Enforcing Type-Safe Linking using Inter-Package Relationships

M. Dogguy[1] & S. Glondu[1] & S. Le Gall[2] & S. Zacchiroli[1]*

1: Université Paris Diderot,
Laboratoire PPS, UMR 7126
`{dogguy,glondu,zack}@pps.jussieu.fr`
2: OCamlCore SARL
`sylvain.le-gall@ocamlcore.com`

Abstract

Strongly-typed languages rely on *link-time checks* to ensure that type safety is not violated at the borders of compilation units. Such checks entail very fine-grained *dependencies* among compilation units, which are at odds with the implicit assumption of *backward compatibility* that is relied upon by common library packaging techniques adopted by FOSS (Free and Open Source Software) package-based distributions. As a consequence, package managers are often unable to prevent users to install a set of libraries which cannot be linked together.

We discuss how to guarantee link-time compatibility using inter-package relationships; in doing so, we take into account real-life maintainability problems such as support for automatic package rebuild and manageability of ABI (Application Binary Interface) strings by humans. We present the `dh_ocaml` implementation of the proposed solution, which is currently in use in the Debian distribution to safely deploy more than 300 OCaml-related packages.

1. Introduction

Type safety is a tricky business, even more so when separate compilation is desired. In the world of system-level languages and linkers [12]—such as the C language and the widespread GNU linker—very few checks are performed at the final linking stage; a bit of formalization will help in understanding them. Given a set of *compilation units* $\{u_1, \ldots, u_n\}$ to be linked together the linker checks, in essence, a form of referential integrity, i.e. that all symbols needed by involved compilation units are actually available within the set. We call *requirements* of a compilation unit $\mathcal{R}(u)$ the set of required symbols and *application binary interface* (ABI) of a compilation unit $\mathcal{A}(u)$ the set of provided symbols. The linker notion of "linkability" can hence be grasped as follows:[1]

Definition 1 (Linkability). *link*$\{u_1, \ldots, u_n\} = \checkmark$ *iff:*

1. $\bigcup_i \mathcal{R}(u_i) \subseteq \bigcup_i \mathcal{A}(u_i)$
2. $\forall i\, j, \quad i \neq j \rightarrow \mathcal{A}(u_i) \cap \mathcal{A}(u_j) = \emptyset$

where the second conditions avoids multiple definitions.[2] No matter the type system expressivity, it is evident that such a linking discipline cannot help in enforcing type safety *across compilation units*, if not relying on name mangling hacks [13], or delegating it to external whole program verification [7].

*Partially supported by the European Community FP7, MANCOOSI project, grant agreement n. 214898

[1] We do not strive for completeness here, we grasp only the linker checks that will help in comparing with the strongly typed language world. For the same reason, we do not distinguish between static and dynamic linking.

[2] Actually, in some corner cases, the linker can allow them, but that is uninteresting for our purposes.

Listing 1: foo.ml

```
let hello () =
  Printf.printf "Hi!\n"
```

Listing 2: bar.ml

```
let hello () =
  Foo.hello ()
```

Listing 3: main.ml

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

Figure 1: sample OCaml compilation units

Moving to the world of functional, staticly typed programming languages, such as OCaml or Haskell, link-time checks get more thorough mainly because types come into play. Not only cross-module type compatibility is challenging to verify *per se* [1, 10], but also technical guarantees that ABIs do not change between compile time of individual units and link time are requested to be type-aware. The solution adopted by OCaml is, for each compilation unit, to expose two sets of *module names*, associating each name to a cryptographic hash or *checksum* that grasps the type information of that module.

Example 1. *Let's consider the sources of Figure 1. After (bytecode) compilation of* `bar.ml`*—which in turn needs a compiled version of* `foo.ml`*—the resulting compilation unit contains the following "assumptions":*

```
$ ocamlc -c foo.ml bar.ml
$ ocamlobjinfo bar.cmo
Unit name: Bar
Interfaces imported:
        807ecd3a1538992580464c03462c9964        Printf
        da00042bb934260afe41d004bc91fe2e        Foo
        9e340434237964195546 1e6944482508       Bar
```

where we can see that `bar.cmo` *exports an ABI consisting of the interface* `Bar` *with a specific MD5 checksum, and that it* requires *some other checksum-tagged interfaces. Among them we can spot* `Foo`*, provided by* `foo.ml`*, and* `Printf`*, provided by the OCaml standard library which is linked in by default. If the ABI of* `foo.cmo` *changes between the compile time of* `bar.cmo` *and the final link time, the user will incur in the sadly well-known "inconsistent assumptions" error:*

```
$ ocamlc -c foo.ml bar.ml
$ echo "let gotcha () = ()" >> foo.ml
$ ocamlc -c foo.ml
$ ocamlc foo.cmo bar.cmo main.ml
Files bar.cmo and foo.cmo make inconsistent assumptions over interface Foo
```

In our simple formalization, the additional checks performed by OCaml already fit, by simply considering both $\mathcal{R}(u)$ and $\mathcal{A}(u)$ to be sets of *pairs* $\langle m, c\rangle$, where m is the name of an OCaml module and c is its associated checksum. The only additional property checked by the OCaml linker is that, given a set of compilation units, the mapping between module names and checksums is a function, i.e. that a module is not associated with different checksums. Simplifying the characteristics of the standard library *stdlib*, the failure of Example 1 can now be explained as follows, where the final equation was supposed to equate the empty set in order to satisfy Definition 1.

