Type-level module aliases: independent and equal

Jacques Garrigue and Leo White

1 Synopsys

We promote the use of *type-level module aliases*, a trivial extension of the ML module system, which helps avoiding code dependencies, and provides an alternative to strengthening for type equalities.

2 Background

The richness of ML's module system theoretically allows one to flexibly structure libraries, without requiring extra features found in other languages, such as namespaces. Nested structures allow one to define hierarchies of modules, and functors provide flexible linking.

However, while this may be true from an internal point of view, the module system alone does not account for all aspects of libraries. A prominent problem is how to map module names to the file system. Another is how to support separate compilation, in a way that minimizes recompilations.

OCaml [5] chooses an extremely simple approach: module names which are not found in the internal environment are searched on the file system (under the same name), using a library path which is just an ordered list of directories. Such external modules are called *compilation units.* Separate compilation is supported by providing separate interface files for compilation units, containing only their signatures, which allows one to compile other units independently of the concrete implementation. Since the pair of a compilation unit's interface and implementation can be understood as a module with an opaque signature, this mechanism directly fits inside the semantics of the language. Coherence is enforced by keeping digests of all dependencies in compiled files. Smart recompilation is left to external tools.

However, this simplicity comes at a cost: the file system view of modules being completely flat, there is no way to have two compilation units with the same name linked into the same program. For programs relying on multiple libraries, this can be a severe stumbling block.

Since Objective Caml 3.05, this problem can be avoided by combining several compilation units as submodules of a packed unit. Note however that this packed unit completely replaces the original compilation units, and the packed unit's implementation must contain (or link) the implementations of all its submodules. This means that using any of these submodules requires you to link all of them. In that respect, one can say that it weakens separate compilation, even though what is no longer separate is not compilation but rather the result of compilation.

For SML/NJ, these problems are handled by the *Compilation Manager* [2, 1]. The compilation manager uses special files, written in a dedicated syntax, whose role is to define the mapping from module names to their implementations inside libraries, and where to find them on the file system. Using this information, it is possible to provide smart recompilation, and to avoid module name clashes, either by making names local to a library, or by using local mappings that modify the original module name binding. Thanks to its rich expressivity, the compilation manager avoids most restrictions of a file-system based approach. However, in order to do that, it introduces another language whose semantics is external to that of ML.

All this seems to lead to a contradiction: if the ML module system is so powerful, why should we need something else to express the namespace structure of concrete programs?

In this presentation, we propose a new approach to this problem, based on *type-level module aliases*, *i.e.* allowing interfaces express the aliasing structure of modules, and use this information to improve separate compilation. By extending OCaml with a single syntactic construct, we are able to handle namespaces inside the language itself, in a transparent and intuitive way. This feature is available in OCaml 4.02.

3 Aliases for equality

Originally, the idea of using module aliases in signatures was not related to separate compilation. It was introduced by Nakata and Garrigue in Traviata [7], as a mechanism to allow type reconstruction for recursive modules. It was later recognized that it was also useful in the absence of recursive modules, as it improves the behavior of OCaml style applicative functors [3]. See the following example, where `Set.Make` builds a module containing an ADT `t` together with some set operations.

```
module StringSet1 = Set.Make(String)
module StringSet2 = Set.Make(String)
module S = String
module StringSet3 = Set.Make(S)
```

Since OCaml's functors are applicative, the identity of nominal types produced by a functor application only depends on the functor's arguments. Here this means that `StringSet1.t` and `StringSet2.t` are equal. However, while it is clear from the source code that modules `S` and `String` are equal, we need type-level module aliases to derive a type equality between `StringSet1.t` and `StringSet3.t`.

In presence of module aliases, the signature of `module S = String` becomes

```
module S = String
```

or, using a syntax reminiscent of singleton kinds,

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

In this approach, one checks type equality by normalizing module paths, which is stronger than just expanding strengthened type definitions.

