figure 3: Syntax of SmallOCaml

Identifiers Still, as SmallOCaml include a presentation of modules, it much larger than core OCaml. We distinguish identifiers $\mathcal{X}$ for modules, t for types, T for module types x for expressions. All of them can also be prefixed by a path P, which is a module name, an application of a path to a path, or a projection of a path on a module name.

Node variables Both module-dependent and usual arrow types carry a node variable ϵ that helps keep track of sharing and ensure the existence of principal types. The annotations have been grayed so that they may easily be ignored at first glance, as in fact OCaml does by default (hence breaking the principal type property). Node variables are new to modular explicits: they were first introduced in OCaml for semi-explicit polymorphism by Garrigue and Rémy (1999) and are already used in OCaml to pass immediate first-class modules as arguments without the need for a signature annotation.

Node variables and type variables are of different kinds and cannot be mixed. Node variables may only be substituted by other node variables and type variables by types. Still, they often behave the same and we use γ to range over both of them. Type schemes are types that may be polymorphic in both kinds of variables.

Node variables matter in inferred types, but do not often matter in source terms, as the typing rule will rename them. We may thus omit then, in which case the corresponding nodes should be understood as having fresh node variables.

Named signatures A named signature Q is a particular form of signature. It may be just a signature name T, a path projection on a name T, or a constrained signature Q **with type** $t = \tau$. Named signatures are used as types of first-class modules. This is just a syntactic restriction enforced by OCaml for simplicity and efficiency of the implementation and conciseness of printed types, which may mention paths but not signatures. Still, the **with type** $t = \tau$ construct allows named signatures to contain core-language types at their leaves, including explicitly bound type variables, e.g.,. by (**type** a), but not flexible type variables such as α. Named signatures are used for all core-level expressions, i.e., module-dependent arrows, module-dependent functions, and first-class modules, but not for module-level expressions.

Arrow types We remind that module variable $\mathcal{X}$ acts as a binding in (**module** $\mathcal{X} : Q$) $\to^\epsilon \tau$ whose scope extends to τ and hence may appear in τ. There is an overlap between the two kinds of arrows (**module** $\mathcal{X} : Q$) $\to^\epsilon \tau$ and (**module** Q) $\to^\epsilon \tau$ when $\mathcal{X}$ does not actually appear in τ, in which case they actually mean the same thing—and the implementation, which treats them as two different type constructors, will freely convert the former into the latter in such cases. Here, (**module** Q) $\to^\epsilon \tau$ is really a usual arrow $\tau_0 \to^\epsilon \tau$ where τ_0 happens to be a first-class module type (**module** Q).

We distinguish regular functions $\lambda x.\, a$, which expect an expression as a parameter, from explicit functions λ(**module** $\mathcal{X} : Q$)$.\, a$, which expect a module as an argument, whose signature Q must have been explicitly given.

Similarly, we distinguish the regular application $a_1\ a_2$ from the application to an immediate module expression a_1 (**module** M). While we may also turn a module into a first-class module expression (**module** $M : Q$), which can then be stored in a data-structure or passed to functions, the immediate applications a (**module** M) does not require an explicit signature Q for M when the type of a is known.

Explicit functor type arguments We come back to the extension of modular explicits with explicit functor type arguments that we have used in the examples. This extensions allows the

$$\textsc{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{App}\ \frac{\Gamma \vdash a_1 : \tau_2 \to^{\epsilon} \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1} \qquad \textsc{Fun}\ \frac{\Gamma, x : \tau' \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau' \to \tau} \qquad \textsc{Gen}\ \frac{\Gamma \vdash M : \sigma \qquad \gamma \# \Gamma}{\Gamma \vdash M : \forall \gamma.\, \sigma}$$

$$\textsc{InstT}\ \frac{\Gamma \vdash M : \forall \alpha.\, \sigma}{\Gamma \vdash M : \sigma[\alpha \leftarrow \tau]} \qquad \textsc{InstN}\ \frac{\Gamma \vdash x : \forall \epsilon.\, \sigma}{\Gamma \vdash x : \sigma[\epsilon \leftarrow \epsilon']} \qquad \textsc{FunA}\ \frac{\Gamma, x : \forall \bar{\epsilon}.\, \tau_2 \vdash a : \tau' \qquad \tau \lessapprox \tau_1 \Rightarrow \forall \bar{\epsilon}.\, \tau_2}{\Gamma \vdash \lambda(x : \tau).\, a : \tau_1 \to^{\epsilon} \tau'}$$

