Extending OCaml’s open

Runhang Li
Twitter, Inc
rli@twitter.com

Jeremy Yallop
University of Cambridge
jeremy.yallop@cl.cam.ac.uk

Abstract

We propose a harmonious extension of OCaml’s open construct with many useful applications.

The construct open M imports the names exported by the module M into the current scope. Currently M is required to be the path to a module. Extending open to instead accept an arbitrary module expression makes it possible to succinctly address a number of existing scope-related difficulties that arise when writing OCaml programs.

1 open vs include

OCaml provides two operations for introducing names exported from one module into another module:

open M           include M

Both operations introduce M’s bindings into the current scope; include additionally re-exports the bindings from the current scope.

A second difference between open and include concerns the form of the argument. The argument to open is a module path:

open A.B.C

The argument to include can be any module expression:

include F(X)
include (M:S)
include struct...end

This paper describes an extension to open that eliminates that second difference, so that both open and include accept an arbitrary module expression as argument (Figure 1). In practice, allowing the form open struct ... end extends the language with a non-exporting version of every type of declaration, since any declaration can appear between struct and end.

The extended open has many applications, as we illustrate with examples condensed from real code (Section 2). Our design also resolves some problems in OCaml’s signature language (Section 3). We touch briefly on restrictions and other design considerations (Section 4).

Status Following the presentation of this proposal at the OCaml 2017 workshop [6], the design was accepted for inclusion into OCaml. We anticipate that it will appear in the OCaml 4.08 release.
Extending OCaml's open

Current design:
Only basic paths are allowed

```ocaml
open M.N
```

Extended design (this paper):
Arbitrary module expressions are allowed

```ocaml
open M.N open F(M) open (M:S)
open struct ... end
```

Figure 1: The open construct and our proposed extension

2 Extended open in structures: examples

Unexported top-level functions
The extended open construct supports bindings that are not exported. In the code on the left, x is available in the remainder of the enclosing module, but it is not exported from the module, as shown in the inferred signature on the right:

```ocaml
open struct let x = 3 end
let y = x
val y : int
```

A workaround for type shadowing
One common programming pattern is to export a type t in each module. For example, the standard library defines types Float.t, String.t, Complex.t, and many more. However, this style leads to problems when the definition of one such t must refer to another. For example, in the following code, t₁ and t₂ cannot both be renamed t, since both names are used within a single scope, where all occurrences of t must refer to the same type:

```ocaml
type t₁ = A
module M = struct
  type t₂ = B of t₂ * t₁ | C
end
```

The extended open construct resolves the difficulty, making it possible to give an unexported local alias for the outer t:

```ocaml
type t = A
module M = struct
  open struct type t' = t end
  type t = B of t * t' | C
end
```

Local definitions scoped over several functions
A common pattern involves defining one or more local definitions for use within one more more exported functions. Typically, the exported functions are defined using tuple pattern matching. Here is an example, defining f and g in terms of an auxiliary unexported function, aux:

```ocaml
let f, g =
  let aux x y =
    ...
    in (fun p → aux p true),
       (fun p → aux p false)
```

---

1 See draw_poly, draw_poly_line and dodraw in the OCaml Graphics module for an example. [https://github.com/ocaml/ocaml/blob/4697ca14/otherlibs/graph/graphics.ml](https://github.com/ocaml/ocaml/blob/4697ca14/otherlibs/graph/graphics.ml) lines 105–117
This style has several drawbacks: the names \( f \) and \( g \) are separated from their definitions by the definition of \( \text{aux} \); the unsugared syntax \( \text{fun} \ x \to \ldots \) must be used in place of the sugared syntax \( \text{let} \ f \ x = \ldots ; \) and the definition allocates an intermediate tuple. With extended \texttt{open}, these problems disappear:

\[
\text{include struct}
\begin{align*}
\text{open struct let} & \ aux \ x \ y = \ldots \ 	ext{end} \\
\text{let} & \ f \ p = \ aux \ p \ \text{true} \\
\text{let} & \ g \ p = \ aux \ p \ \text{false}
\end{align*}
\text{end}
\]

**Local exception definitions** OCaml’s \texttt{let module} construct supports defining exceptions whose names are visible only within a particular expression\(^2\). Limiting the scope of exceptions supports a common idiom in which exceptions are used to pass information between a raiser and a handler without the possibility of interception \footnote{OCaml 4.04 adds a more direct construct \cite{OCaml404}}. (This idiom is perhaps even more useful for programming with effects \cite{effects}, where information flows in both directions.)

