# A Proposal for Non-Intrusive Namespaces in OCaml

Pierrick Couderc
INRIA
pierrick.couderc@inria.fr

Benjamin Canou
OCamlPro
benjamin.canou@ocamlpro.com

Pierre Chambart
OCamlPro
pierre.chambart@ocamlpro.com

Fabrice Le Fessant
INRIA & OCamlPro
fabrice.le_fessant@inria.fr

## Abstract

We present a work-in-progress about adding namespaces to OCaml. Inspired by other languages such as Scala or C++, our aim is to design and formalize a simple and non-intrusive namespace mechanism without complexifying the core language. Namespaces in our approach are a simple way to define libraries while avoiding name clashes. They are also meant to simplify the build process, clarifying and reducing (to zero whenever possible) the responsibility of external tools.

## 1 Introduction

Packages, or namespaces, are a standard feature in many languages. They allow to distinguish multiple names into distinct entities, allowing especially names that could have overlapped otherwise. They are also a canonical way to describe libraries inside the language, without letting the build system deal with such an issue. It makes the program easier to understand by simply looking at the source code (and not its building process).

OCaml's modules can be seen as a namespace component, but without the expressiveness of Scala's packages[3], or the ability for the user to extend existing namespaces like in C++[1]. Moreover, the current way in OCaml to avoid modules' name clash at usage is either by prefixing them to get a name long enough to make conflicts unlikely to occur, or by *packing*, making them submodules of a big one. The recent work on *module aliases* to avoid module copy in packs strenghten the need for a dedicated namespace mechanism in the language.

In this talk, we are going to present our work-in-progress on namespaces and the implementation in OCaml.

## 2 Objectives

Namespaces are a mechanism to describe a hierarchy of modules, a meaning to rearrange modules into some kind of *folders* and avoid name clashes. However, namespaces should not subsume modules, and as a result, they can only be used as a disambiguation construction in the language and only contain modules or *subnamespaces*.

Moreover, we don't want to clutter too much the core language with new features and constructs, just providing a way for the programmer to describe in the prelude of the module its compilation environment. Namespace declaration should not be mandatory, especially for a simple program without a complex architecture.

Another point we want to solve is having "functorized" namespaces to solve the *functor-pack* [2] feature. Since namespaces are the solution for `-pack`, this issue should also be resolved.

Finally, we think that namespaces are tightly related to the build system. As a result they should ease the use of external libraries for the user and the compiler, making external tools less vital for simple programs. In our view, a good namespace mechanism would have the ability to completely subsume the use of tricky options of the compiler (`-pack`), that makes dependencies computation harder without the use of a complete build system.

To summarize:

- A better replacement for `-pack`;
- Avoid naming conflicts;
- Simplify building without using external libraries;
- Solving the functor-pack to namespace formalization.

## 3 Current Proposal

In our current proposal, we aim at fulfilling the objectives previously described, by adding some constructs that are only usable in the prelude of modules (only *compilation unit*, not submodules), making them non-intrusive without making the source code more complex to read and understand. The namespace declarations *builds* the environment of following code, by explicitely declaring where the current module resides in the namespace hierarchy, and then what should be imported from it. It should be expressive enough to restrain the verbosity that arises in other languages, and would be helped by the recently introduced modules aliases.

We said namespaces should help the compilation scheme to put aside the use of external tools as much as possible. For this, what we propose is a canonical way to organize libraries in the file system, by mapping namespaces as folders directly, with the possibility of flattening those by a configuration file. Adding a standard path were installed library reside would help considerably the compiler to find and link the correct modules.

Finally, extensible namespaces allow any user to declare its modules into an already existing hierarchy. This allows any user to provide *extensions* of existing library (like the **stdlib**).

As an example, what we could achieve inside the language would be:

```
in namespace OCamlPro.Stdlib
  with {Hashtbl as OCPHashtbl, String}
  and {String as StdString, Hashtbl} of Inria.Stdlib
  and {String as JSString} of JaneStreet.Core
  and _ of Community.Batteries

open Hashtbl
open Enum
...
```

## 4 Conclusion

Namespaces are a tool in the language that have the ability to organize compilation units, leading to a more comprehensive library hierarchy. Our proposal aims to formalize a mechanism to add such features into OCaml, without cluttering the core language too much, and making the sources easier to read. It discharges the library provider to use long-prefixed names to avoid clashes, and helps the user and the compiler compute more efficiently the environment and dependencies.

Our proposal also aims to harmonize how libraries are installed in the file system and how the compiler reasons about them. In the actual configuration, using external tools is almost unavoidable. Namespaces would simplify the toolchain and let the compiler deal with external libraries and namespaces.

Our goal is to present the work that has been achieved toward adding namespaces into OCaml, from the proposal to hopefully an implementation.

## References


[1] *Working Draft, Standard for Programming Language C++*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf.

[2] F. Le Fessant. Packing and functors. http://www.ocamlpro.com/blog/2011/08/10/ocaml-pack-functors.html.

[3] M. Odersky. *The Scala Language Specification*, 2014. http://www.scala-lang.org/docu/files/ScalaReference.pdf.