$$\textsc{Mod}\ \frac{\Gamma \vdash M : Q}{\Gamma \vdash (\textbf{module}\ M : Q) : \forall \bar{\epsilon}.\, (\textbf{module}\ Q)} \qquad \textsc{AppM}\ \frac{\Gamma \vdash a_1 : \forall \epsilon.\, (\textbf{module}\ Q) \to^{\epsilon} \tau \qquad \Gamma \vdash M : Q}{\Gamma \vdash a_1\ (\textbf{module}\ M) : \tau}$$

$$\textsc{AppD}\ \frac{\Gamma \vdash a_1 : \forall \epsilon.\, (\textbf{module}\ \mathcal{X} : Q) \to^{\epsilon} \tau \qquad \Gamma \vdash M : Q \qquad \Gamma \vdash M \rightsquigarrow P}{\Gamma \vdash a_1\ (\textbf{module}\ M) : \tau[\mathcal{X} \leftarrow P]}$$

$$\textsc{FunD}\ \frac{\Gamma, \mathcal{X} : \forall \bar{\epsilon}.\, Q_2 \vdash a : \tau \qquad Q \lessapprox Q_1 \Rightarrow \forall \bar{\epsilon}.\, Q_2}{\Gamma \vdash \lambda(\textbf{module}\ \mathcal{X} : Q).\, a : (\textbf{module}\ \mathcal{X} : Q_1) \to^{\epsilon} \tau}$$

Figure 4: Typing rules for functions and applications (simplified)

grayed syntactic constructions. In particular, module arguments N may now be an inlined type argument (**type** π). However, for the moment π is still reduced to a named type as defined in Figure 3. However, it could in principle be any type τ, modulo the option to allow or not free type variables in module signatures (see the discussion in §6). The extension does not raise technical difficulty. Hence, we ignore it for sake of simplicity in the rest of this section.

3.2 Typing rules

An excerpt of typing rules for expressions is given on Figure 4. The judgment $\Gamma \vdash M : Q$ for typing modules is not affected by this extension and is omitted. For people with some knowledge of OCaml, it should be enough to take it intuitively. Otherwise, we refer the reader to Leroy (1995).

There are actually just two new rules FunD and AppD for typing module-dependent functions and their application to immediate module arguments. In particular, rules FunA, AppM and Mod for typing functions receiving non-dependent module arguments and their applications to immediate module arguments are already part of OCaml with first-class modules. However, we will explain them simultaneously as they interact.

Functions with explicitly typed arguments Let us start with rule FunA. At first, one could take τ_1 and $\forall \bar{\epsilon}.\, \tau_2$ to be equal to τ, which would look less surprising, as the relation $\tau \lessapprox \tau \Rightarrow \tau$ always hold. However, this would require the type annotation to be equal to the type of the argument and the type of x could not be considered as known while typing a.

The use of relation $\tau \lessapprox \tau_1 \Rightarrow \forall \bar{\epsilon}.\, \tau_2$ allows more freedom. Formally, the relation $\tau \lessapprox \tau_1 \Rightarrow \forall \bar{\epsilon}.\, \tau_2$ is composed of all triples of the form

$$\eta(\tau) \lessapprox \theta(\tau) \Rightarrow \theta(\forall \bar{\epsilon}.\, \tau)$$

where η may only substitutes node variables, while θ may substitute both node and type variables. That is, starting with τ_0 we may first take a term τ that is equal to τ_0 except on its node variables that are decorrelated (i.e., $\tau_0 = \eta\tau$ for some η); we then pick $\bar{\epsilon}$ to be any subset of the node variables of τ' (taking the whole set is better) and any substitution θ (of both type and node variables) to return $\tau_0 \lessapprox \theta(\tau) \Rightarrow \theta(\forall\bar{\epsilon}.\,\tau)$.

