Namespaces for OCaml: a proposal

Gabriel Scherer Didier Rémy ...

November 26, 2012

Contents

The compilation environment

In our view, namespace handling is about configuration of the compilation environment: which libraries are visible under which name to the OCaml source file currently being compiled. We define the term “namespace” (space of names) to refer to the compilation environment (the space of compilation units accessible through module names).

The current implementation of OCaml configures the compilation environment in a fairly naive way: it collects the `-I` flags passed in the compiler invocation command as a set of paths, looks for compilation units (a bundle of `.cmi`, `.cmo` and `.cmx` files with the same basename, with the sought-after extension depending on the compilation phase), and produces a compilation environment where a compilation unit ”`foo`.cm*” is represented, in an OCaml source file, by the capitalized module name `Foo`. The basename of the compilation unit must corresponds to the internal name stored inside the unit by the compiler, which means that the original source file for `foo.cmo` must have been named `foo.ml`, or else compilation fails.

(In practice such a compilation environment is not build eagerly before starting to analyze the compiled file, but the compilation units are looked-up on demand. This doesn’t change the semantics of external module references in the program.)

1 The Problem: naming conflicts among distributed libraries are hard to avoid

If two programmers happen to choose the same filename for a source file (and later compiled unit), it is not possible to use both units as two parts of the same program. The notion of "use" here is transitive with respect to dependencies, because ultimately all dependencies must be linked together without conflicts. If two different libraries each use one of the conflicting compilation units, you cannot use them together in a program.

This is a fundamental problem as the user is not able to change the name of the distributed libraries (the internal names are encoded in the units). Our ideal goal is to let the user combine two libraries that have been independently developed, without communication or coordination (explicit or implicit) between the providers.

1.1 Pack is not a satisfying solution

One could consider Pack as a potential solution to this conflict problem: library distributor can pack libraries into module hierarchies, that limit the risk of name clashes.

Unfortunately, packing produce huge .cmo files that the user then has to link with. Packing an alternative standard library such as Core or Batteries would make any user executable take tens of megabytes; users do not accept this choice, and library authors have to abandon packing.

A good namespace proposal should subsume `-pack`, in the sense that (most) current uses of `-pack` could be replaced by a corresponding use of namespaces.

1.2 Using long hopefully-unique names is impractical

The convention some libraries have adopted to avoid post-distribution conflicts is to use long, hopefully-unique module names (with no actual guarantees of unicity). We argue that the current combination of naive namespace semantics and module system semantics make this choice impractical.

To avoid being imposed the long hopefully-unique name throughout the program, a user could alias a distributed module under a shorter name.

```
module List = MyProg_Prelude_List
```

However the semantics of module aliasing implies a copy of the aliased module into the current compilation unit. In particular, it is not practical to factorize a coherent set of such renamings (possibly on the library side) and reuse it across programs, because that is equivalent to manual `-pack`-ing: it creates huge .cmo files.

Rather than bending the semantics of modules to satisfy this purpose, we expect a good namespace proposal to make the approach of long names practical, by giving users and library providers ways to refer conveniently to arbitrary compilation units.

2 Our proposal

We propose to accept namespace configuration as part of the language semantics, by specifying a clean set of operator to configure compilation environments. By letting users specify the compilation environment in which the source file is to be understood, we solve the problem of name clashes and long name inconveniences *on the user side.*

2.1 A hierarchical compilation environment

We suggest that such compilation environments could have a hierarchical structure. While a flat structure would be enough to solve the aforementioned problems, a hierarchical structure:

- is consistent with the way most other languages organize their library ecosystem
- naturally maps to user filesystems, which allows to conveniently infer the namespace from structured library storage
- allows a natural set of operators, for example to combine distinct namespaces: to be able to speak about libraries in the namespaces Core and Inria without risk of conflict, it's natural to define them as two subtrees (labeled Core and Inria) of a bigger environment

A hierarchical namespace is structured as a name-labeled tree whose leaves are compilation units. For example, we could invoke the compiler on a source file, passing the following compilation environment (where `+` is a shortcut for the OCaml default library directory):

```
{
  Joe: {
    List: "+site-lib/joe/joe_list",
    Array: "+site-lib/joe/joe_array"
  },
  Jenny: {
    List: "+list",
    ListDev: "/opt/ocaml/trunk/stdlib/list"
  }
}
```

We can refer to a compilation unit through the sequence of labels to access it through this environment: `Joe#List` for example (making an arbitrary choice of separator, here distinct from the `module-path` separator `.` to avoid ambiguities).

