Type-level module aliases: independent and equal

Jacques Garrigue
Nagoya University

Leo P. White
Cambridge University

Type-level module alias

- A very simple feature,
- which naturally complements applicative functors,
- and reconciles the module-as-namespace approach with separate compilation.
- Available in OCaml 4.02 (released last week).

The definition

Allow aliases in signatures:

```
module S = String
```

in a signature means that `S` is an alias for `String`,
i.e. it will be expanded to `String` when used.

This can be understood as a singleton type:

```
module S : (module String)
```

The typing rules

– Infer type-level module aliases

```
# module M = P ;;
module M = P
```

$$\frac{\Gamma \vdash P \text{ well formed}}{\Gamma \vdash P : (\texttt{module}\ P)}(*)$$

– Extend the subtyping relation

$$\frac{\Gamma(P) = S}{\Gamma \vdash (\texttt{module}\ P) <: S}$$

Here P is a module path.

$(*)$ Only if P appears as rhs of a binding.

The origin

– The concept of type-level module alias appeared first in Traviatta [Nakata&G, ICFP 2006].

– Used to allow type inference of recursive modules.

```
module rec Tree = struct
  module F = Forest
  ...
end
and Forest  = struct ... Tree ... end
```

Here we do not know yet the type of `Forest` when we typecheck the binding of `F`, so we handle it as an alias.

The discovery

Later, we discovered that type-level module aliases were a good match for OCaml-style applicative functors.

```
module S = String
module SSet = Set.Make(S)
module StringSet = Set.Make(String)
let f (x : StringSet.t) = (x : SSet.t)
```

The last statement fails in OCaml before 4.01, but succeeds with type-level module aliases.

The application

- Helping applicative functors was probably not enough to justify a new feature.

- However, remember that the original goal was to simplify program analysis.

- By simplifying the typechecking of aliases, type-level module aliases allow to remove dependencies,

- which in turns allows to use them to construct flexible hierarchical namespaces.

Modules as namespaces

- The ML module system is very powerful.
- Through nested structures, it allows for hierarchical design of libraries.
- Sharing of types allows grafting a module somewhere else in the hierarchy.
- ML modules: the ultimate namespace design?

The broken hierarchy: OCaml

This ideal view only applies in theory.

– For separate compilation, root modules, aka compilation units, are mapped to files.

– Libraries are just forests of modules, and using several libraries simultaneously mixes their modules.

→ Risk of name conflicts (breaks linking).

– The -pack command allows to turn a library into a module with submodules, but it is monolithic.

→ Using modules as namespaces creates large interfaces and binaries. (e.g. Jane Street's Core library)

The re-built hierarchy: SML/NJ

– SML/NJ avoids many of these problems,

– but this is thanks to an external mechanism:
the Compilation Manager.

– Namespaces are declared in special files, using a dedicated syntax.

– An essential part of the language falls out of the specification.

A packed library

This library contains two units: `mylibA.ml` and `mylibB.ml`, and a wrapper `mylib.ml`.

```
Mylib                          Interface
 module A = MylibA   :          module A : sig ... end
 module B = MylibB              module B : sig ... end
```

Can be used comfortably with `open`.

```
open Mylib
let x = A.f 3
```

However, as separate compilation only allows to see the interface, this program links `Mylib`, `MylibA` and `MylibB`.

Using type-level module aliases

Thanks to type-level module aliases, the typing changes:

```
Mylib                      Interface
 module A = MylibA    :     module A = MylibA
 module B = MylibB          module B = MylibB
```

As a result, references to `Mylib.A` can be expanded to `MylibA`, on the client side.

```
open Mylib
let x = A.f 3
```

This program now just requires `MylibA`, like if we had written `MylibA.f` in place of `A.f`.

Induced dependencies

For backward compatibility reasons, a new compilation flag `-no-alias-deps` enables refined dependencies.

Here are the dependencies for the previous example:

Link/Compile-time deps	`Mylib`	`MylibA`	`MylibB`
`Mylib` (default)	—	√	√
`Mylib` (`-no-alias-deps`)	—	—	—
`Client` (default)	√	√	√/—*
`Client` (`-no-alias-deps`)	ct	√	—

(*) depends on whether `Mylib` was compiled with `-no-alias-deps`.
(ct) only required at compile time.

Application to build libraries

One can avoid monolithic packing by using the following recipe:

1. Create a mapping unit whose role is only to map short names to prefixed names, for all member units.
2. Open this unit in all members, so that one can use short names inside them.
3. Create an export unit, which again maps short names to prefixed names, but may choose to omit some internal modules.

Using this approach, `MylibB` can refer to `MylibA` just as `A`.

Library example

```
Mylib
 module A = MylibA        (* does not require MylibA *)
 module B = MylibB        (* does not require MylibB *)
MylibA
 open Mylib               (* compile-time dependency *)
 let f x = x+1
MylibB
 open Mylib               (* compile-time dependency *)
 let g x = (A.f x) * 2            (* requires MylibA *)
```

`Mylib` needs to be compiled first, but this is fine as it has no dependency at all on `MylibA` and `MylibB`.

Ease of use

Compared to the `-pack` command, which completely hides the original files, this approach requires to

- rename the source files to add a unique prefix
- add an open statement at the top of each file

In order to smooth transition, this can be done through command-line options:

```
ocamlopt -no-alias-deps -open Mylib -o mylibA.cmx a.ml
```

Performance

We have no complete benchmark, but empirical evidence on the Core/Async libraries shows that

- just using the new compiler divides the size of compiled interfaces by 3, which speeds up compilation too,
- using `-no-alias-deps` reduces the size of executables by 2 (up to 10 in some cases).

Limitations and future work

Currently, type-level module aliases can be created only for a limited subset of module paths.

The following are excluded:

- Functor applications
- Opaque coercions
- Functor arguments
- Recursive modules

While this is sufficient for the application to namespaces, in the future we would like to support these cases to improve the use of applicative functors.

PR#4049

Aside of performance, some "design" bugs of applicative functors are solved.

```
module A = struct
  module B = struct type t let compare x y = 0 end
  module S = Set.Make(B)
  let empty = S.empty
end
module A1 = A;;
A1.empty = A.empty;;
```

In this program, the last line was causing a type error, but is now fixed by type-level module aliases.

PR#3476

```
module FF(X : sig end) = struct type t end
module M = struct
  module X = struct end
  module Y = FF (X)                                (* XXX *)
  type t = Y.t
end
module F (Y : sig type t end)
         (M : sig type t = Y.t end) = struct end
module N = F (M.Y) (M);;
```

In this program the last line fails, but the required equality involves paths containing functor applications.