A Proposal for Non-intrusive Namespaces

OCaml 2014

Pierrick COUDERC (INRIA), Fabrice LE FESSANT (INRIA & OCamlPro), Benjamin CANOU (OCamlPro), Pierre CHAMBART (OCamlPro)

September 5, 2014

Problem: using modules of the same name

- LibA (Misc, Map, AnotherModule, ...)
- My program (Misc, Env, ...)

Problem: using modules of the same name

- LibA (Misc, Map, AnotherModule, ...)
- My program (Misc, Env, ...)

ocamlopt : *"Error: Files libA/misc.cmx and misc.cmx both define a module named Misc"*

Problem: using modules of the same name

- LibA (Misc, Map, ...)
- My program (Misc, Env, ...)

Problem: using modules of the same name

- LibA (Misc, Map, ...)
- My program (MyMisc, Env, ...)

Problem: using modules of the same name

- LibA (Misc, Map, ...)
- My program (MyMisc, Env, ...)

ocamlopt : *"Error: Files libA/anotherModule.cmx and env.cmx make inconsistent assumptions over interface Map"*

Problem: using modules of the same name

- LibA (Misc, Map, ...)
- My program (MyMisc, Env, ...)

ocamlopt : *"Error: Files libA/anotherModule.cmx and env.cmx make inconsistent assumptions over interface Map"*

$\rightarrow$ stdlib/Map no longer usable

Problem: using modules of the same name

What now?

- Ask LibA dev to rename Map?
- Copy stdlib/map.ml sources? (bad idea)
- Abandon stdlib?

Problem: using modules of the same name

What now?

- Ask LibA dev to rename Map?
- Copy stdlib/map.ml sources? (bad idea)
- Abandon stdlib?

Problem: using modules of the same name

What now?

- Ask LibA dev to rename Map?
- Copy stdlib/map.ml sources? (bad idea)
- Abandon stdlib?

Before 4.02: developers' trick #1

Common practice: long names

Before 4.02: developers' trick #1

Common practice: long names

- LibA (Misc, Map, ...)
- → LibA (LibA_Misc, LibA_Map, ...)

Before 4.02: developers' trick #1

Common practice: long names

- LibA (Misc, Map, ...)
- $\rightarrow$ LibA (LibA_Misc, LibA_Map, ...)

Before 4.02: developers' trick #1

Common practice: long names

- LibA (Misc, Map, ...)
- → LibA (LibA_Misc, LibA_Map, ...)

Long names can be quite long...

Before 4.02: developers' trick #2

Packs

Before 4.02: developers' trick #2

Packs

- LibA (Misc, Map, ...)
- → LibA = a unique module LibA with submodules: (Misc, Map, ...)

Before 4.02: developers' trick #2

Packs

- LibA (Misc, Map, ...)
- $\rightarrow$ LibA = a unique module LibA with submodules: (Misc, Map, ...)

Before 4.02: developers' trick #2

Packs

Pros:

- **Developer**: No code change, simply a matter of options;
- **User**:
  - Use path to distinguish modules
  - Use the module as any other module

Before 4.02: developers' trick #2

Packs

Pros:

- **Developer**: No code change, simply a matter of options;
- **User**:
  - Use path to distinguish modules
  - Use the module as any other module

Before 4.02: developers' trick #2

Packs

Pros:

- **Developer**: No code change, simply a matter of options;
- **User**:
  - Use path to distinguish modules
  - Use the module as any other module

Cons:

- Too much recompilations
- Dependencies
- Size of executables really large

Before 4.02: developers' trick #2

Packs

Pros:

- **Developer**: No code change, simply a matter of options;
- **User**:
  - Use path to distinguish modules
  - Use the module as any other module

Cons:

- Too much recompilations
- Dependencies
- Size of executables really large

Before 4.02: developers' trick #2

Packs

Pros:

- **Developer**: No code change, simply a matter of options;
- **User**:
  - Use path to distinguish modules
  - Use the module as any other module

Cons:

- Too much recompilations
- Dependencies
- Size of executables really large

Since 4.02: developers' trick #3

Module aliases

Since 4.02: developers' trick #3

Module aliases

- LibA (Misc, Map, ...)
- $\rightarrow$ LibA $=$ (LibA_Misc, LibA_Map, ...) $+$ LibA

```
(* libA.ml *)
module Misc = LibA_Misc
module Map = LibA_Map
...
```

compiled with **-no-alias-deps**

Since 4.02: developers' trick #3

Module aliases

- LibA (Misc, Map, ...)
- → LibA = (LibA_Misc, LibA_Map, ...) + LibA

```
(* libA.ml *)
module Misc = LibA_Misc
module Map = LibA_Map
...
```

compiled with **-no-alias-deps**

Since 4.02: developers' trick #3

Module aliases

- LibA (Misc, Map, ...)
- $\rightarrow$ LibA = (LibA_Misc, LibA_Map, ...) + LibA

```
(* libA.ml *)
module Misc = LibA_Misc
module Map = LibA_Map
...
```

compiled with -**no-alias-deps**

Using advantages of aliases

Using advantages of aliases

Developer POV: using short names

```
(* libA_misc.ml *)
open LibA
...

(* libA_map.ml *)
open LibA

open Misc
...
```