Limiting the scope of exceptions can make control flow easier to understand and, in principle, easier to optimize; in some cases, locally-scoped exceptions can be compiled using local jumps \cite{localjumps}.

The extended \texttt{open} construct improves support for this pattern. While \texttt{let module} allows defining exceptions whose names are visible only within particular expressions, extended \texttt{open} also allows limiting visibility to particular declarations. For example, in the following code, the \texttt{Interrupt} exception is only visible within the bindings for \texttt{loop} and \texttt{run}:

\[
\text{include struct}
\begin{align*}
\text{open struct exception} & \ \text{Interrupt} \ 	ext{end} \\
\text{let rec} & \ \text{loop} () = \ldots \ \text{raise} \ \text{Interrupt} \\
\text{let rec} & \ \text{run} = \text{match} \ \text{loop} () \ 	ext{with} \\
& | \ exception \ \text{Interrupt} \to \ \text{Error} \ "\text{failed}" \\
& | \ x \to \ \text{Ok} \ x
\end{align*}
\text{end}
\]

**Shared state** Similarly, extended \texttt{open} supports limiting the scope of global state to a particular set of declarations:

\[
\text{open struct}
\begin{align*}
\text{open struct let} & \ \text{counter} = \text{ref} \ 0 \ 	ext{end} \\
\text{let} & \ \text{inc} () = \text{incr} \ \text{counter} \\
\text{let} & \ \text{dec} () = \text{decr} \ \text{counter} \\
\text{let} & \ \text{current} () = !\text{counter}
\end{align*}
\text{end}
\]

Here the names \texttt{inc}, \texttt{dec} and \texttt{current} are accessible in the code that follows, but the shared reference \texttt{counter} is not.

**Local names in generated code** It is common in OCaml to use low-level code generation in the implementation of libraries and programs.

Until recently, the most common system for compile-time code generation was the Camlp4 preprocessor that performs transformations on the concrete syntax of programs. These transformations can result in the generation of entirely new functions and modules as is the case with

\footnote{OCaml 4.04 adds a more direct construct \cite{OCaml404}}
the **deriving** framework that generates pretty-printers, serializers, and other functions from type definitions [8].

More recently, the **ppx** framework, which supports transformations on abstract syntax [7], has become popular. Syntax transformers based on **ppx**, such as **ppx_deriving** (a reimplementation of **deriving**) and **ppx_stage** (a preprocessor for typed multi-stage programming), may also generate large amounts of code.

The definitions introduced by Camlp4 and **ppx** extensions are often intended to be details of the implementation, not exposed to the programmer, and with names that do not interact with the remainder of the program. However, it is currently difficult to introduce completely anonymous declarations in OCaml. A common solution is to generate instead a module with a "sufficiently unique" name — i.e. a name that is unlikely to clash with names defined by the programmer. For example, here is a simple expression, representing a function that generates a code fragment, written using **ppx_stage**:

\[
\text{fun } x \rightarrow \text{[\%code [\%e x]]}
\]

The **ppx_stage** extension transforms the function body to generate a module with various components that implement the behaviour of the code fragment:

\[
\text{module Staged_349289618 =}
\]

\[
\begin{align*}
\text{struct} \\
\text{let staged0 hole''}_1 = \\
\text{let contents''}_1 = \text{hole''}_1 \text{ in} \\
\text{...}
\end{align*}
\]

If, as is often the case, the user of **ppx_stage** does not provide an interface file, the generated module **Staged_349289618** will appear in the interface to the module, exposing the internal details of the code generation scheme.

**Restricted open** It is sometimes useful to import a module under a restricted signature. For example, the statement

\[
\text{open (Option : MONAD)}
\]

imports only those identifiers from the **Option** module that appear in the **MONAD** signature.