The order of operations matters as the substitution θ may introduce new node variables that will then be the same on both τ_1 and τ_2. The decorrelation of node variables mean that their values in the annotation do not matter (we could have equivalently removed node-variables on source annotations). Their generalization in $\forall\bar{\epsilon}.\,\tau_2$ allows $\bar{\epsilon}$ to be considered as known while typing a. The substitution θ amounts to interpret free variables of the annotation τ as flexible variables that may be guessed, i.e., unified during type inference.

Module-dependent functions Rule FunD is similar to FunA but takes a first-class module as argument. We somehow assume $\mathcal{X}$ of type Q_2, an instance of (a decorrelated copy of) Q while typing the body a with type τ and returns $(\textbf{module}\ \mathcal{X} : Q_1) \to^\epsilon \tau$ where Q_1 is the same instance of another decorrelated copy of Q.

The relation $Q \lessapprox Q_1 \Rightarrow \forall\bar{\epsilon}.\,Q_2$ is just a special case of $\tau \lessapprox \tau_1 \Rightarrow \forall\bar{\epsilon}.\,\tau_2$ when τ is $(\textbf{module}\ Q)$. In fact, since OCaml does not currently allow Q to have free type variables, types Q_1, Q and Q_2 may only differ by the nodes. (This restriction of OCaml is not actually required for the typing of modular explicits, and expressions $Q\ \textbf{with type}\ t = \tau$ could allow free type variables to appear in τ.)

First-class modules Mod turns a module M of type Q into a first-class module of type $\forall\bar{\epsilon}.\,(\textbf{module}\ Q)$. Here, since Q is the type of a module, it cannot contain free type variables but it may still contain node variables that may be freely renamed.

Applications There are three typing rules for applications:

- Rule AppD for module-dependent application can only be used when the type is known $(\epsilon \# \Gamma)$ to be a module-dependent function type $(\textbf{module}\ \mathcal{X} : Q) \to^\epsilon \tau$ and the argument of the application is an immediate first-class module $(\textbf{module}\ M)$ where M is of type Q and can be elaborated to a path P, which we write $\Gamma \vdash M \rightsquigarrow P$.
- Rule AppM for non-dependent application can be used when the type is known $(\epsilon \# \Gamma)$ to be $(\textbf{module}\ Q) \to^\epsilon \tau$ and the argument of the application is an immediate first-class module $(\textbf{module}\ M)$ where M is of type Q.
- Otherwise, we default to a normal application App of an expression a_1 to an expression a_2. Hence a_1 is a non dependent arrow whose domain may be unknown (ϵ need not be polymorphic).

Rule Mod turns a module M into a first-class expression. It requires an explicit signature Q.

Hence, as a particular case of Rule App, a_2 may still be a first-class module $(\textbf{module}\ M : Q)$, but it has be to explicitly annotated with a signature Q, since Q is not known from context (otherwise, rule AppM should have been used).

$$\frac{\Gamma \vdash a_1 : (\textbf{module}\ Q) \to^\epsilon \tau_1 \qquad \dfrac{\Gamma \vdash M : Q}{\Gamma \vdash (\textbf{module}\ M : Q) : (\textbf{module}\ Q)}\ \textsc{Mod}}{\Gamma \vdash a_1\ (\textbf{module}\ M : Q)}\ \textsc{App}$$

That is, even though the domain of the type a $a_1\ \tau_2$ happens to be $(\textbf{module}\ Q)$, the specification Rule App does not allow the use of such information to elude the annotation on the argument of

a_1, thus forcing an explicit annotation. This has the advantage to let type inference proceed in any order, since even if we learn information by guessing earlier, it will not bring any advantage, so that the same guessing could have happen later.

Due to the compatibility between $(\textbf{module}\ \mathcal{X} : Q) \to^\epsilon \tau$ and $(\textbf{module}\ Q) \to \tau$ when $\mathcal{X}$ does not appear free in τ, rules AppD and AppM are not exclusive. However, being able to apply both means that $\mathcal{X}$ does not appear free in τ. Thus, both types $\tau[\mathcal{X} \leftarrow P]$ an τ are equal: whichever rule is used we obtain the same resulting type. Hence, this overlapping of rules does not break principal types.

