Namespaces for OCaml: ode to discussion

Gabriel Scherer   Didier Rémy   Fabrice Le Fessant   . . .

November 26, 2012

Yes, we have a Problem

Two library providers use the same filename.
This clash cannot be solved by the user.

Workaround 1: `-pack`-ing libraries
Problem: produces bloated `.cmo` that library users hate

Workaround 2: `jane_street_core_list.ml`
Problem: users get long names in their source code, and there is no satisfying way to alias modules (and distribute aliasing choices).

The module system of OCaml is complicated enough. We look for a solution that is not about modules *per se*.

Where does the problem come from?

We made the somewhat arbitrary choice to use the *file name* of a compilation unit as both:

1. the *module name* used to refer to compilation units from source
   (Can't talk about two clashing compunits in the same program)
2. the *internal name* encoded into compiled objects to be linked
   (Can't link together two clashing compunits)

We need to change both to support user-side clash resolution.

Making internal names more unique

A range of strategies:

- use long filenames, plus strategies for shortening them in source
- use a distinct provider-defined *provenance* field, for example `inria.gallium.stdlib.list`
- use the `.cmi` hash of the compilation unit
- use a random unique identifier (fixed in the `.cmi`) to get strong unicity
- let the user manually override the internal name

Any non-overridable name rules out some uncommon scenario, such as:

- linking together two compunits with the same interface (incompatible with pure `.cmi` hash)
- linking together two versions of the same library (incompatible with provider-defined filenames or provenance)

We must decide whether we want to leave the door open to them.

Naming compilation units from OCaml source files

Difference between *local modules* accessible in the typing env., and the *external units* looked up in the filesystem.

We formalize the latter lookup with a *compilation environment* that maps in-source *compilation unit names* to external compilation units.
It is currently defined by the include path.

We want richer ways to construct the compilation environment passed to the compiler. We call those compilation environments *namespaces*.

To solve clashes, let users refine their namespace.
For example, map
`{FooA -> "a/foo.cm*", FooB -> "b/foo.cm*"}`
to avoid a clash on `Foo` (if the internal names are unique enough).

The structure of a namespace

Namespaces are hierarchical: natural, convenient and expressive. Subsumes most uses of `-pack`.

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

This is only a semantic value, like the “mapping from module names to compunits in the module path” is imaginary today.
The *interface* to define these values (command-line flags, etc.) can change, but the notion is robust.

The namespace description interface: default namespace

General idea : with a good default choice, most users never hear about namespaces.

Reasonable default: scanning the content of the include path recursively, to get a hierarchical structure: `Camlp4#Printers#OCaml`.
(note: # is abstract syntax)

If the `provenance` field is present, we can alternatively use it: `Inria#Gallium#Stdlib#List`.
For additional guarantees, can even check that the two hierarchies coincide.

Namespace description: explicit constructs

Minimal additional construct (supported by Fabrice): provide an additional `open` construct to shorten some paths from the environment.
Conflicts would be resolved by letting the other choose which of `Stdlib#List` or `Core#List` is shortened to `List`.

Namespace description: explicit constructs

Minimal additional construct (supported by Fabrice): provide an additional `open` construct to shorten some paths from the environment.
Conflicts would be resolved by letting the other choose which of `Stdlib#List` or `Core#List` is shortened to `List`.

We can also ask for a richer description language. For example:

```
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 } }
```

A rigid common convention, or a more expressive description language?
What do users need?

Extremal description language

As an extremal design point, rich combinators plus special cases for common conventions.

$E, F ::=$	$\{\ (\mathit{label} : (\text{"}\mathit{path}\text{"} \mid E))^*\ \}$	literal structur
$\mid$	$E\#\mathit{label}$	projection
$\mid$	$E\ \texttt{only}\ S$	sig. restrictio
$\mid$	$E\ [\texttt{shallow} \mid \texttt{deep}]\ \texttt{merge}\ [\texttt{left} \mid \texttt{right} \mid \texttt{strict}]\ F$	merge
$\mid$	$\texttt{load}\ \text{"}\mathit{path}\text{"}$	file loading
$\mid$	$[\texttt{file} \mid \texttt{provenance} \mid \texttt{strict}]\ \texttt{scan}\ [\texttt{flat} \mid \texttt{rec}]\ \text{"}\mathit{path}/\text{"}$	directory scan
$\mid$	$\texttt{let}\ x\ =\ E_1\ \texttt{in}\ E_2$	let binding

Debate! Which subset do you need?

The view from the language

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

```
let test li =
  let foo = ... in
  let curr_result = Joe#List.map foo li in
  let dev_result = Jenny#ListDev.map foo li in
  Test.assert_equal curr_result dev_result
```

```
ocamlc -I foo -namespace bar.ns ...
=> (scan rec "foo") merge right (load "bar.ns")
```

Proposal: distinguish compilation unit names from modules

Currently `List` is ambiguous: internal module names or external compunit name.

The backward-compatible proposal preserves this.

Proposal: allow `#List` for the compunit name, and a `-strict-namespace` flag to disable the ambigous syntax `List`.

Benefit: `ocamldep` becomes `grep`!

Thanks to the people that discussed namespaces with me.

Didier Remy
Fabrice Le Fessant
Nicolas Pouillard
Alain Frisch
Martin Jambon
Jacques Garrigue
François Pottier
Edgar Friendly
Scott Kilpatrick