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.

OCaml's existing construct `open M` imports the names exported by the module `M` into the current scope. At present `M` is required to be the *path* to a module. We propose extending `open` to instead accept an arbitrary module *expression*, making it possible to succinctly address a number of existing scope-related difficulties that arise when writing OCaml programs.

1 Introduction: `open` vs `include`

Programming languages intended for large-scale, modular programming often include features for making names defined in one scope available without qualification in another scope. OCaml provides two such operations, via the keywords `open` and `include`:

```
open M                    include M
```

Both of these operations introduce the bindings exported by the module `M` into the current scope. Additionally, `include` re-exports the bindings from the current scope. This distinction is a useful one, since it is not always appropriate to re-export the names used within a module.

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

```
open A.B.C
```

In contrast, the argument to `include` can be any module expression, such as a functor application, signature-constrained expression, or structure body:

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

This distinction is less useful: there is no fundamental reason why `include` should accept arbitrary module expressions, while `open` should not.

This paper explores the consequences of extending `open` to eliminate the 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 useful 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) before sketching the implementation (Section 5) and comparing some alternative designs (Section 6).

Submitted to:
ML 2017

Current design:
Only basic paths are allowed

```
open M.N
```

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

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

Figure 1: The `open` construct and our proposed extension

1.1 Status

Following the presentation of this proposal at the OCaml 2017 workshop [12], a variant of this design was discussed at the Caml developers meeting and accepted for inclusion into OCaml 4.08. Section 7 gives more details.

2 Extended `open` in structures: examples

Effectively managing names and scope is a crucial part of structuring programs. The examples in this section show how the lack of a facility for local (non-exporting) declarations can result in awkward structure or inappropriate scoping in OCaml programs, and further show how these problems are eliminated by the extended `open` construct[1].

2.1 Unexported top-level values

A straightforward use of the extended `open` construct is the introduction of local declarations 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:

```
open struct let x = 3 end        (* no entry for x *)
let y = x                        val y : int
```

2.2 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, renaming the `s` to `t` requires some care:

```
type s = A
module M = struct
  type t = B of s | C
end
```

Since type definitions are recursive by default, naively renaming `s` to `t` in the definition of `M.t` changes the meaning of the definition so that the argument of `B` now refers to the inner `t`:

```
type t = A
module M = struct
  type t = B of t | C
end
```

[1] The reader familiar with Standard ML will recognise that the `local` construct in that language, an inspiration for this proposal, can also solve the problems described here. We return to this point in Section 6.1.

The `nonrec` keyword, added in a recent version of OCaml (4.02.2, released June 2015), overrides this default, making the definition of `t` non-recursive, and restoring the original meaning:

```
type t = A
module M = struct
  type nonrec t = B of t | C
end
```

However, in cases where a single type definition must contain both recursive references and references to another type of the same name, `nonrec` cannot help. For example, in the following code, $\texttt{t}_1$ and $\texttt{t}_2$ 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:

```
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`:

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

Similarly, for GADT-style definitions [6] such as the following

```
type t = B : t' → t
       | C : t
```

`nonrec` can never be used, since every such definition refers to the definiendum in the return type of each constructor[2].

2.3 Local definitions scoped over several functions

A common pattern involves defining one or more local definitions for use within one more more exported functions[3]. 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`:

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

This style has several drawbacks. First, the names `f` and `g` are separated from their definitions by the definition of `aux`. Second, the unsugared syntax for creating functions `fun x → ...` must be used in place of the more typical sugared syntax `let f x = ...`. Finally, the definition allocates an intermediate tuple. With the extended `open` construct, all of these these problems disappear:

[2] Mantis Issue 6934: *nonrec misbehaves with GADTs* https://caml.inria.fr/mantis/view.php?id=6934

[3] 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, lines 105–117

```
include struct
  open struct let aux x y = ... end
  let f p = aux p true
  let g p = aux p false
end
```

The surrounding `include struct ... end` delimits the scope of the local binding `aux`, so that `aux` is only visible in the definitions of `f` and `g`, not in the code that follows.

2.4 Local exception definitions

OCaml's `let module` construct supports defining exceptions whose names are visible only within a particular expression. For example, in the following code, the `Interrupt` exception is only visible within the body of the `let module ... in` binding:

```
let module M = struct exception Interrupt end in
  let rec loop () = ... raise M.Interrupt
      and run () = match loop () with
    | exception M.Interrupt → Error "failed"
    | x → Ok x
  in run ()
```

Since OCaml 4.04, a construct that supports local exceptions more directly is also available [4]:

```
let exception Interrupt in
  let rec loop () = ... raise Interrupt
      and run () = match loop () with
    | exception Interrupt → Error "failed"
    | x → Ok x
  in run ()
```

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 [7]. (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 [4].

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

```
include struct
  open struct exception Interrupt end
  let rec loop () = ... raise Interrupt
      and run () = match loop () with
    | exception Interrupt → Error "failed"
    | x → Ok x
end
```

As with the previous example, this style of local definition is supported in Standard ML by the `local` construct discussed in Section 6.1.

2.5 Shared state

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

```
open struct
  open struct let counter = ref 0 end
  let inc () = incr counter
  let dec () = decr counter
  let current () = !counter
end
```

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

2.6 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 the deriving framework that generates pretty-printers, serializers, and other functions from type definitions [18].

More recently, the ppx framework, which supports transformations on abstract syntax [17], has become popular. Syntax transformers based on ppx, such as ppx_deriving (a reimplementation of the deriving generic programming framework [18]), js_of_ocaml-ppx (an extension for manipulating JavaScript properties, distributed as part of the js_of_ocaml OCaml-to-JavaScript compiler [16]), ppx_lwt (a syntax for constructing promise computations, part of the lwt lightweight concurrency framework [15]) 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:

```
fun x → [%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:

```
module Staged_349289618 =
struct
  let staged0 hole''_1 =
  let contents''_1 = hole''_1  in
   ...
```

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.

2.7 Restricted open

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

```
open (Option : MONAD)
```

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

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`:

```
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 [8], 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 the 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.

3.1 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:

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

```
type t = T                      type t = T
module M = struct               module M : sig
  type 'a t = S                   type 'a t = S
  let f T = S                     val f : t → t
end                             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 (Section 2.2), 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 the extended `open`.

```
                    type t = T1
type t = T1         open struct type t' = t end
module M = struct   module M : sig
  type t = T2         type t = T2
  let f T1 = T2       val f : t' → t
end                 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, the extended `open` also makes it possible for users to write those signatures that are currently inexpressible.

3.2 Local type aliases in signatures

Even in cases with no shadowing, it is sometimes useful to define a local type alias in a signature[4]. In the following code, the type `t` is available for use in `x` and `y`, but not exported from the signature.

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

4 Restrictions and design considerations

4.1 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

[4]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`

type each type defined in the functor argument [11]. For example, given the following functor definition

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

the following application is valid:

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

and B receives the following type:

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

However, the following application is not allowed:

```
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:

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

Here is the error message from the compiler:

```
1 | open struct type t = T end
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: The type t/89 introduced by this open appears in the signature
       Line 2, characters 4-5:
         The value x has no valid type if t/89 is hidden
```

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 [11].

4.2 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 from 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:

```
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`:

```
# 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` [10]:

```
# 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 less general type than `N`, even though the two modules are semantically equivalent:

```
module M = struct
  open struct type t = T end
  type u = t and v = t
end
```

```
module N = struct
  open struct type t = T end
  type u = t and v = u
end
```

Here are the types assigned by OCaml:

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

```
module N : sig
  type u and v = u
end
```

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.

4.3 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:

```
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:

```
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:

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

5 Implementation sketch

As the discussion in Sections 4.2 and 4.1 indicates, the subtleties in the static semantics of the extended `open` also occur with OCaml's functors. Our implementation takes advantage of this fact, reusing existing functions in OCaml's type checker. In particular, the function `nondep_supertype`