2.2 A language of namespace construction

We can understand the current OCaml semantics as a restricted set of conventions and "language" (command-line `-I` flags) to build such an environment – flat, with no observable nesting.

We propose to extend this to a proper description language for environments. Any reasonable set of operators on name-labeled trees could be fitting. Driven by the usual application needs, we suggest the following set of base operators.

E, F ::=		namespaces
$\mid$	$\{ \ (label : (\text{"}path\text{"} \mid E))^* \ \}$	literal structures
$\mid$	$E\#label$	projection
$\mid$	$E - label$	subenvironment removal
$\mid$	$E[label \backslash label]$	label renaming
$\mid$	$E[label := E]$	subenvironment replacement
$\mid$	E `only` S	signature restriction
$\mid$	E [`shallow`$\mid$`deep`] `merge` [`left`$\mid$`right`$\mid$`strict`] F	merge
$\mid$	`load` "*path*"	file loading
$\mid$	[`file`$\mid$`provenance`$\mid$`strict`] `scan` [`flat`$\mid$`rec`] "*path/*"	directory scan
$\mid$	x	variables
$\mid$	`let` x = E `in` F	`let` binding
S ::=		namespace signatures
$\mid$	$\{ \ (label \, [: S])^* \ \}$	literal signatures

The semantics of most operators are clear; we will only detail projection, `scan`, `only` and `merge`.

`E#Foo` accesses the subtree labelled `Foo` of the environment `E`. It should fail if `E` has no label `Foo`, not return the empty environment. This is the difference between our choice of modeling namespaces as tree structures (with a clear distinction between empty and non-existent subtrees), and the alternative choice of modeling them as mappings from paths to compilation units – as in an earlier proposal.

`file scan` builds an environment by scanning a directory, keeping its compilation units, and using the standard capitalization convention to choose their label. If the `rec` flag is given rather than `flat`, we scan subdirectories recursively to get a hierarchical environment corresponding to the filesystem structure. The other flag, `provenance scan`, has a slightly different semantics (it explores the filesystem but uses internal module information to choose the label path) that we will discuss later in subsection 2.5. If no left flag is given, `file` is the default. If no right flag is given, `rec` is the default.

`only` extracts from an environment a minimal environment satisfying a given signature. The term *signature* here refers only to the shape of environments as a tree of labels, with no type information. The restriction `E only` {`Foo`} is equivalent to a projection {`Foo:E#Foo`}, and `E only` {`Foo:S`} corresponds to, recursively, {`Foo:(E#Foo only S)`}.

Finally, `merge` returns the reunion of two environments. The optional flag on the right indicates the conflict resolution strategy: if the two merged environments both define the same path (sequence of labels), the strategy `strict` fails, `left` uses the content of label of the left argument, and `right` the right; the default value is `strict`. The optional flag on the left indicates a shallow or deep merge, that is whether two subenvironments under the root label are considered in conflict (`shallow`; fail or pick either subtree) or recursively merged (`deep`; conflicts happening only when a path designates a compilation unit in either environment); the default choice is `shallow`.

The previous example could be built in the following way:

```
let stdlib = scan "+" in
let joe = load "+site-lib/joe/joe.ns" in
let trunk = scan "/opt/ocaml/trunk/stdlib" in
{ Joe: joe only {List, Array},
  Jenny: { List: stdlib#List,
           ListDev: trunk#List } }
```

The current OCaml compilation environment could be described in the following way: we initially have just `scan "+"`, and each flag `-I "foo"` transforms the environment `E` into `E merge right (scan "foo")`.

We propose to keep this convention of command-line interface by adding an option `--namespace foo.ns` (or `-N foo.ns`) that would transform the environment `E` into `E merge right (load "foo.ns")`. More generally, we could map other elements of the namespace description language into environment-transforming command-line options, as deemed useful.

On a rich description language A lot of the discussion on namespaces and previous proposals evolved around an extremely minimal namespace description language, tailored to enforce one or another convention of name handling. On the contrary, we propose a semantic notion of namespace and a relatively rich description language to build such semantic objects.

We think this richness is not a source of complexity or bloat. We expect the proposal to respect a "pay for what you use" principle: users with simple needs could ignore the description language, keeping the default environment, but the more general descriptions would be useful in case of conflicts that currently require sophisticated workarounds or are unsolvable. Finally, the description language enforces no particular convention and empower tool writers and organizations to select them, rather than have to accept them. Of course, we can support the use of some widespread conventions (such as the current semantics) by providing additional description constructs for them (eg. `scan`).

