Enforcing OCaml link-time compatibility using Debian dependencies

Stefano Zacchiroli
<zack@debian.org>

27/01/2009

Abstract

To ensure type-safety, OCaml has strict link-time rules: involved objects are not allowed to change between (separate) compilation time and linking time. This does not cope well with the implicit assumption of future compatibility relied upon in general library packaging in F/OSS distributions such as Debian.

We discuss how to enforce OCaml link-time type safety using Debian inter-package relationships (i.e., *dependencies*). In doing so, we take into account Debian maintainability problems such as support for `binNMU`s (binary non-maintainer uploads) and preservation of human-readable dependency strings.

Latest PDF:	http://upsilon.cc/~zack/stuff/ocaml-debian-deps.pdf
Sources (`gitweb`):	http://git.upsilon.cc/cgi-bin/gitweb.cgi?p=papers/ocaml-debian-deps.git
This version ID:	f784af83f0b840e6f437782ead7f088412c7107a

1 Introduction

1.1 OCaml's inter-module assumptions

To ensure type-safety, OCaml does not allow object files subject to linking to change between their compilation and their linking. In particular, when a `.cmo` file is built from a `.ml` file representing an OCaml module (usually with something like `ocaml -c file.ml`) a set of *assumptions* about all referenced OCaml modules are stored in the `.cmo` itself.

Those assumptions take the form of checksums (currently MD5 hash values) about various parts of the referenced modules. Two kind of assumptions are currently stored:

Interface assumptions checksums of the (implicit or explicit) `.mli`[1] files of all modules referenced by the `.ml` file being compiled

[1] not exactly of the `.mli` file itself, but for what concerns this analysis it can be assumed so: if the `.mli` file changes, the checksum will change

Implementation assumptions checksums of all the `.ml` files which are going to be inlined as part of the file being compiled

While interface assumptions are recorded for both bytecode and native code compilation, implementation assumptions are recorded only for native code compilation (since it is the only kind of compilation which performs inlining).

Interface assumptions can be inspected using the `ocamlobjinfo` tool which is shipped as part of the legacy OCaml distribution; it is enough to invoke it on some kind of bytecode OCaml object.

Example 1.1 (Interface assumptions)**.** *Consider the following source files:*

Listing 1: `libFoo.ml`

```
open Printf

let hello () = printf "Hello, world!\n"
let gotcha () = ()
```

Listing 2: `libBar.ml`

```
let hello () =
  LibFoo.hello ();
  LibFoo.hello ()
```

Listing 3: `myApp.ml`

```
let _ = LibBar.hello ()
```

After building with the bytecode compiler, the object of `libBar` *will store an interface assumption over* `libFoo`*, which can be inspected with* `ocamlobjinfo`*:*

```
$ ocamlc -c libFoo.ml
$ ocamlc -c libBar.ml
$ ocamlobjinfo libBar.cmo
File libBar.cmo
  Unit name: LibBar
  Interfaces imported:
        5dbbf45a03b54e6dbfcf39178d0d6341        Printf
        f6cef633ea14963b84b79c4095c63dc3        Buffer
        8ba3d1faa24d659525c9025f41fd0c57        Pervasives
        404a90bf6e42d9cf101edde25eba92db        LibFoo
        5cfae708052c692ea39d23ed930fd64d        Obj
        ffce1e91f2a0e2c49dbaa52bbe7cb364        LibBar
  Uses unsafe features: no
$
```

At link time, the stored assumption over `libFoo` must match what the linker discover by itself about `libFoo`. If this is the case, linking would work properly:

```
$ ocamlc libFoo.cmo libBar.cmo myApp.ml
$ ./a.out
Hello, World!
Hello, World!
```

If that is not the case, linking would fail:

```
$ echo "let gotcha () = ()" >> libFoo.ml
$ ocamlc -c libFoo.ml
$ ocamlc libFoo.cmo libBar.cmo myApp.ml
Files libBar.cmo and libFoo.cmo
make inconsistent assumptions over interface LibFoo
$
```

Note that the mismatch can be detected a priori, with respect to linking, using `ocamlobjinfo`:

```
$ ocamlobjinfo libFoo.cmo libBar.cmo |grep LibFoo|tail -n 2
  Unit name: LibFoo
       50c3f57970c65dd807a3c48940300a15       LibFoo
       404a90bf6e42d9cf101edde25eba92db       LibFoo
$
```

TODO: mention that implementation assumptions are similar

TODO: mention that there is no equivalent to `ocamlobjinfo` to inspect implementation assumption in the legacy distribution, but that there is an external tool (where is it?)

