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 extend-
ing 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 pro-
grams.

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 applica-
tion, 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 lan-
guage (Section 3). We touch briefly on restrictions and other design considerations (Section 4)
before sketching the implementation (Section 5) and comparing some alternative designs (Sec-
ction 6).
Extending OCaml’s open

Current design:
Only basic paths are allowed

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

\begin{align*}
\text{open M.N} & \quad \text{open M.N} \quad \text{open F(M)} \quad \text{open (M:S)} \\
\text{open struct ... end} & \quad \text{open struct ... end}
\end{align*}

Figure 1: The open construct and our proposed extension

1.1 Status
Following the presentation of this proposal at the OCaml 2017 workshop [11], the design was discussed at the Caml developers meeting and accepted for inclusion into OCaml. The subsequent GitHub pull request and further discussion may be found at the following URL:

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

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.

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:

\begin{verbatim}
open struct let x = 3 end    (* no entry for x *)
let y = x
val y : int
\end{verbatim}

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:

\begin{verbatim}
type s := A
module M = struct
type t = B of s | C
end
\end{verbatim}

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 \( t \):

\[1\] 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:

\[
\text{type } t = A \\
\text{module } M = \text{struct} \\
\quad \text{type nonrec } t = B \text{ of } t | C \\
\end{array}
\]

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, \( t_1 \) 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:

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

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

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

Similarly, for GADT-style definitions \(^5\) such as the following

\[
\text{type } t = B : t' \to t \\
\quad | C : t
\]

`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 or 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:

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


\(^3\) See `draw_poly`, `draw_poly_line` and `dodraw` in the OCaml `Graphics` module for an example. [https://github.com/ocaml/ocaml/blob/4697ca16/otherlibs/graph/graphics.ml](https://github.com/ocaml/ocaml/blob/4697ca16/otherlibs/graph/graphics.ml) lines 105–117
This style has several drawbacks. First, the names \( f \) and \( g \) are separated from their definitions by the definition of \( \text{aux} \). Second, the unsugared syntax for creating functions \( \text{fun} \ x \rightarrow \ldots \) must be used in place of the more typical sugared syntax \( \text{let} \ f \ x = \ldots \). Finally, the definition allocates an intermediate tuple. With the extended \textit{open} construct, all of these problems disappear:

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

The surrounding \textit{include struct} ... \textit{end} delimits the scope of the local binding \( \text{aux} \), so that \( \text{aux} \) is only visible in the definitions of \( f \) and \( g \), not in the code that follows.

2.4 Local exception definitions

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

\[
\texttt{let module } M = \texttt{struct exception Interrupt end in}
\begin{align*}
\texttt{let } \text{rec } \text{loop } () & = \ldots \texttt{raise M.Interrupt}\; \\
\texttt{and run } () & = \texttt{match loop } () \texttt{ with}\; \\
& | \texttt{exception M.Interrupt } \rightarrow \texttt{Error "failed"}\; \\
& | \texttt{x } \rightarrow \texttt{Ok x}
\end{align*}
\]

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

\[
\texttt{let } \text{exception Interrupt in}
\begin{align*}
\texttt{let } \text{rec } \text{loop } () & = \ldots \texttt{raise Interrupt}\; \\
\texttt{and run } () & = \texttt{match loop } () \texttt{ with}\; \\
& | \texttt{exception Interrupt } \rightarrow \texttt{Error "failed"}\; \\
& | \texttt{x } \rightarrow \texttt{Ok x}
\end{align*}
\]

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 [6]. (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 \textit{open} construct improves support for this pattern. While \texttt{let module} allows defining exceptions whose names are visible only within particular expressions, the extended \textit{open} also allows limiting visibility to particular declarations. In the following snippet, the \texttt{Interrupt} exception is only visible in the definitions of \texttt{loop} and \texttt{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 trans-
formations 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 [17].

More recently, the ppx framework, which supports transformations on abstract syntax [16],
has become popular. Syntax transformers based on ppx, such as ppx_deriving (a reimple-
mentation of the deriving generic programming framework [17]), js_of_ocaml-ppx (an extension
for manipulating JavaScript properties, distributed as part of the js_of_ocaml OCaml-
to-JavaScript compiler [15]), ppx_lwt (a syntax for constructing promise computations, part
of the lwt lightweight concurrency framework [14]) and ppx_stage (a preprocessor for typed
multi-stage programming), may also generate large amounts of code.