Also note that a well-specified language based on a simple and general concept can evolve easily. We do not expect the namespace convention language to be set in stone, rather the semantics to be general enough to be gracefully extended to new needs in the future – more so than the current "file name is module name and linking name" design.

Finally, the clean separation between the compilation environment and the language improves robustness. Independently of the details of namespace description design, the value the descriptions produce (a tree-shapped mapping of names to compilation units, possibly lazily computed) is clear, and it is the only thing the OCaml source code will depend on.

2.3 The view from the language

We propose to add paths of the compilation unit environment, said otherwise "compilation unit names", as a new syntactic category `compunit-name` in the OCaml grammar. The `module-path` rule would be changed in the following way:

```
module-path ::=
  | compunit-name (. module-name)*
  | module-name (. module-name)*
```

Note that there is an ambiguity: if, as in our previous example, the lexical class for `compunit-name` includes the lexical class for `module-name`, the module path `List.Ord` may be seen either as referencing the compilation unit `List` or an internal module `List` (previously defined as `module List = ...` in the source file). This ambiguity, already present in the current version of the OCaml language, is the reason why `ocamldep` is so painfully approximative – more on that in the next section (3.2).

That is the only change to the OCaml language that is necessary to implement our proposal. If we were to use a flat rather than hierarchical namespace (as Alain Frisch suggested as a minimal proposal), mangle hierarchical compilation unit names into single module names, or use `.` as the label separator for compilation unit names, we could even avoid introducing any syntactic change at all, but we believe that there is sufficient value (additional structure and avoided confusion) in this proposal.

We discuss some possible additional changes in the "Considered extensions" section of this document (3).

2.4 Compilation unit information

The changes proposed so far essentially allow to solve the problems evoked in the introduction, under the assumption that programmers were to use long, hopefully-long module names. We are not satisfied with this as the only way to avoid name clashes for two different reasons:

- It does not coincide with the development techniques of most OCaml users right now, which prefer small, readable source file names. Helping users to *refer* to two different modules compiled from the same filename doesn't actually do any good if the OCaml linker rejects them.
- A naming strategy is not enough to ensure uniqueness in some fairly realistic scenarios, for example if an user needs to use two different versions of the same library in the same program. We hinted at this use case with our `Jenny#List` and `Jenny#ListDev` example: it is a very plausible need, and the "long filenames" approach to avoid clashes would impose to rename all the files of one of the version, which is impractical.

We therefore suggest the addition of two different information fields to the metadata of compiled units:

- an additional `suffix` field that would be part of the identity of the compilation unit along with the original file name (in compiled object code), and could be arbitrary data (for example a hash, or a randomly-generated unique identifier) with strong unicity guarantees.
- an additional `provenance` field that would provide provenance information to help humans distinguish two compilation units (coming from sources with the same filename).

Our suggestion for the `suffix` field would be to let the user optionally force its value, and otherwise pick a reasonable strategy that tries to ensure that two independent developers choosing the same file name do not result in a module identity clash situation at link-time. For example, reusing the digest present in the `.cmi` file would be a reasonable default choice. We could also consider generating a random nonce at `.cmi` production time (even less clashes, but makes compilation non-deterministic by default). There a design space here, and implementation constraints to take into account.

We're not aware of any "perfect strategy" that would fit all situations. Only using the interface digest could result in potential clashes for broad interfaces independently developed (`monad.mli`), and picking a random suffix at `.cmi` generation time could be problematic for specific test reproductibility needs and modes of code exchange (provide a `.mli` and expect a compilation unit in return to link with code already compiled locally against a corresponding `.cmi`). In those complicated situations (that we suspect less frequent than module name clashes are right now in the OCaml ecosystem), asking the user creating the compilation unit to explicitly set the suffix is the right solution.

2.5 Restrictions and naming conventions

The `provenance` field is a suggestion of Fabrice Le Fessant; he envisions a provenance convention similar to Java URL-like package names: `inria.stdlib.list` for the standard `List` module for example. This provenance would also have a path structure. Such a field would not be guaranteed to be unique among compilation units (eg. when compiling two versions of the same library) but could still provide useful guidance for user interaction, for example:

1. As a form of user-directed documentation: if module names are not unique among the compilation units of a project, we don't want the user to have to identify compiled units (in low-level error messages, etc.) by their unreadable `suffix` field; provenance information can help disambiguating.
2. To help populate the compilation environment in a "conventional" way respected across the OCaml users, to make it easy to read other users' code without details on their namespace.

