Extending OCaml’s open

Runhang Li
Twitter, Inc
rii@twitter.com

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

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

1. open vs include
OCaml provides two operations for introducing names exported from one module into another module:

\[ \text{open } M \quad \text{include } M \]

Both operations introduce \( M \)’s bindings into the current scope; include also 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:

\[ \text{open } A.B.C \]

The argument to include can be any module expression:

\[ \text{include } F(X) \quad \text{include struct...end} \]

This note proposes extending open to eliminate that second difference, so that both open and include accept an arbitrary module expression as argument (Fig. 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 (§2). Our proposal also resolves some problems in OCaml’s signature language (§3). We touch briefly on restrictions and other design considerations (§4).

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 signature on the right:

\[
\begin{align*}
\text{open struct} & \quad \text{let } x = 3 \quad \text{end} \quad (* \text{no entry for } x*) \\
\text{let } y & = x \\
\text{val } y & : \text{int}
\end{align*}
\]

A workaround for type shadowing One common programming pattern is to define a type \( t \) in each module. However, this style leads to problems when the definition of one such \( t \) must refer to another. For example, in the following code, \( t_1 \) and \( t_2 \) 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.

\[
\begin{align*}
type \ t_1 & = A \\
\text{module } M & = \text{struct} \\
& \quad \text{type } t_2 = B \text{ of } t_2 * t_1 | C \\
\text{end}
\end{align*}
\]

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

\[
\begin{align*}
type \ t & = A \\
\text{module } M & = \text{struct} \\
& \quad \text{open struct } t' = t \text{ end} \\
& \quad \text{type } t = B \text{ of } t' * t' | C \\
\text{end}
\end{align*}
\]

Local definitions scoped over several functions A common pattern involves defining one or more local definitions for use within one more more exported functions\(^2\). Typically, the exported func-

\[
\begin{align*}
\text{fun } x & = \text{aux } x \cdot y \\
\text{in } \text{aux } x \cdot y
\end{align*}
\]

This style has several drawbacks: the names \( f \) and \( g \) are separated from their definitions by the definition of aux; the sugared syntax \( \text{let } x = \ldots \) must be used in place of the sugared syntax \( \text{let } \ldots \) and the definition allocates an intermediate tuple. With extended open, these problems disappear:

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

Local exception definitions OCaml’s let module construct supports defining exceptions whose names are visible only within a particular expression\(^3\). 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 \([3]\). (This idiom is perhaps even more useful for programming with effects \([1]\), 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 \([2]\).

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

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

Shared state Similarly, extended open supports limiting the scope of global state to a particular set of declarations:

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

Restricted open It is sometimes useful to import a module under a restricted signature\(^4\). For example, the statement...
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 when 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
open (Option : MONAD)

type t = T
module M = struct
  type t = T
  let f T1 = T2
end

val x : t
val y : t
```

The input and output types of M.t 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
open (Option : MONAD)

val x : t
val y : t
```

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
open (Option : MONAD)

val x : t
val y : t
```

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. 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 → 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: types defined in the argument of a functor must be “eliminable” in the result [4]. For example, given the following functor definition

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

let f T1 = T2
val f : t → t
end
```

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 = T end
let x = T
```

Here is the error message from the compiler:

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

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 = (* 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 construct assert false end 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! *)
```

Local open It would be also be possible to extend expression-local open constructs of the form let open M.N in e. However, since expressions, unlike declarations, do not export names, it does not appear very useful to do so.
Acknowledgments

We would like to thank Leo White for his helpful comments and suggestions.

Notes

1 A modified compiler implementing this design can be tested out at: http://ocamllabs.io/iocamljs/open-struct.html
3 OCaml 4.04 adds a more direct construct [2]
4 Drawn from a proposal by Leo White on the compiler hacking tasks: https://github.com/ocamllabs/compiler-hacking/wiki/Things-to-work-on#signatured-open-command
5 For example, the functions comment, maintainer, run, cmd, user, workdir, volume, and entrypoint in the Dockerfile module would benefit from such an alias. https://github.com/avsm/ocaml-dockerfile/blob/e0dad1a/src/dockerfile.mli

References