1.2 Naive inter-package dependencies

Now, let's say that `libFoo`, `libBar`, and `myApp` of Example 1.1 are packaged as separate Debian packages, respectively `libfoo-ocaml-dev`, `libbar-ocaml-dev`, and `my-app`. Consider the natural, yet simplified, dependency scheme:[2]

```
Package: my-app
Version: 1
Build-Depends: libbar-ocaml-dev (>= 1)

Package: libfoo-ocaml-dev
Version: 1

Package: libbar-ocaml-dev
Version: 1
Depends: libfoo-ocaml-dev (>= 1)
```

Such a scheme it is not enough to ensure the following desirable property is fulfilled by all the involved packages:

[2] to simplify, we report build dependencies together with dependencies in the same stanza, while in `debian/control` they must belong to different stanzas

Property 1 (OCaml packages dependency soundness)**.** *A set of packages $p_1, \ldots, p_n$ has* sound OCaml dependencies *iff the satisfaction of their inter-package dependencies implies that no link-time failures due to inconsistent assumptions can arise among OCaml objects shipped by $p_1, \ldots, p_n$.*

Dependency satisfaction is defined as usual accordingly to the Debian policy [2].

Trivially, the property cannot be fulfilled because future versions of, say, `libfoo-ocaml-dev` can defeat the assumptions under which `libbar-ocaml-dev` has been built. This, in turn, will induce a link-time failure when a rebuild of `my-app` will be attempted, *unless* `libbar-ocaml-dev` will be rebuilt against the newer `libfoo-ocaml-dev`.

Using more strict version predicates, such as:

```
Package: libbar-ocaml-dev
Version: 1
Depends: libfoo-ocaml-dev (>= 1), libfoo-ocaml-dev (<< 2)
```

will not work either, because there is always a version number between any two version numbers, and that version can be the one defeating the OCaml assumptions of reverse-dependent packages.

To conclude, two observations are in order:

1. The problem arises because OCaml assumptions are not reflected in the dependency language.

2. An analogy with the packaging of C libraries stands: each package shipping (linkable) OCaml objects define its own ABI, that ABI can be defined as follows:

   Definition 1.1 (OCaml package ABI)**.** *The* OCaml ABI *of a package shipping linkable OCaml objects is a function $abi : \mathcal{M} \to \mathcal{C}$, where $\mathcal{M}$ is the set of valid OCaml module identifiers and $\mathcal{C}$ is the set of (MD5) module checksums.*

   TODO: this definition does not account for the distinction between interface and implementation assumptions, refine it!

   Each change to the ABI function, even point-wise, triggers a link-time failure, whereas for C libraries point-wise (i.e., symbol) changes only affects users of that symbol. In particular, this enables C libraries to ship backward-compatible changes as ABI additions.

   Practically, this means that OCaml library ABIs change very frequently, possibly at each library release. This makes unfeasible using the same solution adopted by C library packagers, namely reflect ABI versions in package *names*.

1.3 Debian shared library information files

A related infrastructure piece, which interacts with what we are aiming for, is the `shlibs` registry, used by some cooperative `debhelper`s [1] to automatically infer package dependencies starting from shared library dependencies.

The typical inter-package dependencies which can be inferred using `shlibs` are those from packages shipping C objects linked against shared libraries (typically dynamically linked executables or libraries) to the packages actually shipping the shared libraries.

A brief description of how the `shlibs` mechanism works follows; more details can be found in the following manual pages: `deb-shlibs(5)`, `dh_shlibdeps(1)`, `dh_makeshlibs(1)`.

Each package shipping a C shared library (i.e., one or more `.so` files) should ship a `shlibs` file which gets installed by `dpkg` as `/var/lib/dpkg/info/PKGNAME.shlibs`.