However, there is a caveat here: besides excluding identifiers not found in **MONAD**, OCaml’s module ascription also hides concrete type definitions behind abstract types, which is typically not the desired behaviour for **open**. This behaviour can be avoided by adding an explicit constraint to the constraining **MONAD** signature to maintain the equality between the type **t** in the signature and **Option.t**:

\[
\text{open (Option : MONAD with type } 'a t = 'a Option.t)}
\]

However, this is rather verbose. The difficulty could be more succinctly addressed by extending OCaml with a construct found in Standard ML, namely transparent signature ascription, a useful feature in its own right.

### 3 Extended **open** in signatures: examples

In signatures, as in structures, the argument of **open** is currently restricted to a qualified module path (Figure 1). As in structures, we propose extending **open** in signatures to allow an arbitrary module expression as argument. However, while extended **open** in structures evaluates its argument; **open** in signatures is used only during type checking.
This section presents examples of signatures that benefit from extended open. Our examples all involve type definitions, but it is possible to construct similar examples for other language constructs, such as functors and classes.

**Unwriteable, unprintable signatures** The OCaml compiler has a feature that is often useful during development: passing the -i flag when compiling a module causes OCaml to display the inferred signature of the module. However, users are sometimes surprised to find that a signature generated by OCaml is subsequently rejected by OCaml, because it is incompatible with the original module, or even because it is invalid when considered in isolation.

Here is an example of the first case. The signature on the right is the output of ocamlc -i for the module on the left:

```ocaml
type t = T1
module M = struct
  type t = T2
  let f T1 = T2
end
```

```ocaml
type t = T1
module M : sig
  type t = T2
  val f : t → t
end
```

The input and output types of M.f are different in the module, but printed identically. That is, the printed type for f is incorrect.

Here is an example of the second case, again with the original module on the left and the generated signature on the right:

```ocaml
type t = T
module M = struct
  type 'a t = S
  let f T = S
end
```

```ocaml
type t = T
module M : sig
  type 'a t = S
  val f : t → t
end
```

This time the generated signature is ill-formed because the type M.t requires a type argument, but is used without one.

If these problems arose from a shortcoming in the implementation of the -i flag then there would be little cause for concern. In fact, they point to a more fundamental issue: many OCaml modules have signatures that cannot be given a printed representation. It is impossible to generate suitable signatures; more importantly, it is impossible even to write down suitable signatures by hand.

The problem in both cases is scoping: an identifier such as t always refers to the most recent definition, and there is no way to refer to other bindings for the same name. The nonrec keyword, introduced in OCaml 4.02.2, solves a few special cases of the problem, by making it possible to refer to a single other definition for t within the definition of t itself. But most such problems, including the examples above, are not solved by nonrec.

The extended open solves the problem entirely, by making it possible to give internal aliases to names. For example, here is a valid signature for the first case above using extended open.

```ocaml
type t = T1
module M = struct
  open struct
    type t' = t
  end
  type t = T2
  let f T1 = T2
end
```

```ocaml
type t = T1
module M : sig
  type t = T2
  val f : t' → t
end
```
The OCaml compiler might similarly insert a minimal set of aliases to resolve shadowing without the need for user intervention. (At the time of writing, however, our implementation does not yet include this improvement to signature printing.)

And, of course, extended `open` also makes it possible for users to write those signatures that are currently inexpressible.

**Local type alias in a signature**  Even in cases with no shadowing, it is sometimes useful to define a local type alias in a signature\(^3\). In the following code, the type `t` is available for use in `x` and `y`, but not exported from the signature.

```ocaml
open struct type t = int → int end
val x : t
val y : t
```

### 4 Restrictions and design considerations

**Dependency elimination**  OCaml’s applicative functors impose a number of restrictions on programs beyond type compatibility. One such restriction arises in functor application: it must be possible to “eliminate” in the functor result type each type defined in the functor argument\(^5\). For example, given the following functor definition

```ocaml
module F(X: sig type t val x: t end) =
struct
let x = X.x
end
```

the following application is valid:

```ocaml
module A = struct type t = T let x = T end
module B = F(A)
```

and `B` receives the following type:

```ocaml
module B : sig val x : A.t end
```

However, the following application is not allowed:

```ocaml
F(struct type t = T let x = T end)
```

since the result of the application cannot be given a type, as there is no suitable name for the type of `x`.