3.3 Tracking the unknowns

Typing rules AppD and AppM for applications both require the type of a to be *known*, which is ensured by the universal quantifier $\forall\epsilon$. Equivalently, we could have written $\Gamma \vdash a_1 : (\textbf{module}\ \mathcal{X} : Q) \to^\epsilon \tau$ for the first premise and added the side condition $\epsilon \# \Gamma$.

This still allows to guess types, but prevent the use of *guessed* types as *known* types. For instance, an attempt typing $\lambda f.\, a[f\ (\textbf{module}\ M)]$ will fail, as if we guessed that f had the module-dependent type $(\textbf{module}\ \mathcal{X} : S) \to^\epsilon \tau$[6], then ϵ would appear in the typing context preventing from its generalization while typing the expression $f\ (\textbf{module}\ M)$ and rule AppD would not apply.

By contrast, FunD introduces a node variable ϵ in the type $(\textbf{module}\ \mathcal{X} : Q) \to^\epsilon \tau$ of its conclusion, which may be taken fresh for Γ, thus letting the (module-dependent) arrow type be considered as *known*. Typically, in an expression of the form

$$\textbf{let}\ f = \lambda(\textbf{module}\ \mathcal{X} : Q).\, a_0\ \textbf{in}\ a[f\ (\textbf{module}\ M)]$$

variable ϵ will be generalized in the type of f so that the application $f\ (\textbf{module}\ M)$ can be typed without the need for an annotation on M.

A guessed type may also be turned into a known type using a *type annotation* of the form $(a : \sigma)$. This is actually syntactic sugar for $(\tau)\ a$, i.e., the application of the retyping function (τ) to a, where (τ) stands for $\lambda(x : \tau).\, x$, of principal type $\forall\bar{\epsilon}, \bar{\alpha}.\, \tau_1 \to \tau_2$ where τ_1 and τ_2 are both equal to τ except for their node variables that are all distinct. A derived rule would be

$$\frac{\Gamma \vdash a : \tau_1 \qquad \tau \lessapprox \tau_1 \Rightarrow \forall\bar{\epsilon}.\, \tau_2}{\Gamma \vdash (a : \tau) : \tau_2}\ \textsc{Annot}$$

Alternatively, we could have taken type annotations on expressions as primitive with the typing rule Annot and defined $\lambda(x : \tau).\, a$ as syntactic sugar for $\lambda x.\, \textbf{let}\ x = (x : \tau)\ \textbf{in}\ a$.

3.3.1 Principal types

The typing rules have been designed to preserve the principal type property. This is just a conjecture for the moment. A formal proof would require a specification of typing rules for modules (or abstract the result up to appropriate assumptions on the typing of modules). Still, the key for the proof are node-variables, which are used is a slightly different but similar way to semi-explicit polymorphism introduced in OCaml by Garrigue and Rémy (1999).

[6] During type inference, this would amount to substituting the unknown type α of f that appears in the typing context by $(\textbf{module}\ \mathcal{X} : S) \to^\epsilon \tau$.

3.4 Greedy inference

The requirement that the arrow node variable be polymorphic in rules AppD and AppM just prevents guessing polymorphism. Removing this requirement would preserve type soundness but would break the distinction between known types and inferred types. Not only we would loose the principal type property, but typability would likely become intractable.

This would actually allow instantiating type variables by arbitrary module-dependent arrow types, even inventing module-types that do not appear elsewhere !

However, an implementation need not do that. It may instantiate type variables by module-dependent types *only when some typing constraint requires it* and thus never invent module-dependent types. This is actually what OCaml does by default, when it is not explicitly required to infer principal types. We call this process *greedy inference*: it infers polytypes when forced to, and immediately treats them as known. Typechecking may then succeed or fail, depending on the order in which type inference, hence type instantiation is being performed.

Soundness While the greedy typing rules break principality, they should still preserves type soundness, as well as the original typing rules, which defines a subrelation. A proof of type safety could be done by translation of SmallOCaml to F^ω, following Blaudeau, Rémy, and Radanne (2024), but it is out of the scope of this paper.