To avoid conflicts with programmer-defined names, ppx transforms often introduce completely
anonymous declarations. 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:

```ocaml
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

```ocaml
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`:

```ocaml
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 [7], 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:
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:

\[
\begin{align*}
\text{type } t &= T \\
\text{module } M &= \text{struct} \\
\text{type } t' &= S \\
\text{let } f T &= S \\
\text{end}
\end{align*}
\]

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 \texttt{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 type each type defined in the functor argument \([10]\). 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 \texttt{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 = int 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#0.t
```

Since the restriction for the extended \texttt{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 \texttt{Mtype.nondep_supertype} function to check if introduced identifiers can be eliminated from rest of the structure \([10]\).

---

\(^4\) For example, the functions \texttt{comment}, \texttt{maintainer}, \texttt{run}, \texttt{cmd}, \texttt{user}, \texttt{workdir}, \texttt{volume}, and \texttt{entrypoint} in the \texttt{Dockerfile} module would benefit from such an alias. [https://github.com/avsm/ocaml-dockerfile/blob/e0dad1a/src/dockerfile.mli](https://github.com/avsm/ocaml-dockerfile/blob/e0dad1a/src/dockerfile.mli)
4.2 The Avoidance Problem

The avoidance problem \cite{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 \cite{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:

\begin{verbatim}
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
\end{verbatim}

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:

\begin{verbatim}
# 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
\end{verbatim}

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 $G$: \cite{9}:

\begin{verbatim}
# 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
\end{verbatim}

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:

\begin{verbatim}
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
\end{verbatim}

\footnote{A reviewer asks whether OCaml really requires two sets of parentheses here. Surprisingly, it does: the outer pair encloses the application argument, and the inner pair encloses the signature ascription.}
Here are the types assigned by OCaml:

```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:

```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! *)
```

### 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`

```ocaml
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 [10].

In more detail, the updated typechecker 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 [12]:

local declarations₁ in declarations₂ end

As the keyword suggests, names introduced by the first set of declarations (declarations₁) are in scope only within the second set declarations₂, not in the code that follows.

The original 1990 Definition of Standard ML [8] 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 [12] no longer allows local in specifications. However, they are still supported in the latest release of at least one implementation, Moscow ML [13].

To a first approximation, 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

(whence M is not free in d)

---

6 There are some inessential differences: with Standard ML’s local, type names in declarations₁ that cannot be eliminated in the types of declarations₂ become abstract, while the corresponding situation with open is treated as an error in our proposal (Section 4.1).
Unlike the translation from \texttt{local} to \texttt{open}, this second translation makes use of the surrounding context of the translated expression. First, the declarations $d$ following the \texttt{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 \texttt{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 \texttt{local} is \textit{macro expressible} \cite{OCaml2018} in terms of \texttt{open}, \texttt{open} is not macro expressible in terms of \texttt{local}.

The reader may note the similarity between the translation of \texttt{open} into \texttt{local} and the elaboration into a program with a freshly generated module name that occurs during type-checking of \texttt{open} (Section \ref{sec:type-checking}). This generativity appears to be an essential part of the expressiveness enabled by the extended \texttt{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 \texttt{local}).

The translations show, then, that \texttt{open} is a little more expressive than \texttt{local}. In fact, the extra expressiveness is sometimes useful in practice. Programs that generate code must be careful to avoid name shadowing (Section \ref{sec:type-abstract}). 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\footnote{For example, in the expression \texttt{let open M in} \texttt{x + y} whether \texttt{x} and \texttt{y} are free depends on whether \texttt{M} exports those identifiers — that is, it depends upon the type of \texttt{M}.}. For such use cases, extended \texttt{open} is a little more convenient than \texttt{local}.

\subsection{private}

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

\begin{verbatim}
class c = object val private x = 3 end
\end{verbatim}

However, for the last two decades\footnote{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, \url{https://github.com/ocaml/ocaml/commit/87b17301} Jérôme Vouillon (August 13, 1998): Mise à jour des classes, \url{https://github.com/ocaml/ocaml-manual/commit/63bea030}} only private methods, not private instance variables are supported, and \texttt{private} has a meaning closer to \texttt{protected} in other object-oriented languages, limiting scope to the current class and its sub-classes.

As with \texttt{local}, it would be possible to support the examples in Sections \ref{sec:local} and \ref{sec:open-definitions} by adding support for \texttt{private} annotations on declarations in structures and signatures. However, supporting \texttt{private} annotations introduces additional syntactic considerations. In particular, it is natural to extend \texttt{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 \texttt{open}), but it is less natural to support \texttt{private} annotations on every type of declaration. For example, while private type aliases in signatures are clearly useful (Section \ref{sec:private-types}), there do not appear to be any uses for private exception declarations in signatures. A design based around \texttt{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).

Acknowledgments

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

References