$$\begin{array}{rcl|rcl}
\mathcal{R}(stdlib) & = & \emptyset & \mathcal{A}(stdlib) & = & \{\langle \texttt{Printf}, \texttt{807ec}\ldots\rangle\} \\
\mathcal{R}(\texttt{foo.cmo}) & = & \{\langle \texttt{Printf}, \texttt{807ec}\ldots\rangle\} & \mathcal{A}(\texttt{foo.cmo}) & = & \{\langle \texttt{Foo}, \texttt{a4293}\ldots\rangle\} \\
\mathcal{R}(\texttt{bar.cmo}) & = & \{\langle \texttt{Printf}, \texttt{807ec}\ldots\rangle, & \mathcal{A}(\texttt{bar.cmo}) & = & \{\langle \texttt{Bar}, \texttt{9e340}\ldots\rangle\} \\
 & & \ \ \langle \texttt{Foo}, \texttt{da000}\ldots\rangle \quad\ \} & & &
\end{array}$$

$$\bigcup_{u\in\{stdlib,\texttt{foo.cmo},\texttt{bar.cmo}\}} \mathcal{R}(u) \setminus \bigcup_{u\in\{stdlib,\texttt{foo.cmo},\texttt{bar.cmo}\}} \mathcal{A}(u) = \{\langle \texttt{Foo}, \texttt{da000}\ldots\rangle\}$$

The ability of the linker to detect this kind of unsound assumptions comes at a cost: ABI changes more frequently than in the C case. Indeed, with system-level linking, the ABI of a given unit can inhibit linkability only by removing symbols from it. Backward ABI compatibility—which of course does not imply type safety—can be retained by simply adding new symbols, which is unsurprisingly common practice in the work flow of C libraries. With type-aware linking, ABIs break at each change in a module,[3] no matter if it is an addition or a removal, because each such modification will change the checksum of the module. While C libraries offer (type-unsafe) backward binary compatibility by default, OCaml libraries have the converse default: they break binary compatibility at each change.

Unfortunately for most users of languages such as OCaml and Haskell, packaging systems and techniques used in mainstream FOSS (Free and Open Source Software) distributions have been designed with *implicit backward compatibility* among libraries in mind. Dependencies on library packages are usually expressed in forms like the following:

```
Package: my-app
Depends: libfoo1 (>= 1.2.3)
```

where it is implicitly assumed that future versions of the `foo` library will either be backward binary compatible, or change the package name *tout court*, for instance switching to `libfoo2`.

The advantage of the backward compatibility assumption is that inconsistent software installations, where installed libraries lack the needed objects to be linkable, are detected at the dependency level and can hence be spotted by package managers. Given that there is no reliable mapping between library versions and ABIs (new versions *can* change ABIs, but are not forced to), we observe that with the OCaml linking discipline the advantage of spotting linkability errors at the dependency level is gone. As most OCaml users on distributions such as Debian[4] have experienced, it indeed frequently happens that upgrades involving OCaml libraries temporarily leave the build toolchain in an inconsistent state, [5] requiring users to: recompile dependent libraries by hand, rollback the upgrade (if possible), or simply wait for fixed packages (of all involved libraries) from the distribution.

In this paper we show how to make the packaging system aware of complex and frequently changing ABIs used by languages such as OCaml and Haskell, hence shielding users from the installation of non-linkable libraries. Attempting to do so will equate to trying to install a package whose dependencies cannot be satisfied. We achieve that result by coalescing ABIs in a single, human-readable, checksum—called *ABI approximation*—which is then reified as a "virtual" package known to the packaging system, de facto closing the gap between coarse-grained inter-package relationships and fine-grained inter-unit linking assumptions.

Paper structure. The next section introduces some concepts of the FOSS distribution context that will be needed throughout the paper. Section 3 poses the requirements for the solution we look for, solution which is then described and analyzed in Section 4; its actual implementation—in the context of the Debian distribution and for the OCaml language—is discussed in Section 5. From now on, we will focus on the OCaml language and its packaging in Debian, but reasoning and choices are general enough to be ported to similar languages (with strong typing and inspectable linking assumptions) and distributions (based on binary packages).

2. Packaging basics

Packages. In most FOSS distributions—and, more generally, in component-based systems [14]—software components are managed as *binary packages* [2] which define the granularity at which

[3] Here, we do not distinguish among assumptions on interfaces and assumptions on implementations, these details are postponed to Section 5.

[4] `http://www.debian.org`

[5] For example, `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=238727` (retrieved October 2009); similar reports are very frequent during transitions to one OCaml version to the next one, since all libraries need to be rebuilt.

users can add or remove software. A binary package is essentially a bundle of data (executables, documentation, media, etc.) to be deployed on the file system. Additionally, binary packages are described by meta-information which include complex *inter-package relationships* that describe the static requirements to run properly on a target system. Requirements are expressed in terms of other binary packages, possibly with restrictions on the desired versions. Both positive requirements (*dependencies*) and negative requirements (*conflicts*) are supported by most packaging systems.