4 Implementation details

The implementation of modular explicits contains a few subtleties that should be reported.

Current type inference in OCaml typechecks n-ary applications $\overline{\tau_i \to} \tau$ by pairing each argument with its expected type before typechecking the arguments. However when typing a module-dependent function application, we typecheck the module before looking at the rest of arguments.

As already mentioned, OCaml currently requires that Q has no free type variables (it may still have node-variables).

Unification in OCaml links both types together. Thus, when unifying $(\textbf{module}\ \mathcal{X}_1 : Q_1) \to^\epsilon \tau_1$ with $(\textbf{module}\ \mathcal{X}_2 : Q_2) \to^\epsilon \tau_2$ we need to unify τ_1 and τ_2 together, leaving us a type τ which is a mix of both. However τ_1 and τ_2 did not live in the same context. Thus we also unify $\mathcal{X}_1$ with $\mathcal{X}_2$ in order to ensure the validity of $(\textbf{module}\ \mathcal{X}_1 : Q_1) \to^\epsilon \tau$ and $(\textbf{module}\ \mathcal{X}_2 : Q_2) \to^\epsilon \tau$.

As with first-class modules, the arguments of two module-dependent functions are compatible only if both modules have the same runtime representation and are coercible to one another, i.e., the values in the same order but types can change position. By contrast, ground coercion requires only the same runtime representation and one way of coercion.

Exhaustiveness and redundancy checks when pattern matching on GADTs isn't complete due to the undecidability of this problem (Garrigue and Normand, 2017). As all first-class modules are considered compatible, we don't compare the arguments of two module dependent functions. Thus $(\textbf{module}\ \mathcal{X}_1 : Q_1) \to^\epsilon \tau_1$ and $(\textbf{module}\ \mathcal{X}_2 : Q_2) \to^\epsilon \tau_2$ are considered compatible if τ_1 and τ_2 are compatible in an incomplete context (because we don't add $\mathcal{X}_1$ and $\mathcal{X}_2$ to the context).

When typing recursive functions, we try to "guess" the type of function before typing the body. This allows more precise error messages. However, in the case of polymorphism we cannot do this without a complete annotation. This problem already exists in OCaml currently with `type a`. For example the two following codes fails to type:

```
let rec search (type a) (module X : Ord with type t = a)
              (x : X.t) : X.t list → X.t option = function
```

```
  | [] → None
  | h::_ when equal (module X.Eq) x h → Some x
  | _::t → search (module X) x t

Error: The signature for this packaged module couldn't be inferred.
```

```
let rec search (module X : Ord) (x : X.t) : X.t list → X.t option = function
  | [] → None
  | h::_ when equal (module X.Eq) x h → Some x
  | _::t → search (module X) x t

Error: The signature for this packaged module couldn't be inferred.
```

The reason being that we cannot approximate the type inside a scope definition such as `(type a)` or `(module X : Ord)`, thus we give the most permissive type to `search` before typing the body of the function : an unconstrained type variable.

The solution is to annotate the recursive definition itself:

```
let rec (search : (module X : Ord) → X.t → X.t list → X.t option) =
  fun (module X) x → function
  | [] → None
  | h::_ when equal (module X.Eq) x h → Some x
  | _::t → search (module X) x t

let _ = search (module Ord_int) 2 [1; 2; 3]
```

This typechecks because this provides sufficient annotation to `search` to know its type before typing the body.

4.1 Path condition of dependent application

In the implementation we managed to relax the path condition of the module argument when using rule AppD. One could understand typing the application of a module that isn't a path to a dependent function `f (module M) v` as `(let module X = M in f (module X)) v` which is typable using the rules presented before.

The scope mechanism is then applied in order to ensure type-soundness and that types abstract introduced in `M` does not appear in the return type of the application.

4.2 Greedy OCaml inference

By default, the OCaml typechecker is order dependent, i.e., some programs are accepted based on the order in which type inference proceeds. Modular explicits are also impacted by this and are typechecked by default in greedy mode. We use the same mechanism as with labels, first-class modules, and polymorphic methods to allow the user to receive warnings (which can be turned into errors that reject the program) when non-principal types are inferred, i.e., using the *-principal* option.