Using advantages of aliases

Developer POV: using short names

```
(* libA_misc.ml *)
open LibA
...

(* libA_map.ml *)
open LibA

open Misc
...
```

→ False circularity

Using advantages of aliases

- Deceive ocamldep for better dependencies

```
(* map.ml *)
open Misc
...
```

- + Namespace used transparently
  →
  ocamlc -c -o libA_Map.cmo -open LibA map.ml

Using advantages of aliases

- Deceive ocamldep for better dependencies

```
(* map.ml *)
open Misc
...
```

- + Namespace used transparently
  $\rightarrow$
  ocamlc -c -o libA_Map.cmo -open LibA map.ml

Current solutions

- Dependencies issues
- Build system sometimes complex
- No extensibility

Current solutions

- Dependencies issues
- Build system sometimes complex
- No extensibility

Current solutions

- Dependencies issues
- Build system sometimes complex
- No extensibility

Our solution: *namespaces*.

Principle:

- A module belongs to a namespace
- Modules in namespaces are imported explicitely

Our solution: *namespaces*.

Principle:

- A module belongs to a namespace
- Modules in namespaces are imported explicitely

Our solution: *namespaces*.

Principle:

- A module belongs to a namespace
- Modules in namespaces are imported explicitely

To compare with Java or Scala's packages.

Writing LibA with namespaces

What happens to LibA?

```
(* misc.ml *)
in namespace LibA
...
```

```
(* map.ml *)
in namespace LibA
...
```

Writing LibA with namespaces

What happens to LibA?

```
(* misc.ml *)
in namespace LibA
...
```

```
(* map.ml *)
in namespace LibA
...
```

Using our library

How to use those modules in my program?

Using our library

```
in namespace MyNamespace
with MyNamespace.Misc
and LibA.(Misc, Map)

open Misc (* from LibA *)
let empty = Map.empty
...
let _ = ...
  Misc.pprint 42 (* function only in my own Misc *)
```

Using our library

```
in namespace MyNamespace
with MyNamespace.Misc
and LibA.(Misc, Map)

open Misc (* from LibA *)
let empty = Map.empty
...
let _ = ...
  Misc.pprint 42 (* function only in my own Misc *)
```

ocamlc: *“Unbound value Misc.pprint”*

Using our library

We need Misc and “Misc from LibA” at the same time:

```
in namespace MyNamespace
with MyNamespace.Misc
and LibA.(Misc as Misc2, Map)

open Misc2
let empty = Map.empty
...
let _ = ...
  Misc.pprint 42
```

Using all modules

We need all the modules.

Using all modules

We need all the modules.

```
in namespace MyNamespace
with LibA._

open Misc (* from LibA *)
...
```

Not recommended: it shadows existing names

Using all modules

We need all the modules.

```
in namespace MyNamespace
with LibA._

open Misc (* from LibA *)
...
```

Not recommended: it shadows existing names

Using almost all the modules

We need all the modules, except this one.

Using almost all the modules

We need all the modules, except this one.

```
in namespace MyNamespace
with MyNamespace._
and LibA.(Misc as _, _)
...
open Misc (* not the Misc from LibA *)
```

- Misc is not imported from LibA (no dependency)
- Still not recommended.

Using almost all the modules

We need all the modules, except this one.

```
in namespace MyNamespace
with MyNamespace._
and LibA.(Misc as _, _)
...
open Misc (* not the Misc from LibA *)
```

- Misc is not imported from LibA (no dependency)
- Still not recommended.

Naming conflict

If I imports two modules with the same name?

Naming conflict

If I imports two modules with the same name?

```
in namespace MyNamespace
with Stdlib._
and LibA._
...
```

Naming conflict

If I imports two modules with the same name?

```
in namespace MyNamespace
with Stdlib._
and LibA._
...
```

ocamlc: *"The module Map from LibA will shadow one previously imported"*

Extensibility

Namespaces are not closed.

Extensibility

Namespaces are not closed.

```
in namespace LibA
...
```

Adding a module in a namespace *a posteriori* is possible

Extending Pervasives to namespaces

Pervasives: automatically opened.