Example 2. *An excerpt of the meta-data of the* `ocamlnet` *library in Debian currently reads:*

```
1  Package: libocamlnet-ocaml-dev
2  Version: 2.2.9-3
3  Depends: ocaml-nox-3.10.2, ocaml-findlib, libocamlnet-ocaml (= 2.2.9-3),
4   libpcre-ocaml-dev (>= 5.11.1), libcryptgps-ocaml-dev (>= 0.2.1)
5  Provides: libequeue-ocaml-dev, libnetclient-ocaml-dev, librpc-ocaml-dev
6  Conflicts: libequeue-ocaml-dev (<< 2.2.3-1),
7   libnetclient-ocaml-dev (<< 2.2.3-1), librpc-ocaml-dev (<< 2.2.3-1)
8  Description: OCaml application-level Internet libraries - core libraries
```

In this short but representative example we can recognize the distinguishing features of inter-package relationships [6]:

- dependencies over other libraries, tools, and the OCaml interpreter (line 3) which can be both versioned (e.g. `libocamlnet-ocaml`—the runtime part of the package shown) or non-versioned (e.g. `ocaml-findlib`);
- conflicts with some libraries (line 6), in this case with superseded packages now included into `libocamlnet-ocaml-dev` itself;
- *virtual packages* (line 5), i.e. the ability to declare *features* provided by the owning "real" package so that others can depend on (or conflict with) feature names. In this case `libocamlnet-ocaml-dev` provides old library names such as `libequeue-ocaml-dev`; dependencies on it by other packages can be satisfied by installing PKGlibocamlnet-ocaml-dev. The other typical use case of virtual packages is to add an indirection layer about system services (e.g. `mail-transport-agent`) between packages providing the service (e.g. `postfix` or `exim4`) and packages needing it (e.g. `cron`).

Auto-building and binNMUs. *Source packages* are a different type of packages that contain the source code. Compiling them produces binary packages.[6] Source packages are usually manipulated by package *maintainers* working for specific distributions. The mapping between source and binary packages is one to many; for instance, the `ocamlnet` source package produces the package shown in Example 2 together with eight other binary packages. Also, there exists a relation of *build-dependency* between a source package and all binary packages which are needed to build it.

The natural work-flow of packages is rather complex [8], the part that mostly concerns us is sketched in Figure 2. Maintainers upload source packages to a package queue, together with the corresponding binary packages; considered as a whole, such an upload is said to be a *sourceful upload.* Given that maintainers usually own only a machine for a single architecture, *auto-builders* (or *buildd*) pick up the uploaded source package and rebuild the corresponding binary packages for each architecture supported by the distribution (about a dozen, in the case of Debian). The only exception to this scheme are *architecture independent* packages (or `arch:all`) that once built on any given architecture, can then be installed on all architectures (e.g. because they contain portable interpreted code or bytecode); hence, `arch:all` packages do not get rebuild at all. All obtained binary packages flow to the "unstable" *suite* which is ready to be used by developers and testers in preparation of the future stable release.

[6]According to folklore, we use "packages" to refer to binary packages, and explicitly "source packages" for the others.

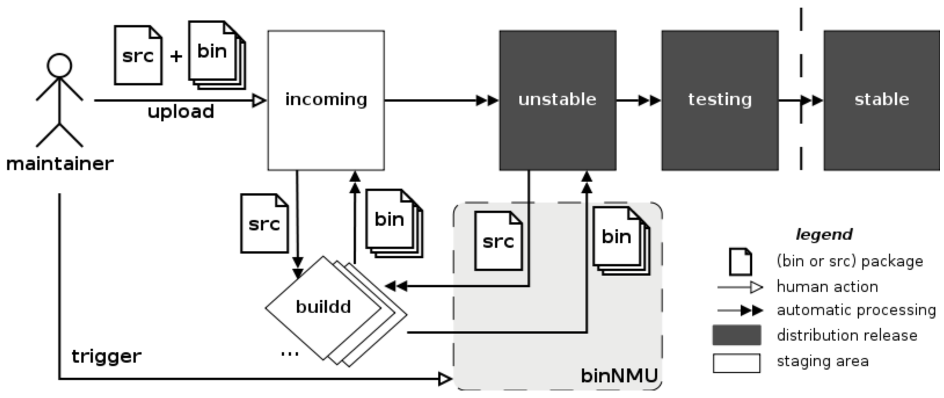

Figure 2: package (auto-)building in Debian.

In specific occasions (library transitions, etc.), binary packages can be rebuilt without intervention on their source packages. This might be needed to fix builds performed in malformed environments (e.g. buggy compiler) or to ensure a package still builds against a new version of a library it depends upon. To that end, maintainers can trigger rebuilds by requesting a *binNMU* (a historic and unfortunate name standing for "binary Non-Maintainer Upload"). For our needs, the important aspect of binNMUs is that they do not allow to perform any change to the source package.

Dependency inference. But how can a binary package change if its source package remains unchanged? The contents of the package (e.g. binary files) can of course change as the toolchain used to build changes (compiler, libraries, etc.), but it turns out that also package meta-information such as dependencies can change across rebuilds (and hence differ across architectures). In fact, maintainers can exploit a mechanism of *dependency inference* to ease the tedious task of maintaining dependency information. For instance, the dependency information of the `mldonkey-server` package read as follows in the corresponding *source* package:

```
Package: mldonkey-server
Depends: adduser, mime-support, ucf, ${shlibs:Depends}, ${misc:Depends}
```

whereas in the resulting binary package they read:

```
Package: mldonkey-server
Depends: adduser, mime-support, ucf, libbz2-1.0, libc6 (>= 2.3.2), libjpeg62,
 libfreetype6 (>= 2.2.1), libgcc1 (>= 1:4.1.1), zlib1g (>= 1:1.1.4),
 libgd2-noxpm (>= 2.0.36~rc1~dfsg) | libgd2-xpm (>= 2.0.36~rc1~dfsg),
 libpng12-0 (>= 1.2.13-4), libstdc++6 (>= 4.2.1), debconf | debconf-2.0
```