The choice of the greedy mode as the default is to avoid duplicating types at some program points in order to decorrelate node variables, which could significantly slow down typechecking. This also allows removing some type annotations, but this is not fragile as not stable by program transformations.

5 Polymorphic parameters

A recent extension by White (2023) proposed to allow polymorphic parameters as arguments to functions, allowing the user to write:

$$
\begin{array}{ll}
1: & \textbf{let}\ f = \lambda(x : \sigma).\, a_0\ \textbf{in} \\
2: & \textbf{let}\ h = \lambda(f : \sigma \to^{\epsilon} \tau).\, a_1[f\ a_2]\ \textbf{in} \\
3: & h\ f
\end{array}
$$

On the first line we defined a function f whose parameter expects a value of a polymorphic type σ that can be used in a_0 at different instances. The inferred type of f is $\sigma \to \tau$. On the second line, we define a higher-order function that receives a function f of type $\sigma \to \tau$ as argument that can be applied to an argument a_2 without any explicit type annotation—but will indeed have to be at least as polymorphic as σ. Finally, we apply h to f.

This extensions uses the two following typing rules:

$$
\textsc{Fun}\ \frac{\Gamma, x : \forall\bar{\epsilon}.\, \sigma_2 \vdash a : \tau \qquad \sigma \lessapprox \sigma_1 \Rightarrow \forall\bar{\epsilon}.\, \sigma_2}{\Gamma \vdash \lambda(x : \sigma).\, a : \sigma_1 \to^{\epsilon} \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash a_1 : \sigma \to^{\epsilon} \tau \qquad \epsilon \# \Gamma \qquad \Gamma \vdash a_2 : \sigma}{\Gamma \vdash a_1\ a_2 : \tau}
$$

Just extending the relation $\tau \lessapprox \tau_1 \Rightarrow \forall\bar{\epsilon}.\, \tau_2$ to type schemes in the obvious way.

Interestingly, when σ is closed, this can be encoded as:

$$
\begin{array}{l}
\textbf{let module type}\ S = \textbf{sig val}\ \mathit{value} : \sigma\ \textbf{end in} \\
\textbf{let}\ f = \lambda(\textbf{module}\ \mathcal{X} : S).\, \textbf{let}\ x = \mathcal{X}.\mathit{value}\ \textbf{in}\ x\ x\ \textbf{in} \\
a_1[\textbf{let module}\ M = \textbf{struct let}\ \mathit{value} = a_2\ \textbf{end in}\ f\ (\textbf{module}\ M)]
\end{array}
$$

using a *non-dependent* first-class module. The amount of information to be provided is similar, i.e., just the type σ of the parameter of f, except that it has to be defined as a named signature, which make it more involved when the type is not closed.

6 Discussion

Modular explicits are just about manipulating first-class functors as module-depend functions. They provide syntactic and typing shortcuts that avoid the unpractical boiler plate encoding, and facilitate programming with modules at a smaller scale.

Still, despite the conciseness their provide, some programming patterns they enable remain somewhat verbose. This is due to several restrictions on the typing of first-class modules that are orthogonal to modular explicits, but still have an impact on modular explicits, which in turn calls for further extensions of the module language.

For example, functor type arguments, which we used to have shorter, more convincing code-pattern examples, are strictly speaking not part of modular explicits. Below, we discuss a few other small extensions that could improve the users experience with modular explicits. Although they are all orthogonal extensions to modular explicits, they are even more striking in the presence of modular explicits and should be considered altogether with modular explicits, at least to ensure coherent design choices.

While we argue that modular explicits are a standalone proposal, they will still be used as the target of elaboration of modular implicits. Hence, the syntax of modular implicits should also be taken into account when introducing modular explicits, so that we may later easily switch between implicit and explicit modular arguments. We may also benefit from the rich set of programming patterns and examples that have been explored for modular implicits by Reader et al. (2024), and their recommendations (Reader and Vlasits, 2024), most of which should also be relevant for modular explicits.

6.1 Compatibility with first-class modules

There has been discussions in the OCaml community on whether module-dependent function types should be an extension of first-class module types or a different construct with convertibility between the two views.