`shlibs` files are line-oriented, one record per line; considered all together they form a `shlibs` registry. Each record associates together 3 fields: a *library name*, a *package name*, and a *version predicate* on that package.

The intended usage of such a tuple is, intuitively, that every package containing C objects referencing a library appearing in the first field must have a dependency on the package reported in the second field, with version requirement according to the third field.

Example 1.2 (`shlibs` file)**.** *Here is the content of `shlibs` shipped by the `libglib2.0-0`:*[3][4]

Listing 4: `/var/lib/dpkg/info/libglib2.0-0.shlibs`

```
libglib-2.0 0 libglib2.0-0 (>= 2.16.0)
libgthread-2.0 0 libglib2.0-0 (>= 2.16.0)
libgmodule-2.0 0 libglib2.0-0 (>= 2.16.0)
libgio-2.0 0 libglib2.0-0 (>= 2.16.0)
libgobject-2.0 0 libglib2.0-0 (>= 2.16.0)
```

Hence, if for instance a package ships a library or executable dynamically linked against `libglib-2.0`, that package must have among its dependencies something like: `Depends: libglib2.0-0 (>= 2.16.0)`, according to the first line of `libglib2.0-0 shlibs` file.

Practically, `shlibs` files are created automatically by the debhelper `dh_makeshlibs` which inspects installed `.so` files and can be driven with specific information about version requirements. Symmetrically, inferred dependencies are added by the debhelper `dh_shlibdeps` which looks up all shared library dependencies of shipped objects (as can be obtained using `ldd`) against the `shlibs` registry and collect the relevant dependency snippets. The final step of filling the `Depends` field is delegated to the `substvars` mechanism.[5]

[3] Note: some irrelevant entries of that file have been omitted as irrelevant, in particular all entries related to `udeb` management have been omitted

[4] note that `shlibs` files are managed directly by `dpkg` and are not shipped as the *content* of `.deb` packages. Hence, trying to lookup `shlibs` files using commands such as `dpkg -L` or `dpkg -S apt-file search` will not work as expected

[5] See the `deb-substvars(5)` manual page.

1.4 binNMUs

Nothing to see here yet, in the meantime have a look at `http://wiki.debian.org/binNMU`

TODO: fill this section

2 Dependency management desiderata

In this section we state the good properties we are looking for, in dependency management mechanism for OCaml libraries and packages.

Dependency soundness First of all we want Property 1 (*dependency soundness*) to be satisfied: inter-package dependencies should be enough to ensure link-time compatibility between OCaml objects. For Debian users this would mean that temporary link-time incompatibilities[6] can be detected by package managers, avoiding the annoying routine of "upgrade first" and "discover then" that the OCaml toolchain has been broken.

Dependency inference Then, we want dependencies to OCaml libraries (or, more generally, to whatever package shipping OCaml objects) to be automatically inferred by (possibly OCaml-specific) packaging tools. Ideally, using helper external tools we want our `debian/control`s to be, with respect to dependencies, something like:

```
Package: my-app
Version: 1
Build-Depends: libbar-ocaml-dev (>= 1)
Depends: ..., ${ocaml:Depends}, ...

Package: libbar-ocaml-dev
Version: 1
Build-Depends: libfoo-ocaml-dev (>= 1)
Depends: ..., ${ocaml:Depends}, ...
```

where `${ocaml:Depends}` is a usual substitution variable automatically filled with the (possibly versioned) names of Debian packages shipping the needed OCaml objects for the package at hand.

`binNMU`-safety We want OCaml-related packages to be `binNMU`-safe. That not only means that inferred dependencies should not be tied to source package version, but also that we cannot rely on source uploads. As we will have no guarantee of source uploads, architectures relying on changes to be incorporated in source packages are not suitable for the task. All OCaml-specific dependency information should be recomputed during build, depending only on the (binary) packages installed as build dependencies.

[6] Such incompatibilities can appear from time to time, most notably in the `unstable` or `experimental` archives, and during library or compiler transitions.

Note that binNMU-safety is not only for the sake of doing things "right", but also a real need to reduce maintenance burden of OCaml packages. We are floating towards about 100 OCaml related source packages. Transitioning them by the means of source upload would be unnecessary painful and time consuming, binNMUs help in keeping the task manageable.