What is going on is that variables—expressed in `${this:form}`—get expanded during build according to dependency inference. While maintainers can use custom expansion mechanisms, the common way to expand variables is to rely upon specific *registries*. For instance, the `${shlibs:Depends}` variable expands to all dependencies that can be inferred by C shared library relationships [3]. To that end, each C shared library installed on the system provides a `shlibs` file which is in essence a record of tuples $\langle libraryname, packagename, versionpredicate \rangle$ which contributes to the C shared library registry. For instance, the `libpng12-0` package provides a record ⟨`libpng12`, `libpng12-0`, `(>= 1.2.13-4)`⟩. When the `mldonkey-server` package gets built, all executables it ships are inspected for relationships with C shared libraries (using tools like `objdump`). Since the `/usr/bin/mldonkey-server` executable needs a C library called `libpng12` to be loaded, a lookup for that library name in the registry provides the matching dependency `libpng12-0 (>= 1.2.13-4)` which is added to the expansion of `${shlibs:Depends}`.

3. Requirements

The goal of the solution we are looking for is to detect type-aware linking incompatibilities at the dependency level. As there are several possible solutions to that problem, we define in this section a set of contingent requirements, mostly inherited from the context. The first obvious requirement is goal fulfillment.

Requirement 1 (Dependency soundness)**.** *All binary packages shipping compilation units should satisfy the* dependency soundness *property.*

Property 1 (Dependency soundness)**.** [7] *A package p shipping OCaml compilation units $\{u_1, \ldots, u_n\}$ has* sound dependencies *with respect to a repository R if and only if for all healthy installation $I \ni p$, it holds that* $link(\{u \mid \exists q \in R|_{\{p\}} \quad q \in I \text{ and } q \text{ ships } u\}) = \checkmark$

Intuitively, a package has sound dependencies if all compilation units shipped by its (transitive) dependencies can be linked together. Dependency soundness is clearly not a local property—i.e. it cannot be decided by considering a package in isolation—but this is not surprising given that our notion of linking is geared towards obtaining a self-contained executable.

Requirement 2 (Dependency inference)**.** *Given a source package s and its build-dependencies $B = \{p_1, \ldots, p_n\}$, the dependencies that ensure soundness on all binary packages obtained by building s, should be* inferrable *on the basis of B and its (transitive) dependencies.*

Acknowledging that OCaml ABIs change very frequently, Requirement 2 asks for a mechanism of dependency inference (see Section 2) that relieves maintainers from the error-prone task of maintaining by hand the needed dependencies.

Requirement 3 (binNMU-safety)**.** *If a package p has sound dependencies, performing a binNMU on it should not make its dependencies unsound.*

This requirement not only means that inferred dependencies should not be tied to source package version, but also that we cannot rely on sourceful uploads. Hence, solutions requiring changes to be incorporated in source packages at each upload are not suitable for the task. All dependencies ensuring soundness should be recomputed during build, on the basis of the build environment. While binNMU-safety mimics the homonymous accepted best-practice of packaging, it is also a real need to reduce maintenance burden. Since transitions from one compiler version to the next require rebuilding all packages, and given that distributions like Debian contain nowadays more than a hundred OCaml-related source packages, the ability to do transitions via binNMUs is crucial for the maintainability of the whole stack in package-based binary distributions.

Requirement 4 (Light dependencies)**.** *All inter-package relationships needed to ensure dependency soundness should be terse, readable, human-manageable.*

Albeit admittedly very informal, this requirement is meant to ensure that package dependencies remain manageable by humans. Even if there is no clear definition for that, discussions within the Debian project between OCaml maintainers and both the release team[8] and users has given evidence that solutions like a simple dump of all MD5 requirements as dependencies would not be acceptable. Indeed, during release management, quality assurance, or simply broken dependency analysis within a package manager, such dependencies will be too many (one for each ABI requirement) and too unfriendly. Requirement 4 attempts to grasp this and similar desiderata.

[7] The notions of *healthy installation* and *generated subrepository*, noted $R|_{\Pi}$, are easily explainable: an installation is a set of installed packages, it is healthy if all dependencies are satisfied and no conflicts occur among installed packages; a generated subrepository is a repository subset obtained as the closure of the “depends on” relation starting from a given set of packages. Formal details can be found in [11].

[8] The release team is a group of Debian developers who coordinate transitions, goals and deadline for the next stable release.

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

Figure 3: Past Debian dependency scheme for OCaml-related packages

3.1. Related work

Until the adoption of the solution presented in this paper, the Debian distribution were using the natural dependency scheme [9] depicted in Figure 3, where the snippets of Figure 1 are considered shipped by packages with matching names. The drawbacks of that solution are twofold: no dependency inference (the maintainer writes dependencies by hand), no dependency soundness (there is no guarantee that future versions will preserve ABI compatibility). Note that stricter version predicates, such as e.g. `libfoo-ocaml-dev (>= 1), libfoo-ocaml-dev (<< 2)`, (as it is currently done for Haskell packages in Debian) will not provide soundness either because version numbers are densely ordered. In 2005, a helper tool called `dh_ocaml` was proposed [15] to ease the burden of maintaining OCaml-related dependencies,[9] mimicking the architecture of the `shlibs` registry for C shared libraries. Dependencies generated by (old-style) `dh_ocaml` follow the scheme of Figure 3. The obtained solution hence also fulfills Requirement 2, but not yet Requirement 1 (arguably the most important one).