Treating them as two different constructions avoids the overlapping. Then, only the new construct, module-dependent functions, need to be understood in isolation, which is simpler to comprehend than the overloaded superposition of two closed but different constructs.

Identifying them may arguably benefit to the user who can then think of modular explicits as just an extension of first-class modules without the need for a deep, formal understanding of the differences between their two faces.

A consensus seems to have emerged in favor of identifying them. We have shown that both typings agree when the two constructs overlap, which therefore makes it possible to identify them as a single construct.

6.2 More inlining

One of the main limitations we encountered when programming with modular explicits is the restriction that module arguments and their signatures should to be named paths.

In some cases, we could work around use the **`let module`** $\mathcal{X} = M$ **`in`** $a[\mathcal{X}]$ construct. Still, in these cases, it would be more convenient to allow M to be inlined. Of course, this may change the evaluation order or duplicate some computation, although we may expect that M be an applicative module and $\mathcal{X}$ just used once in most cases.

However, a primitive treatment relaxing this restriction would indeed be preferable.

6.2.1 Functor type arguments

In examples above, we have used functor type arguments, a proposal by Vivien (2024b) currently under discussion in the OCaml community, although they are not part of modular explicits per say. In the simplest form of this extension, the functor **`functor`** (**`type`** t) $\to M[t]$ is to be understood as **`functor`** $(\mathcal{X}_\tau : \mathsf{Type}) \to M[X_\tau.t]$ where Type is a predefined module type equal to **`sig type`** t **`end`** for some fixed, reserved field t and (**`module type`** τ) as an argument means (**`module`** T_τ) where T_τ is the predefined module equal to **`struct type`** $t = \tau$ **`end`**.

Therefore, **`functor`** (**`type`** t) $\to M[t]$ behaves as a functor and blocks the evaluation of the body, which is then performed and repeated at each application site. An alternative choice would be that *type* abstraction behaves as in the core language and does not block the evaluation, while as a counterpart the body would be restricted to be a value form.

Perhaps surprisingly, our previous examples only used one such functor T defined as λ(**`type`** a). **`struct type`** $t = a$ **`end`** to inline module arguments restricted to a single type–and not turn them into type arguments. The reason is that the syntax (**`type`** t) for module type abstraction is conflicting with the core language syntax for explicit type abstraction and therefore does not allow to write a module dependent function that receives a `(type a)` argument. This has to be fixed.

6.2.2 Relaxing functor arguments

Functor *type* arguments, discussed just above solve the particular problem of module arguments restricted to a single type definition. These can now be inlined. Although a common case, this is just a particular instance of functor arguments that should be allowed to be inlined.

The implementation restricts module arguments M in a (**module** M) to have a named signature Q, which amounts to require that M be extractable to a path P, which includes, in particular, type annotation $M : Q$. This restriction is not necessary. Indeed, one could always see a (**module** M) as **let module** $\mathcal{X} = M$ **in** a (**module** $\mathcal{X}$) when M is not a path P. The scope of $\mathcal{X}$ is then restricted to the expression a (**module** $\mathcal{X}$) and therefore $\mathcal{X}$ cannot appear in its type. Even though, this should not be a problem when the signature of $\mathcal{X}$ is concrete (i.e., it does not introduce abstract types) including the case where it contains core language free (unification) type variables.

6.2.3 Parametric signatures

More generally, we wish type signatures to have core language free (unification) type variables. Then, signature definitions should be allowed to be parametrized by type variables, such as:

$$\textbf{module type}\ T(\bar{\alpha}) = S[\bar{\alpha}]$$

so that $T(\bar{\tau})$ can then be used to mean $S[\bar{\tau}]$, if signatures could be inlined as described just above in §6.2.2, or otherwise a named signature $T_{\bar{\tau}}$ that would have been predefined as **module type** $T_{\bar{\tau}} = S[\bar{\tau}]$. In fact, when $S[\bar{\alpha}]$ is a signature **sig** D **end**$[\bar{\alpha}]$, this can actually be emulated by defining **module type** $T =$ **sig type** $\bar{t}\ D[t]$ **end** and seeing $T(\tau)$ as syntactic sugar for T **with type** $\bar{t} := \bar{\tau}$. Therefore, parametric signatures are just a more natural and standard notation for type abstraction that prevent T to be used non-applied and avoids the introduction of useless names $\bar{\tau}$ for free the free type variables $\bar{\alpha}$.