4 Aliases for independence

While useful, these extra type equalities alone would have been a weak justification for extending the language. However, module aliases have another important benefit: the information provided by types does not need to be duplicated in the implementation code, avoiding dependencies. For instance, consider the following compilation unit `Mylib`:

```
module A = MylibA
module B = MylibB
```

Since the aliases for `A` and `B` are revealed by its interface, the compiled implementation does not reference `MylibA` and `MylibB`. Which in turn means that a program using `Mylib.A` but not `Mylib.B` needs only to link `MylibA`, not `MylibB`[1]. This avoids the loss of independence observed with packed units. This also applies to `Mylib` itself: as long as the program does not use concrete values in `Mylib`, but only its aliases, there is no need to link a compiled implementation for it.

At the interface level, we can also remove dependencies: the interfaces for `MylibA` and `MylibB` are not accessed when compiling the interface for `Mylib`. As a result, the compiled interface of `Mylib` is completely independent of `MylibA` and `MylibB`. There is no need to recompile it when they change, and it is even possible to compile it before them. We will see that this provides a possible design for hierarchized libraries.

5 Application to namespaces

The approach we have just seen already gives us a way to avoid name clashes when building libraries: one should just prefix all unit names in the library with a common library name (here, `Mylib`). Ease of use by clients of the library is preserved by accessing it through the `Mylib` module. For instance, one can use `open Mylib` to make all submodules accessible through their non-prefixed names, without other side effects[2]. However, comfort would not be complete if we were not able to use non-prefixed names inside the library implementation. Fortunately, due to the absence of dependency between `Mylib` and its components at the interface level, one can actually use `open Mylib` inside `MylibA` and `MylibB`, without creating circular dependencies.

To summarize, one can replace packed units by applying the following recipe.

1. Create an interface 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.

To stay closer to the packed approach, OCaml 4.02 provides an `-open` command line option, which avoids adding the `open` explicitly to source files inside the library.

Note that mapping files are just plain compilation units. As such, they may do more than a flat mapping. For instance, one may choose to provide more structure to the exported version, by using submodules. One may also alias the same module several times, to provide different views. Since alias are just logical pointers, this comes at no cost.

An important testbed for module aliases has been the Core/Async family of libraries, developed by Jane Street [4]. Originally, they used packed units to avoid name clashes, and provide coherent naming schemes for their module hierarchies. However, this resulted in long compilation times and large executables. Using implicit module aliases already helps, by reducing the size of compiled interfaces (read during compilation) by an average of 3. Moreover, using techniques such at the above, they could keep the same naming scheme, but remove dependencies, and reduce executable sizes on average by a factor of 2 (up to a factor of 10 in some concrete cases) [6, 8].

6 Limitations and future work

In order to avoid a number of practial and theoretical issues, the current implementation of modules aliases in OCaml 4.02 introduces a number of restrictions on which module aliases can be reflected in types. Namely, the following kinds of module expressions (and their submodules) cannot generate type-level aliases.

- Plain structures and functors.
- Functor applications.
- Opaque coercions.
- Functor arguments.
- Recursive modules.

While the first one is essential, as it would amount to including the full language inside signatures, we believe that all other restrictions could be eventually removed, making this feature more elegant.

[1]Actually this behavior is not backward compatible if `MylibB` contains side-effects. For this reason it is enabled by the compiler option `-no-alias-deps`.

[2]In OCaml, `open` only makes components of a module directly accessible locally, it does not re-export them.

References

[1] Matthias Blume. *CM: The SML/NJ Compilation and Library Manager*. Lucent Technologies, Bell Labs, May 2002. `http://www.smlnj.org/doc/CM/new.pdf`.

[2] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, 1999.

[3] Jacques Garrigue and Keiko Nakata. Path resolution for recursive nested modules. *Higher-Order and Symbolic Computation*, 24:207–237, 2012.

[4] Jane Street. Open source software. `http://janestreet.github.io/`.

[5] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.01, Documentation and user's manual*. Projet Gallium, INRIA, September 2013.

[6] Yaron Minsky. Better namespaces through module aliases. `https://blogs.janestreet.com/better-namespaces-through-module-aliases/`, May 2014.

[7] Keiko Nakata and Jacques Garrigue. Recursive modules for programming. In *Proc. International Conference on Functional Programming*, Portland, Oregon, 2006.

[8] Mark Shinwell and Nick Chapman. Personal communication, May 2014.