The Fedora and Red Hat distributions, and after them other RPM-based distributions such as OpenSUSE and Mandriva, adopt a different solution [4]. At the end of build, they automatically inspect all OCaml bytecode objects u_i shipped by each binary package and, for each pair $\langle m, c\rangle \in \mathcal{R}(u_i)$, they add a dependency on a virtual package in the OCaml namespace which has the full MD5 checksum as its version; pairs in $\mathcal{A}(u_i)$ are similarly handled to generate the list of provided virtual packages. No intermediate registry is used. Fedora's solution provides soundness (up to a technical detail discussed in Section 5) and implicit dependency inference, however it fails lightness: as it can be seen in Figure 4, it bloats package lists with one virtual package/dependency per exported/required OCaml module (which can have a performance hit on dependency resolution, in distributions the size of Debian). Also, it will expose users and other team members to both human-meaningless and very long checksums.

The last alternative solution we are aware of is *ABI evolution tracking* [15]: it establishes an

[9] http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=328422, retrieved October 2009

```
$ rpm -qRp ocaml-ocamlnet*.rpm             $ rpm --provides -qp ocaml-ocamlnet*.rpm
ocaml(Arg) =                               ocaml(Equeue) =
 b6513be035dc9c8a458c189cd8841700           329e036bb2778b249d6763d22407af19
ocaml(Array) =                             ocaml(Ftp_client) =
 9c9fa5f11e2d6992c427dde4d1168489           d36822b105eacef219a2b6e0331ba34b
ocaml(Bigarray) =                          ocaml(Ftp_data_endpoint) =
 fc2b6c88ffd318b9f111abe46ba99902           f279805dc3b7ced5d8554f92e287c889
ocaml(Buffer) =                            ocaml(Generate) =
 23af67395823b652b807c4ae0b581211           418dedddda65b04bdc4d0c6e9fb918d4
... snipped 67 more ocaml(*) deps          ... snipped 110 more ocaml(*) virtual packages
```

Figure 4: Sample dependencies (on the left) and provided features (on the right) according to the Fedora solution, for the `ocamlnet` library.

Table 1: Review of linkability enforcement solutions

Solution	**Req. 1** **soundness**	**Req. 2** **inference**	**Req. 3** **binNMU**	**Req. 4** **lightness**
past Debian *status quo*	✗	✗	✓	✓
+ old-style `dh_ocaml`	✗	✓	✓	✓
current Fedora guidelines	✓	✓	✓	✗
ABI evolution tracking	✓	✓	✗	✓
ABI approximation	✓	✓	✓	✓

injective mapping between package ABIs and integers. The integer ideally represents ABI "version" and is used to establish human-friendly virtual package names (e.g. `libpcre-ocaml-dev-1`) that provide soundness. Unfortunately, this solution defeats binNMU-safety. Indeed, to avoid ABI version clashes the mapping should be either maintained by hand (similarly to what happens with C symbols tracking, where unexpected changes at build time trigger build failures), or obtained via automatic monotonic increase of ABI versions. For the latter, we would need to preserve somewhere a history of past ABI numbers, which has no place to stay during binNMUs: network is not accessible to avoid non-deterministic builds and source packages cannot be changed.

Table 1 reviews the discussed solutions against our requirements. The last line is the solution we propose, described in next section.

4. ABI approximation

To overcome the limitations of the discussed approaches, we have devised a solution based on the idea of computing a single approximation of the ABIs of all compilation units shipped by a given package. We call such solution *ABI approximation* and its architecture is sketched in Figure 5. Consider a (binary) package *pkg* shipping compilation units $u_1, \ldots, u_n$ (at the top-left of Figure 5). Its ABI approximation—noted $\widetilde{\mathcal{A}}(pkg)$—is obtained as a cryptographic hash of all ABI pairs of all compilation units; intuitively: $\widetilde{\mathcal{A}}(pkg) = hash(\mathcal{A}(u_1) \circ \cdots \circ \mathcal{A}(u_n))$, where $\circ$ is a concatenation operator which is used to incrementally adds material on which the hash is taken.[10]

The obtained ABI approximation is then used in two different ways, both eventually contributing to form the inter-package relationships of *pkg*. For what concerns the interface exposed by *pkg*, we directly add a virtual package obtained by juxtaposing *pkg*'s name with the ABI approximation itself. All packages relying on the set of ABIs $\mathcal{A}(u_1), \ldots, \mathcal{A}(u_n)$ will depend on that virtual package.

Let's now assume that all dependencies of *pkg* itself already have computed ABI approximations and that they are available at build-time (a sound assumption according to our linking discipline). To ensure that we can compute the dependencies on the correct virtual packages—i.e. the virtual packages corresponding to packages able to satisfy the linking assumptions $\mathcal{R}(u_1), \ldots, \mathcal{R}(u_n)$—we use an *ABI registry* that stores information about which packages provide which ABI. The registry is populated by tuples (simplified in Figure 5) $\langle m, c, pkg, \widetilde{\mathcal{A}}(pkg) \rangle$ where $\langle m, c \rangle \in \mathcal{A}(u)$ for some u shipped by *pkg*. Intuitively, each module m exposed by some compilation unit with checksum c has a tuple in the registry which relates it with a specific package name and overall ABI approximation. Technically, the tuples are computed at build-time and the registry is constructed incrementally by files shipped by *pkg* itself.