The key point remains that signatures in general, may contain free type variables.

6.3 Polymorphic arguments

Technically, type inference for module-dependent functions is quite close to type inference for polymorphic parameters. Indeed, if we allowed first-class module types to have free type variables, the latter could be emulated with the former. Still, when polymorphic arguments applies, they are simpler than their encoding in terms of module-dependent functions. Hence, they are not a redundant, but a useful complementary extension.

6.4 Transparent ascription

Since we presented modular explicits as the target of the elaboration of modular implicits, we should mention another extension, *transparent ascription*, which although not necessary for modular explicits is a key for modular implicits. Hence, to be complete, we briefly discuss and explain the need for transparent ascription in the presence of modular implicits.

When synthesizing modules from signatures, or more generally when the semantics is defined by elaboration, it is necessary that all elaborations of the same program are observationally equivalent, in which case we say that the program is *coherent*, so that its semantics is deterministic and thus clearly defined. Programs for which the coherence cannot be ensured should be rejected.

While there are often several ways to infer a module of a given signature—just by module applications taken from a database of implicit modules, it is often the case that all solutions will actually compute the same module value and are thus observationally equivalent. While it would be difficult to trace expression aliases by typechecking in the core-language expressions, it is easier to prove this at the module-level, using applicative functors and path equivalences. However, the actual typechecking of applicative modules in OCaml is too weak to track modular aliases through functor applications. Hence, an extension to OCaml under consideration is to

add transparent ascription in paths to better keep track of module aliases. An explained above, while essential for modular implicits, this is orthogonal to and not for modular explicits for which there is no coherence problem. Hence, we leave this extension out of the modular explicit proposal.

Conclusion

Modular explicits are a small extension to first-class modules. In principle, they do not increase expressiveness. In practice however, they considerably improve conciseness and the interaction between the core and module levels, by turning module-level first-class functors into core-level module-dependent functions. This enables new programming patterns that were previously unpractical, and programming with modules directly at the core-level.

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*. Interestingly, the implementation reuses the OCaml existing trick to keep track of principal types and smoothly move from dependent to non-dependent arrow types when needed.

Although modular explicits have been originally designed as the language in which *modular implicits* 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 made their way to the compiler as soon as possible.

Modular explicits allow modules to be used for programming in the small, which in turn requires new extensions of the module system to further smoothen the interaction between the module and core levels.

Acknowledgments

We would like to thank Vincent Laviron, the Flambda team, and all the other people at OCaml-Pro for hosting the internship that lead to the implementation of modular explicits. We also thank Leo White for his advices and code reviewing. We wish to acknowledge the work of Matthew Ryan on his own implementation of modular explicits (Ryan, 2019), which lead to various discussions on github that gave an insight on their implementation.

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.

D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 63–70. ACM, 2007. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190229. URL `https://doi.org/10.1145/1190216.1190229`.

J. Garrigue and J. L. Normand. Gadts and exhaustiveness: Looking for the impossible. In *ML Family/OCaml*, 2017. URL `https://api.semanticscholar.org/CorpusID:10817992`.

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.

X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*, pages 142–153, San Francisco, California, United States, 1995. ACM Press. ISBN 978-0-89791-692-9. doi: 10.1145/199448.199476. URL http://portal.acm.org/citation.cfm?doid=199448.199476.

B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360, 2010. doi: 10.1145/1869459.1869489. URL https://doi.org/10.1145/1869459.1869489.

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.

M. Ryan. Modular explicits. On github https://github.com/ocaml/ocaml/pull/9187, Dec. 2019. OCaml pull request.

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

S. Vivien. Type arguments from modules. On github https://github.com/ocaml/RFCs/blob/27e94773c6b191a372ade263195dade779132dc2/rfcs/type_arguments_for_modules.md, June 2024b. OCaml RFC.

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.