2.1 State of the art

How do the current practices in maintaining OCaml packages score with respect to the above desiderata?

2.1.1 Almost "by hand" OCaml dependencies

Currently, the OCaml packaging guidelines [3] prescribe to manually fill inter-package dependencies. The suggested dependency scheme rely on >= predicates between libraries, but are by no means required to be bound to the latest available version of the library in the archive or something such.

Here is how the current scheme scores with respect to our desiderata:

✗ **dep. soundness**: dependencies do not guarantee link-time compatibility, they are too coarse and do not encode OCaml assumptions

✗ **dep. inference**: dependencies are filled by hand

✓ **binNMU-safety**: packages are binNMU-safe, in the sense that dependencies are generally stable and do not change upon (binary) rebuild.

The only exceptions to that stability are the dependencies on the compiler itself and related tools; those exceptions are accounted for by filling them via compiler-specific substitution variables.

2.1.2 dh_ocaml & ocaml-md5sums

Back in 2004, dh_ocaml has been developed to improve over the aforementioned "by hand" dependency management. In spirit, dh_ocaml is very similar to the shlibs mechanism (see Section 1.3): it is based on an underlying registry (the "OCaml md5sums registry"[7]) handled by ocaml-md5sums which plays the role of the .shlibs files.

The registry is composed by .md5sums files shipped together with packages which contain OCaml objects. In this case, there is no specific management required from the package manager, the registry files are ordinary files shipped as part of the package content. Those files are stored under /var/lib/ocaml/md5sums/. To speed up lookup, a comprehensive registry containing all the entries of all shipped files is kept as /var/lib/ocaml/md5sums/MD5SUMS.

[7] ... which should be better renamed to something along the lines of "OCaml module assumption registry"

It can be considered as a `cat`-together of all registry files, and can be updated by invoking `ocaml-md5sums update`.[8]

Each `.md5sums` file in general contributes several entries to the md5sums registry. Each entry is a tuple $\langle md5sum, module, devel\_dep, runtime\_dep, version \rangle$. The various entries are, respectively, the checksum corresponding to a (interface) OCaml assumption, the module name matching the assumption, and dependency information describing the Debian package which is providing a module matching that assumption. We omit the discussion of why such dependency information are split in three parts (devel-time dependency, runtime-dependency, and version).

TODO: expand this part instead of omitting it

Having such information for all installed packages which ship OCaml objects, it becomes possible to lookup the registry to automatically infer dependencies over those packages. The work-flow to infer dependencies for a package p which is being built will be as simple as:

1. build p;
2. identify the OCaml objects that p ships (in case it is a library) or the OCaml executables that p ships (in case it is a binary program);
3. query them for their assumptions over all OCaml objects which are *not* shipped as part of p;
4. lookup each such assumption into the md5sums registry and emit the appropriate dependency snippet.

This work-flow is actually implemented by `dh_ocaml`, which is a `debhelper`-like tool that does all of the above and in addition also provides some convenience substitution variables.

Now, for the score chart of `dh_ocaml`:

✗ **dep. soundness**: `dh_ocaml` is orthogonal to dependency soundness. In fact, in the current implementation, `dh_ocaml` just *implements* the same dependency scheme which is currently used "by hand". Given that that scheme does not guarantee dependency soundness, `dh_ocaml` cannot improve upon that.

✓ **dep. inference**: this is the true improvement offered by `dh_ocaml`, and actually its main design goal.

✓ **`binNMU`-safety**: as per dependency soundness, `dh_ocaml` is orthogonal to this and preserve the (this time good) property of the underlying dependency scheme.

[8] Following a common packaging pattern, packages shipping `.md5sums` files are require to update the registry from theirs `postinst` and `postrm` maintainer scripts.

3 Towards better dependency management

What we are missing for a foreseeable satisfiable solution to OCaml dependency management is, in essence, dependency soundness (Property 1).

At DebConf7 a discussion have been held on the topic between members of the OCaml packaging task force and the release team. We report in this section some of the considered solutions (together with why they have been ditched) and the final, proposed solution.