Using the registry, we can now compute the dependency entries of *pkg* which will ensure its

[10] A more formal description would require more details on the hashing function, on module naming conventions, etc. They are omitted from here for the sake of conciseness, but Section 5 provides some related details about the current implementation.

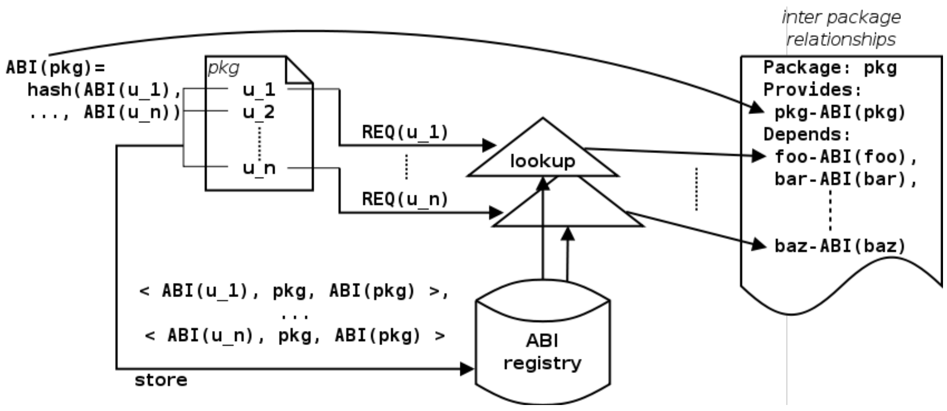

Figure 5: ABI approximation: architecture

linkability. For each $\langle m, c\rangle \in \mathcal{R}(u_i)$, we look up $\langle m, c\rangle$ (which is expected to be a primary key in any possible configuration) in the registry to get a pair of package name and ABI approximation. By juxtaposing them as before, we can now emit the appropriate dependency snippets.

Example 3. *The resulting dependencies for the* `ocamlnet` *package, which we have already seen* before *ABI approximation in Example 2, is as follows.*

```
Package: libocamlnet-ocaml-dev
Version: 2.2.9-7
Depends: ocaml-findlib, libcryptgps-ocaml-dev-139d7, libocamlnet-ocaml-3rxe6,
 libpcre-ocaml-dev-kh2c0, ocaml-nox-3.11.1
Provides: libequeue-ocaml-dev, libnetclient-ocaml-dev, librpc-ocaml-dev,
 libocamlnet-ocaml-dev-3rxe6
Description: OCaml application-level Internet libraries - core libraries
```

We can notice that the package depends on the external `cryptgps` *and* `pcre` *library with specific ABI approximations, and that it exports an ABI approximation itself. By comparison with the Fedora dependencies for the same package (see Figure 4), we observe that the huge amount of provided and expected module checksums is here hidden in the registry and resolved at build time; the package interface exposes a single extra virtual package, and one dependency for each external library needed.*

To review ABI approximation against the requirements of Section 3, we start by observing that soundness is granted up to the risk of clashes of different sets of ABIs to the same approximated ABI. Literature about so-called “birthday paradox” gives an approximation on the number of different, incompatible, versions of a library to have collision with probability p:

$$n(p; H) \approx \sqrt{2H \ln \frac{1}{1-p}}$$

were H denotes the ABI approximation space size. In our practical implementation $H = 36^5$ (see next section), so we reach the negligible probability of 1 % of collision with 1103 versions.

Dependency inference is trivially provided by the ABI registry, according to the lookup mechanism described above. binNMU-safety is granted too as all the state needed to compute both the inferred dependencies and the exposed approximated ABI is not stored in the source package (and hence does not need sourceful uploads to be changed); rather, the state is kept in the registry which is contributed to by build-dependencies of the package being built. Finally, arguably dependency lightness is provided too. While ABI approximated strings are not necessarily human-meaningful, they have the benefit of being short (only 5 alphanumeric characters in the current implementation). Also, the number of such strings that are needed for a given binary package is bounded: one for the exported interface and one for each package dependency, with the latter set replacing dependencies which would have been needed anyhow, just replacing ABI approximations with version numbers.

The most notable limitation of ABI approximation is the "instability" of approximations with respect to backward *compatible* changes. This is not surprising, given that the solution has been designed precisely to cope with frequent backward *in*compatible changes. Still, sometimes, backward compatible changes can happen in a given package, for example by adding a new module to a given library without touching any other existing module. In our formalism that translates to adding a new couple $\langle m_{\text{new}}, c_{\text{new}} \rangle$ to some $\mathcal{A}(u_i)$; trivially, property 1 is unaffected by such addition. Nevertheless, the resulting ABI approximation will change, since the content on which the hash is computed will change. Practically, the problem is non-existent, given the low frequency of such changes and given that—thanks to binNMU-safety—the affected packages can be fixed by just scheduling their recompilation. Also, we observe that this issue can be fixed only by allowing to export ABI approximations whose sizes are linear with respect to the number of modules shipped by a given package; in essence, that would mean mimicking Fedora's solution, getting back its disadvantages.