```
val nondep_supertype: Env.t → Ident.t → module_type → module_type
```

is used in the OCaml type checker to eliminate identifiers without paths from the module types that arise from functor applications; we use it a second time to eliminate identifiers without paths from the types of the declarations that follow an occurrence of the extended `open` (Section 4.1). The interested reader may find a fuller description of `nondep_supertype` in Leroy's article on implementing module systems [11].

In more detail, the updated type-checker in our implementation behaves as follows on encountering the phrase `open modexp; decl`. First, `modexp` is type-checked using the function `type_open`, which returns several components: a fresh name for the module of a form that cannot occur in programs (`M#1`, say), a representation of the module type, and a corresponding typing environment. Next, `decl` is type-checked in this extended typing environment. Finally, the type-checking procedure constructs a representation of the extended type-checked program `module M#1 = modexp; open M#1; decl`. This representation is ultimately used to generate code: OCaml's compiler gives modules a run-time representation and an entry in the parent module; this compilation scheme requires that `modexp` has such a representation, too.

Following this step, the `nondep_supertype` function attempts to eliminate the generated identifier `M#1` from the type of `decl`, failing with a user-facing diagnostic if it cannot be eliminated. Finally, the entry for `M#1` is removed from the type of the enclosing module, so that it does not appear in types seen by the user.

The sketch above covers the essence of the implementation. The full patch also supports local open in signatures (Section 3), let bindings, and signatures. The interested reader may find the full details in the GitHub pull request: `https://github.com/ocaml/ocaml/pull/1506`.

6 Alternative designs

The facilities provided by the extended `open` are frequently useful, as the examples in Sections 2 and 3 indicate, and so it is no surprise that other languages provide comparable facilities. This section compares two of these alternatives, based on the keywords `local` and `private`.

6.1 `local`

The design in this paper draws inspiration from Standard ML's `local` construct [13]:

```
local declarations₁
   in declarations₂
end
```

As the keyword suggests, names introduced by the first set of declarations ($\texttt{declarations}_1$) are in scope only within the second set $\texttt{declarations}_2$, not in the code that follows.

The original 1990 Definition of Standard ML [9] also allows `local` in specifications (signatures), making it possible to similarly encode the examples of Section 3. The language defined in the 1997 revision of the Definition [13] no longer allows `local` in specifications. However, they are still supported in the latest release of at least one implementation, Moscow ML [14].

To a first approximation[5], the `local` construct can be defined straightforwardly in terms of `open` as follows:

```
local d1 in d2 end   ⇝   include open struct d1 end d2 end
```

The definition of the extended `open` in terms of `local` is slightly less straightforward:

```
open modexp; d   ⇝   local structure M = modexp; open M in d end
```

(where `M` is not free in `d`)

Unlike the translation from `local` to `open`, this second translation makes use of the surrounding context of the translated expression. First, the declarations `d` following the `open` statement are included on the left hand side of the translation; this makes it possible to delimit the scope of the identifiers imported from `modexp`. Second, and more significantly, the side condition requires that the name `M` introduced on the right hand side of the translation does not appear free in `d`, to avoid shadowing definitions in the surrounding context. In other words, while `local` is *macro expressible* [3] in terms of `open`, `open` is not macro expressible in terms of `local`.

The reader may note the similarity between the translation of `open` into `local` and the elaboration into a program with a freshly generated module name that occurs during type-checking of `open` (Section 5). This generativity appears to be an essential part of the expressiveness enabled by the extended `open`. Unless the type checker is extended to generate fresh names (as in our implementation), the expressive power can only be recovered if an equivalent step is performed by the user (as with the free-variable check with the translation into local).

The translations show, then, that `open` is a little more expressive than `local`. In fact, the extra expressiveness is sometimes useful in practice. Programs that generate code must be careful to avoid name shadowing (Section 2.6). In OCaml, such programs are typically written as transformations on untyped abstract syntax trees, for which it is often not possible to determine whether a variable is free[6]. For such use cases, extended `open` is a little more convenient than `local`.

[5] There are some inessential differences: with Standard ML's `local`, type names in $\texttt{declarations}_1$ that cannot be eliminated in the types of $\texttt{declarations}_2$ become abstract, while the corresponding situation with `open` is treated as an error in our proposal (Section 4.1).

[6] For example, in the expression `let open M in x + y` whether `x` and `y` are free depends on whether `M` exports those identifiers — that is, it depends upon the type of `M`.

6.2 `private`

Many object-oriented languages use a `private` keyword to mark non-exporting declarations. Indeed, the object oriented part of early versions of OCaml supported `private` instance variables in classes with this meaning:

```
class c = object val private x = 3 end
```

However, for the last two decades[7] only private methods, not private instance variables are supported, and `private` has a meaning closer to `protected` in other object-oriented languages, limiting scope to the current class and its sub-classes.

As with `local`, it would be possible to support the examples in Sections 2 and 3 by adding support for `private` annotations on declarations in structures and signatures. However, supporting `private` annotations introduces additional syntactic considerations. In particular, it is natural to extend `open` to allow arbitrary module expressions (since every form of module expression — functor application, unnamed structure, ascription, etc. — is potentially useful as the argument to `open`), but it is less natural to support `private` annotations on every type of declaration. For example, while private type aliases in signatures are clearly useful (Section 3.2), there do not appear to be any uses for private exception declarations in signatures. A design based around `private` therefore appears to bring a choice between a uniform but loose grammar (i.e. with support for various useless constructs), or a complicated grammar that allows `private` only for constructs where it is useful.

As with `local`, it is possible to define `private` in terms of the extended `open`:

```
private decl  ⇝  open struct decl end
```

Once again, the definition of `open` in terms of `private` is a little less straightforward:

```
open modexp; decl  ⇝  private module M = modexp; open M; decl
```
(where `M` is not free in `decl`)

And, as with the translation from `open` into `local`, the translation from `open` into `private` involves determining the set of free identifiers in the declarations that follow, making `private` a less suitable basis than `open` for code generation involving non-exporting declarations (Section 2.6).

6.3 Signature-local bindings

An alternative approach to supporting the use cases of Section 3 builds on another OCaml feature, *destructive substitution* [5]. Destructive substitution is an operation on signatures that simultaneously eliminates a type (or module) component within a signature and replaces each use of the component with a compatible type (or module) expression.

For example, here is a definition for a module type `T` that defines a type component `t` and a value component `f` whose type uses `t`:

```
module type T = sig type t val f : t → t end
```

[7] The following updates to the OCaml compiler and manual removed private instance variables and introduced private methods with the current semantics:

- Jérôme Vouillon (June 24, 1998): *Nouvelle syntaxe des classes*
  `https://github.com/ocaml/ocaml/commit/87b17301`
- Jérôme Vouillon (August 13, 1998):*Mise a jour des classes*,
  `https://github.com/ocaml-manual/commit/63bea030`

The following code defines a module type `S` by eliminating `t` and replacing each occurrence of `t` with `int` in the remainder of the module:

```
module type S = T with type t := int
```

The result of this destructive substitution is equivalent to the following direct definition of `S`:

```
module type S = sig val f : int → int end
```

Jacques Garrigue has proposed extending destructive substitution to local aliases in signatures, so that a definition of the following form

```
type t := e
```

in a signature would behave equivalently to the following extended `open` code

```
open struct type t = e end
```

i.e. each occurrence of `t` in the remainder of the signature would be replaced by `e`.

This design supports the signature use cases in Section 3. For example, using local aliases, the module on the left below can be given the signature on the right:

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