This suggest the following addition to the language of namespace construction: a variant of the `scan [rec]` construction that would build a namespace where encountered files are indexed not by their filesystem path from the scanned root, but by their provenance information: the command would be `[file|provenance] scan [rec|flat]`, with `file` being the previous behavior selected by default, and `provenance` placing files according to their provenance. For example, `provenance scan rec "+"` could return a namespace such that `Inria#Stdlib#List` points to `"+/list"`, etc.

Fabrice further suggests the availability of an optional check that would enforce the semantics of both scanning methods to coincide, as a way to encourage users and packagers to build in a more structured, corporate-friendly environment: `strict scan rec "foo"` could verify that the filesystem organization under the path `"foo"` is such that the compiled unit at path `"foo/bar/baz"` has provenance information set to `"bar.baz"`.

3 Considered extensions

The previous section form what we consider as a simple and small, yet reasonable and usable, proposition of feature extension for better namespace support. Below are further considerations that could be omitted from the final feature set or its first implementation.

3.1 Integrating the namespace language as OCaml source headers

Other namespace proposals had variants of the operations available today on modules, retargeted for namespaces; considering what it would mean, in particular, to `open` a namespace path inside an OCaml source file. Our core proposal contains no distinct `open namespace` feature – or any other change to the OCaml language itself. Of course, you can write `open Std#List` if `Std#List` is a valid compilation unit name, and therefore is interpreted in OCaml source as the corresponding OCaml module, but there is no corresponding `open Std`.

We wish to maintain a clear phase separation between the compilation environment and the naming and typing environment of the program being processed: the compilation environment has been fully described when invoking the compiler, and it is not expected to evolve during the interpretation of a given source file, just as the granularity of compilation units is the whole source file.

Nevertheless, some programming languages that have this phase separation still allow users to specify namespace construction not anywhere in the source file, but at the very beginning of the program: the syntax allows for a "header" that contains `import` statements and potentially other unit-wide information. We think this would be a valuable addition for the grammar of future OCaml versions: it would allow users to embed (some part of) the namespace description directly in the program itself, rather than in the command-line options passed to the compiler.

This has the benefit of giving a canonical way for users to distribute the compilation environment along with their code – the environment being useful to interpret the code, just as today the semantics of a source file depend on the filesystem state and compilation option used. This is a more invasive language change and it would break the current (rarely used) property that source files can be put between `struct .. end` then concatenated together; we're afraid the elegant and simplifying "a source file is exactly a module" idea is suffering fatal growth pains.

3.2 Breaking compatibility for simpler dependencies

If we could come back in time, we would remove the current ambiguity between internal module names and compilation units. This change, suggested to us by François Pottier, has the very sensible benefit of making it trivial to see which compilation units a given source file depends on: just grep the source for compilation unit names.

In presence of the ambiguity, `ocamldep` has to consider each module name as a potential extern reference, which leads to a surapproximation of a file's dependencies in situations such as:

```
open A
(* a is an external dependency *)

open M
(* no way to know if this is external M or internal A.M
   if A's signature isn't available (not compiled yet) *)
```

Most OCaml build systems rely on `ocamldep`, or somehow run in this problem (you can't compute the dependencies of a file whose dependencies haven't been compiled yet). It can result in useless compilation but, more importantly, failed compilation due to imaginary cyclic dependencies. Solving it in a simple way would be a very attractive change for both tool authors and users.

It is not clear how such a transition could happen without hard compatibility break, and we are open to suggestions. We propose the following compromise:

- Allow the namespace label separator (written `#` in this document, but we are not hung up on the concrete syntax) to be used at the beginning of a namespace path. This would make both `List` and `#List` refer to the same compilation unit, with only the first one ambiguously also a module name.

- Add a `--strict-namespace` flag where the first syntax without the separator, `List`, is not accepted. When also passed this `--strict-namespace` flag, `ocamldep` would just look for occurrences of compilation units.

This side-effect of tremendously simplifying OCaml's dependency computation is one more reason to keep to the "phase separation" of namespaces mentioned in the previous subsection (3.1). In particular, it encourages us to reject proposals such as "local namespace open" – that don't make this impossible, but still more complex.

3.3 `functor-pack` integration