5. Implementation

ABI approximation has been implemented in the context of the Debian distribution to manage the inter-package dependencies of all provided OCaml-related packages. At the time of writing, that amounts to 158 source packages that build-depend on OCaml, producing 353 binary packages, and providing interfaces for 2 502 OCaml modules. Technically, the implementation is based on `dh_ocaml`, a helper tool meant to be compatible with the Debhelper packaging helper suite [5]. Some of its inner workings depends on how OCaml libraries are split into binary packages, which we briefly highlight below.

Compiling a source package results in one or several binary packages. For example, library source packages usually produce (at least) two binary packages:

1. a *runtime* package, containing all the objects that might be needed at runtime by software linked to the library;
2. a *development* package, containing all the objects that might be needed to compile software that uses the library.

In the OCaml world, the runtime package typically contains stubs to C libraries (the so called "bindings") that are dynamically loaded when running pure bytecode executables (`.so`), and the development package contains interfaces in source (`.mli` or `.ml`) and compiled (`.cmi`) form, along with compiled module objects (`.cmo`, `.cma`, `.cmx`, `.o`, `.cmxa`, `.a`), `META` files and (mostly `ocamldoc`-generated) documentation. A library might have no runtime package at all if it is meant to be always statically linked, which happens quite often with pure-OCaml libraries. However, when a library can be dynamically loaded by a program that supports plugins, the runtime package can also provide `.cma`, `.cmxs`, and `META` files.

A library can be further split into several components. This is usually left at the package maintainer discretion, and choices can vary between distributions. A typical reason for splitting a library is optional dependencies that might be needed only by some compilation units. Each resulting component might then have its runtime and development packages. Finally, there are also non-library packages shipping OCaml binaries. Overall, `dh_ocaml` distinguishes three classes of (binary) packages: runtime, development, and others. We henceforth consider as a *library* a development package together with its runtime package, if any. For example, the `ocamlnet` source package produces the following binary packages:

`libocamlnet-ocaml` core library (runtime);

`libocamlnet-ocaml-dev` core library (development);

`libocamlnet-ocaml-doc` documentation;

`libocamlnet-ocaml-bin` miscellaneous tools;

`libocamlnet-gtk2-ocaml-dev` GTK2 layer (development);

`libocamlnet-ssl-ocaml` SSL layer (runtime);

`libocamlnet-ssl-ocaml-dev` SSL layer (development);

`libnethttpd-ocaml-dev` HTTP daemon libraries (development);

`libapache2-mod-ocamlnet` Apache2 module.

Among them, four OCaml libraries can be recognized: `libocamlnet-ocaml/-dev`, `libocamlnet-gtk2-ocaml-dev` (without runtime), `libocamlnet-ssl-ocaml/-dev`, and `libnethttpd-ocaml-dev` (without runtime).

To inspect OCaml compilation units (a required ability to implement ABI approximation), the OCaml standard distribution provides some useful tools:

`ocamlobjinfo` reads assumptions of bytecode objects (`.cmi`, `.cmo` and `.cma` files), as shown in Example 1;

`ocamldumpapprox` reads assumptions of native code objects (`.cmx`, `.cmxa` files).

While OCaml bytecode objects make assumptions only on interfaces, native code objects may also make assumptions on implementations, e.g. due to inlining across compilation units. These assumptions are stored in `.cmx` files, which might hence be useful even if the module they represent are included in a `.cmxa` file. Most notably, the above tools lack the ability to inspect bytecode executables and native plugins (`.cmxs`). Such abilities are much needed: bytecode executables might depend on external C stubs, and a native plugin is a special case of library and should be considered as such. To overcome these limitations, we have implemented `ocamlbyteinfo` and `ocamlplugininfo`; they are currently shipped as part of the `ocaml` source package and relied upon by `dh_ocaml`.

The solution described in section 4 is then implemented in Debian using two tools:

`ocaml-md5sums` manages the ABI registry. This tool is meant to be generic (not Debian-specific). It inspects a set of object files by calling the most appropriate among the above tools, uses their output to compute their dependencies and its approximated ABI. It considers a library as a whole (runtime and development together);

`dh_ocaml` is a frontend to `ocaml-md5sums`; it looks for installed objects and binaries, passes them to `ocaml-md5sums` and fills-in substitution variables corresponding to dependency inference variables; it distinguishes between the three categories of packages:

- development packages depend on their own runtime package (that explains the dependency on `libocamlnet-ocaml-3rxe6` that the attentive reader might have noticed in Example 3), and possibly on other development packages;
- runtime and other packages depend only on other runtime packages.

The actual ABI approximation is computed by considering the union of two different sets of $\mathcal{A}(\cdot)$ for each compilation unit: one for assumptions over interfaces, one for assumptions over implementations (no matter where they come from: plugin, object, or binary). On a simple textual representation of the resulting set, a MD5 checksum is computed and then taken modulo a fixed hash space. Currently, such hash space relies on 5 alphanumeric, lowercase, plain-ASCII characters, accounting for $(26 \text{ letters} + 10 \text{ digits})^5 \approx 60 \cdot 10^6$ different ABI approximations. As discussed in the previous section, that choice gives a negligible clash probability, without sacrificing lightness.