The extended `open` construct has a similar restriction. For example, the following program is rejected by the type-checker because the only suitable name for the type of `x`, namely `t`, is not exported:

```ocaml
open struct type t = int end
let x = T
```

---

\(^3\) For example, the functions `comment`, `maintainer`, `run`, `cmd`, `user`, `workdir`, `volume`, and `entrypoint` in the Dockerfile module would benefit from such an alias. [GitHub](https://github.com/avsm/ocaml-dockerfile/blob/e0dad1a/src/dockerfile.mli)
Here is the error message from the compiler:

**Error:** The module identifier M#0 cannot be eliminated from val x : M#0.t

Since the restriction for the extended open construct is the same as the existing functor restriction, we can reuse the existing implementation of the check in the OCaml type checker. In particular we use the Mtype.nondep_supertype function to check if introduced identifiers can be eliminated from rest of the structure [5].

**The Avoidance Problem** The avoidance problem [2] is closely connected with dependency elimination. The problem is as follows: it is sometimes necessary to find a signature for a module that avoids mention of one of its dependencies; however, it is not always possible to find a best, or principal (i.e. most-specific) such signature, since the candidates may be incomparable.

Dreyer [2] gives the following example of the surprising behaviour that can arise by OCaml’s lack of principal signatures. Suppose a signature S, and two functors F and G that each take an argument of type S, as follows:

```ml
module type S = sig type t end
module F (X : S) = struct type u = X.t type v = X.t end
module G (X : S) = struct type u = X.t type v = u end
```

Semantically, F and G are equivalent: in both cases, the types u, v and X.t are all equal in the body of the functor. If F and G are applied to a module denoted by a path, then the resulting signatures are equivalent. For example, here is the result of applying F and G to the top-level module Char:

```ml
# module FC = F(Char);;
module FC : sig type u = Char.t type v = Char.t end
# module GC = G(Char);;
module GC : sig type u = Char.t type v = u end
```

Since the argument Char has a globally-visible name, OCaml is able to preserve all the equalities in the output types.

However, when the module passed as argument is not denoted by a path then the result of applying F is different from the result of applying G:

```ml
# module FI = F((struct type t = int end : S));;
module FI : sig type u type v end
# module GI = G((struct type t = int end : S));;
module GI : sig type u type v = u end
```

This time OCaml cannot preserve all the equalities, since there is no way of naming the type member of the module passed as argument in the output signature. Consequently, the type equalities that syntactically involve X.t are discarded, making the types FI.u FI.v, and GI.u abstract.

A similar situation arises with the extended open construct, which inherits OCaml’s approach towards elimination of modules in signatures.

In the following examples M is given a more general type than N, even though the two modules are semantically equivalent:
Here are the types assigned by OCaml:

```ocaml
module M : sig
type u and v = u
end

type u = t and v = u
```

As with F and G, the type equalities syntactically involving \( t \) are discarded, even though the two modules are semantically equivalent, since the types \( u, v \) and \( t \) are all equal in each case.

**Evaluation of extended open in signatures** Here is a possible objection to supporting the extended `open` in signatures: although local type definitions are useful within signatures, local value definitions are not, and so it would be better to restrict the argument of `open` to permit only type definitions.

For example, the following runs without raising an exception:

```ocaml
module type S =
sig
  (* no exception! *)
  open struct assert false end
end
```

Within a signature, `open`'s argument is used only for its type, and so the expression `assert false` is not evaluated.

In fact, this behaviour follows an existing principle of OCaml’s design: *module expressions in type contexts are not evaluated.* For example, the `module type of` construct, currently supported in OCaml, also accepts a module expression that is not evaluated:

```ocaml
module type S = (* no exception! *)
module type of struct assert false end
```

And similarly, functor applications that occur within type expressions in OCaml are not evaluated:

```ocaml
module F(X: sig end) =
struct
  assert false
  type t = int
end
let f (x: F(List).t) = x (* no exception! *)
```

**Acknowledgments**

We thank Leo White and the OCaml’17 reviewers for helpful comments and suggestions, and Alain Frisch for helpful suggestions about the implementation.
References