OCaml assumptions as virtual packages This first solution attempt mimics the dependency scheme being used by RedHat to package OCaml-related software. The idea is to add a `Provides` entry (i.e., a virtual package) for each assumption provided by objects shipped by a given package. In a sense, this solution recasts the content of md5sums registry entries into package descriptions as virtual packages.

TODO: add an example of OCaml-related package, together with all its provided virtual packages

This solution is not acceptable in Debian for various reasons:

- It will bloat APT package lists such as `Packages` file.
- It provides virtual package names which are not only entirely meaningless for humans, but also hard to grasp for people which sooner or later will *have* to deal with them, where "read" can mean something as simple as "reading them" (e.g.: users facing dependency problems reported by package managers, release managers following a transition to `testing`).

TODO: following solution names suck big times, look for something better ...

ABI evolution tracking To counter both problems of the previous solution, it would be enough to have a mapping from OCaml package ABIs (in the sense of Definition 1.1) to integers.

Let's imagine we can associate the number 1 to the ABI corresponding to the current version of package `libpcre-ocaml-dev` in the `unstable` archive. Then we can form a meaningful virtual package name such as `libpcre-ocaml-dev-1`, make the real package provide it and reverse-dependent package rely on it. As long as the ABI number, and hence the virtual package name, can be computed during build of both the package itself and of reverse-dependent packages, everything works as we need.

Unfortunately this solution makes unachievable the goal of `binNMU`-safety. Indeed to ensure that ABI numbers will not clash in the future we will have to either maintain by hand a mapping between the ABI function (which roughly correspond to md5sums registry entries) and version numbers,[9]or ensure automatically a monotonic increase of ABI number. Both options defeat `binNMU`-safety; in particular the latter will need to preserve somewhere a history of past ABI numbers which cannot be saved anywhere upon `binNMU`s.

[9]Note: such a solution would resemble the recently added support for symbol management to the `shlibs` mechanism (see the manual page `deb-symbols(5)`). In practice, with that technique, ABI changes are detecting while rebuilding packages and trigger build failures. While that is acceptable for C libraries, where breakages seldom happens, it is not for OCaml libraries, as it will potentially inhibit transitions via `binNMU`

Approximated ABI Building upon previous reasoning, we can achieve a mapping between ABI functions and ABI identifiers by taking a checksum of the md5sums registry entry of a package shipping OCaml objects.

If the checksum is short enough, we can produce manageable/readable virtual package names (e.g., `libpcre-ocaml-dev-1234`) still retaining a low probability of clashes. Rough consensus in the DebConf7 discussion has emerged on choosing as ABI identifiers string of length 4, composed by only digits, as in the previous example.

This dependency management scheme will account for:

- ✓ **dep. soundness**, of course up to clashes. The probability of clashes will be of the order of 1/10'000, which is way better than the current situation anyhow
- ✓ **dep. inference** by simply piggybacking the approximated ABI mechanism on top of the current implementation of `dh_ocaml` / `ocaml-md5sums` (and of course by starting using them on *all* packages shipping OCaml objects!)
- ✓ `binNMU`**-safety** no state is required for computing approximated ABI values, they depend only on registry entries of build dependencies and on shipped OCaml objects

Practically, the implementation of this mechanism can be achieved on top of the readily available `dh_ocaml` and `ocaml-md5sums`.

4 Conclusion

TODO: add paper summary here

4.1 Outstanding issues

Incompleteness of assumption detection currently, the only tool available to inspect OCaml assumptions is `ocamlobjinfo`. It is limited in which it only works for bytecode objects and (as a consequence) only reports about interface assumptions, neglecting implementation assumptions.

References

[1] Joey Hess. `debhelper` debian package: helper programs for `debian/rules`. `http://packages.debian.org/sid/debhelper/`, 2009. Version 7.0.15.

[2] Ian Jackson and Christian Schwarz. Debian policy manual. `http://www.debian.org/doc/debian-policy/`, 2008.

[3] Sylvain Le Gall, Sven Luther, Samuel Mimram, Ralf Treinen, and Stefano Zacchiroli. Debian OCaml packaging policy. `http://pkg-ocaml-maint.alioth.debian.org/ocaml_packaging_policy.html/`, 2009.

A Roadmap

TODO: write a roadmap to achieve full implementation of the proposed solution