```
type t = T1
type t' := t
module M : sig
  type t = T2
  val f : t' → t
end
```

Furthermore, like destructive substitution itself, this extended design is restricted to module and type aliases, and so the syntactic concerns with `private` (Section 6.2) do not arise.

7 Status

A variant of the design proposed in this article was discussed at the Caml developers meeting and accepted for inclusion into OCaml 4.08. The subsequent GitHub pull request and further discussion may be found at the following URL:

https://github.com/ocaml/ocaml/pull/1506

The design for extended `open` in structures has been incorporated directly, and so the use cases of Section 2 can be used as written in OCaml 4.08.

Furthermore, `open` in signatures has been extended beyond simple paths to support functor application (e.g. `open F(X)`), and it is anticipated that it will eventually be further extended to support transparent ascription (Section 2.7) and structures containing only aliases (e.g. `open struct type t = int end`).

However, OCaml 4.08 does not support arbitrary module expressions as the arguments of `open` in signature contexts, so the examples of Section 3 cannot be written directly. Instead, the release also adds support for signature-local bindings (Section 6.3), which covers those use cases.

Acknowledgments

We thank Leo White and the OCaml'17 workshop and post-proceedings reviewers for comments and suggestions, and Alain Frisch and Thomas Refis for help with the implementation.

References

[1] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop & Anil Madhavapeddy (2015): *Effective Concurrency through Algebraic Effects*. OCaml Users and Developers Workshop 2015.

[2] Derek Dreyer (2005): *Understanding and Evolving the ML Module System*. Ph.D. thesis, CMU. Published as technical report CMU-CS-05-131.

[3] Matthias Felleisen (1991): *On the expressive power of programming languages*. Science of Computer Programming 17(1), pp. 35 – 75, doi:10.1016/0167-6423(91)90036-W.

[4] Alain Frisch (2016): *Pull request: Turn local exceptions into jumps*. `https://github.com/ocaml/ocaml/pull/638`.

[5] Alain Frisch & Jacques Garrigue (2010): *First-class modules and composable signatures in Objective Caml 3.12*. ACM SIGPLAN Workshop on ML, Baltimore, MD.

[6] Jacques Garrigue & Didier Rémy (2013): *Ambivalent Types for Principal Type Inference with GADTs*. In Chung-chieh Shan, editor: Programming Languages and Systems, Springer International Publishing, Cham, pp. 257–272, doi:10.1007/978-3-319-03542-0_19.

[7] Robert Harper (2012): *Exceptions are shared secrets*. `https://existentialtype.wordpress.com/2012/12/03/exceptions-are-shared-secrets/`.

[8] Robert Harper & Mark Lillibridge (1994): *A Type-theoretic Approach to Higher-order Modules with Sharing*. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '94, ACM, New York, NY, USA, pp. 123–137, doi:10.1145/174675.176927.

[9] Robert Harper, Robin Milner & Mads Tofte (1990): *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[10] Xavier Leroy (1995): *Applicative Functors and Fully Transparent Higher-order Modules*. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95, ACM, New York, NY, USA, pp. 142–153, doi:10.1145/199448.199476.

[11] Xavier Leroy (2000): *A modular module system*. Journal of Functional Programming 10(3), pp. 269–303, doi:10.1017/S0956796800003683.

[12] Runhang Li & Jeremy Yallop (2017): *Extending OCaml's* `open`. OCaml Users and Developers Workshop.

[13] Robin Milner, Mads Tofte & David Macqueen (1997): *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, doi:10.7551/mitpress/2319.001.0001.

[14] Sergei Romanenko, Claudio Russo & Peter Sestoft (2000): *Moscow ML Language Overview*, version 2.00 edition. Available at `http://mosml.org/mosmlref.pdf`.

[15] Jérôme Vouillon (2008): *Lwt: A Cooperative Thread Library*. In: Proceedings of the 2008 ACM SIGPLAN Workshop on ML, ML '08, ACM, New York, NY, USA, pp. 3–12, doi:10.1145/1411304.1411307.

[16] Jérôme Vouillon & Vincent Balat (2013): *From Bytecode to JavaScript: the Js_of_ocaml Compiler*. Software: Practice and Experience, doi:10.1002/spe.2187.

[17] Leo White (2013): *Extension Points for OCaml*. OCaml Users and Developers Workshop.

[18] Jeremy Yallop (2007): *Practical Generic Programming in OCaml*. In: Proceedings of the 2007 Workshop on Workshop on ML, ML '07, ACM, New York, NY, USA, pp. 83–94, doi:10.1145/1292535.1292548.