Extending Pervasives to namespaces

Pervasives: automatically opened.

```
in namespace MyNamespace
with Stdlib.List
...
let empty = [] (* from Stdlib.Pervasives *)
```

$\rightarrow$ Pervasives modules automatically opened when using their namespace.

Extending Pervasives to namespaces

Preventing auto-opens:

- By renaming:

```
with Stdlib.(Pervasives as P)
```

- And shadowing:

```
with Stdlib.(Pervasives as _)
```

Extending Pervasives to namespaces

Preventing auto-opens:

- By renaming:

```
with Stdlib.(Pervasives as P)
```

- And shadowing:

```
with Stdlib.(Pervasives as _)
```

Using hierarchies organization

Namespaces: natural way to organize modules.

Stdlib could be organized and used like this:

```
with Stdlib.Internals.CamlinternalFormats
and Stdlib.Unsafe.Obj
...
```

Using hierarchies organization

Namespaces: natural way to organize modules.

Stdlib could be organized and used like this:

```
with Stdlib.Internals.CamlinternalFormats
and Stdlib.Unsafe.Obj
...
```

Mapping namespaces and filesystem

Technically: a namespace $\Rightarrow$ a directory

- Logical: distinguish compilation units of the same name
- Practical: automatic organization
- In the future: would allow simpler compilation (-make)

Mapping namespaces and filesystem

Technically: a namespace ⇒ a directory

- Logical: distinguish compilation units of the same name
- Practical: automatic organization
- In the future: would allow simpler compilation (-make)

Mapping namespaces and filesystem

Technically: a namespace $\Rightarrow$ a directory

- Logical: distinguish compilation units of the same name
- Practical: automatic organization
- In the future: would allow simpler compilation (-make)

Mapping namespaces and filesystem

Technically: a namespace $\Rightarrow$ a directory

- Logical: distinguish compilation units of the same name
- Practical: automatic organization
- In the future: would allow simpler compilation (-make)

ocamldep and namespaces

Namespace declaration and imports $\equiv$ header

Dependencies computed easier: each import **obviously is** a file.

With a large adoption and use of namespace:
$\rightarrow$ ocamldep only needs to read the header.

Formal and technical aspect

Namespaces, especially imports:
$\rightarrow$ Description of the compilation environnement

Compiler-side: not too invasive

- Symbols extended to contain the namespace
- Env extended to use and propagate namespaces

Formal and technical aspect

Namespaces, especially imports:
$\rightarrow$ Description of the compilation environnement

Compiler-side: not too invasive

- Symbols extended to contain the namespace
- Env extended to use and propagate namespaces

Comparing our proposal with module aliases

- + Extensibility
- + Simple build system
- + Better dependencies
- + Expressivity
- - New syntax → code not compatible with old compilers

Comparing our proposal with module aliases

- + Extensibility
- + Simple build system
- + Better dependencies
- + Expressivity
- - New syntax → code not compatible with old compilers

Comparing our proposal with module aliases

- + Extensibility
- + Simple build system
- + Better dependencies
- + Expressivity
- - New syntax → code not compatible with old compilers

Comparing our proposal with module aliases

- + Extensibility
- + Simple build system
- + Better dependencies
- + Expressivity
- - New syntax $\rightarrow$ code not compatible with old compilers

Comparing our proposal with module aliases

- + Extensibility
- + Simple build system
- + Better dependencies
- + Expressivity
- - New syntax → code not compatible with old compilers

Works in progress: coercion to module

Transforming the header into modules.

```
with LibA.(Misc, Map)
and Stdlib.(List, String, Map)
```

$\Rightarrow$

```
module LibA = struct
  module Misc = ...
  module Map = ...
end
module Stdlib = struct
  module List = ...
  module String = ...
  module Map = ...
end
```

Works in progress: coercion to module

Transforming the header into modules.

```
with LibA.(Misc, Map)
and Stdlib.(List, String, Map)
```

$$\Rightarrow$$

```
module LibA = struct
  module Misc = ...
  module Map = ...
end
module Stdlib = struct
  module List = ...
  module String = ...
  module Map = ...
end
```

Work in progress: big functors

Primary idea: using packs to generate functors (Fabrice Le Fessant, for OCaml 3.12)

Example: Cohttp $\rightarrow$ uses functors massively to use Lwt and Async.

$\Rightarrow$ Generating automatically functors and applications on entire namespaces.

Highly experimental, design choices to do and change.

Conclusion

- Mechanism of namespaces integrated in the language
- Solves compilation issues, can help tools for dependencies
- Working prototype on 4.02:
  `github.com/pcouderc/ocaml_namespaces`