We notice that alternative solutions, such as Fedora's, are currently not considering assumption on implementations (most likely because the original tools in the OCaml distributions were not able to inspect them). As a consequence they only provide soundness up to implementation incompatibility, which is usually more likely to occur than interface incompatibility. This being a minor drawback easily fixable (now that we have provided the missing tools), we also observe that fixing it naively would mean doubling the already large number of depends/provides of Figure 4.

Example 4. *Here is an excerpt of dependencies from the* `ocamlnet` source *package (i.e. before expansion by* `dh_ocaml`*):*

```
Package: libocamlnet-ocaml-dev
Provides: libequeue-ocaml-dev, libnetclient-ocaml-dev, librpc-ocaml-dev,
 ${ocaml:Provides}
Depends: ocaml-findlib, ${ocaml:Depends}, ${shlibs:Depends},
 ${misc:Depends}
```

The result of the substitution performed by `dh_ocaml` *in the corresponding binary package can be seen in Example 3.*

Both `ocaml-md5sums` and `dh_ocaml` are implemented in Perl to avoid (build-)dependency cycles among the package shipping them and the `ocaml` package itself. Both tools are part of the `dh-ocaml` source package, which is distributed under the terms of the GNU General Public License (version 3) and available from its Git repository.[11]

So far, we have effectively deployed this solution in about 50 source packages (out of 100 library packages). This deployment revealed not only past packaging errors such as missing dependencies, but also unrelated errors like libraries re-exporting modules they do not own. The most common example of that has been the embedding of the `Unix` module into third-party libraries. The remaining packages, and most notably libraries, are being ported to `dh_ocaml` at the time of writing, their ported versions are all expected to be shipped with the next stable release of Debian (codename "Squeeze").

The only currently known `dh_ocaml`-specific limitations are as follows:

- it is theoretically possible to change C stubs in a binary incompatible way while keeping the same interface; it is therefore possible to install an outdated version of C stubs while satisfying all dependencies of a bytecode executable. This limitation is inherent to OCaml;

- architecture-independent packages that depend on libraries with runtime cannot currently benefit from this solution, because the ABI approximation string depends on the architecture (given assumptions on implementations are not present on architectures lacking the `ocamlopt` native code compiler). This limitation can be solved by computing two separate ABIs: a native code and a bytecode one; the latter should be the same on all architectures.

[11] http://git.debian.org/?p=pkg-ocaml-maint/packages/dh-ocaml.git

6. Conclusions and future work

In this paper we have given some insights on the gap existing between fine-grained dependencies that type-aware linkers consider to enforce type-safety at compilation units borders, and the coarse-grained dependencies that can be expressed among packages in FOSS binary distributions. That gap persisting, package managers will not be able to defend users from installing OCaml libraries that can not be linked together due to annoying "`inconsistent assumptions`" link-time errors. We have then introduced a solution called *ABI approximation* that guarantees link-time compatibility as long as inter-package relationships are satisfied. ABI approximation satisfies a set of natural requirements inherited from the context: dependency soundness, inference, and lightness, as well as binNMU-safety.

ABI approximation has been implemented and deployed in the Debian distribution implementing the `dh_ocaml` helper, aiming at managing the dependencies of more than 300 OCaml-related binary packages.

Future work consists in some technical fixes whose need was revealed by using `dh_ocaml`, such as the observation that computing a single ABI for bytecode and native-code is incompatible with architecture-independent packages. Also, we are considering designing a mechanism for computing automatically OCaml-related ABIs for C stubs whose purpose is interacting with the OCaml runtime. That can be done at C stub compilation-time (where and when the interface of the matching OCaml objects are known) and then stored in some section of the resulting object.

References

[1] A.W. Appel and D.B. MacQueen. Separate compilation for Standard ML. In *PLDI 1994: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 13–23. ACM, 1994.

[2] Roberto Di Cosmo, Paulo Trezentos, and Stefano Zacchiroli. Package upgrades in FOSS distributions: Details and challenges. In *HotSWup'08: Proceedings of First ACM Workshop on Hot Topics in Software Upgrades*. ACM, 2008.

[3] Ulrich Drepper. How to write shared libraries. `http://people.redhat.com/drepper/dsohowto.pdf`, 2002. Revision August 2006.

[4] Fedora Project. OCaml packaging guidelines. `http://fedoraproject.org/wiki/Packaging/OCaml`. Retrieved October 2009.

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

[6] Ian Jackson and Christian Schwarz. Debian policy manual. `http://www.debian.org/doc/debian-policy/`, 2009.

[7] S. C. Johnson. Lint, a C program checker. (65), 1978.

[8] Martin F. Krafft. *The Debian system: concepts and techniques*. Open Source Press, 2005.

[9] 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.

[10] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL'94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122. ACM, 1994.

[11] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006: 21st IEEE/ACM Internation Conference on Automated Software Engineering*, pages 199–208. IEEE CS Press.

[12] L. Presser and J.R. White. Linkers and loaders. *Computing Surveys (CSUR)*, 4(3):149–167, 1972.

[13] Diomidis Spinellis. Type-safe linkage for variables and functions. *SIGPLAN Notices*, 26(8):74–79, 1991.

[14] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, 1997.

[15] Stefano Zacchiroli. Enforcing OCaml link-time compatibility using Debian dependencies. Debian wiki: http://wiki.debian.org/Teams/OCamlTaskForce?action=AttachFile&do=get&target=dh-ocaml-design.pdf, January 2009. Design document, retrieved October 2009.