OCaml Package Management with (only!) Dune

Stephen Sherratt[1], Marek Kubica[1], and Rudi Grinberg

[1]Tarides

2025-06-26

Abstract

The OCaml build system Dune keeps track of a project's dependencies on external software packages. Historically however, Dune has been unable to download or install these packages completely independently, relying on additional tools to perform these functions. This complicated the development of projects in OCaml as users needed to be fluent in both Dune as well as an additional tool (often opam) to manage a project's dependencies. Recent work on Dune has added package management capabilities directly to the build system, enabling workflows where Dune is the only tool necessary to develop software in OCaml.

This talk will showcase Dune's new package management features by developing an OCaml program starting from a bare system with no OCaml tooling installed.

1 Dune and Opam

Dune[1] is a build system for OCaml projects. It creates executables and libraries by compiling and linking OCaml source code from multiple files, tracking inter-file dependencies to allow for incremental recompilation. Dune knows how to locate OCaml libraries external to a project by understanding the *findlib*[2] file formats. Dune allows a project to be organized into one or more *packages* using a packaging scheme that approximates that of the *opam*[3] package manager. This is no coincidence; Dune can generate metadata files for packages in the format expected by opam to simplify the process of publishing packages from Dune projects on opam's default package repository[4]. Dune's own metadata files enumerate the packages depended upon by the project for this reason.

Opam metadata files generated by Dune have historically served a second purpose: to allow developers of a Dune project to use opam to install all of the software packages necessary to build their project locally. Dune's reliance on opam as its de facto standard package management tool has led to confusion among developers, in particular newcomers from other ecosystems where a single tool manages both source tree builds and dependency installation. Dune and opam have different user interfaces and file formats, and require different mental models, so familiarity with one tool does not contribute to one's understanding of the other. The functionality of both tools overlaps, since by merit of opam being a source-based package manager, it must also be able to build OCaml projects, exacerbating users' confusion about the relationship between the two tools. Thus it is rare to find someone with sufficient fluency in both Dune and opam to use the two tools in tandem to smoothly setup and maintain a project's development environment, even among the developers of these tools.

2 Dune Package Management

Due to the fact that Dune's project metadata files already enumerate the opam packages necessary to build a project (historically for the purpose of generating a corresponding opam metadata file) and that Dune is already capable of locating external libraries used in a project, an opportunity presented itself to retrofit package management into Dune in such a way that existing Dune projects can take advantage of without modification. This work commenced in 2023

[1]`https://github.com/ocaml/dune`
[2]`https://projects.camlcity.org/projects/findlib.html`
[3]`https://opam.ocaml.org`
[4]`https://github.com/ocaml/opam-repository`

and is now in a usable state, where it's possible to install Dune from a *binary distribution*[5] and develop OCaml projects with no installation of opam or the OCaml compiler being necessary.

Indeed we envision a development workflow where the *only* globally-installed OCaml tool is Dune, installed from its binary distribution or by a system package manager. Any additional OCaml-related software, be it a library, the OCaml compiler itself or developer tooling such a code formatter or LSP server, will be installed and *invoked* by Dune. This simplifies the developer experience for OCaml by making the `dune` command a common entry point for all tasks.

The following listing shows a terminal session building a project using Dune package management:

```
$ dune pkg lock
$ dune build
```

The first command computes the transitive closure of the project's dependencies, creating or updating a copy of this information within the project. This command only needs to be run after the project's dependencies change. The second command builds the project, first downloading and building its dependencies as necessary.

While opam's command-line tool is no longer necessary for managing the dependencies of projects, opam continues to play several roles within the OCaml ecosystem:

- As the opam package repository contains a rich set of OCaml packages it still is a good location to submit and load packages from. Thus it remains the default repository Dune uses when resolving and installing packages.
- Dune uses opam's internal libraries to parse package metadata files and to evaluate dependency formulae.
- Opam's command-line tool is still the only way to globally install opam packages and to query the package repository. It remains an open question among Dune's developers whether this functionality should be added to Dune.

3 Lock Directories

Packages in the opam repository conventionally do not constrain the upper bound of versions of their dependencies. Within the opam repository itself, a manual process prevents API changes from breaking released packages by *adding* constraints where necessary when new packages are released. However these additions are rarely backported to the development branches of projects, and projects that are not released to the opam repository are not protected from breakages due to API changes if they don't adopt a similar policy.

Dune addresses this problem with the introduction of *lock directories*. The command `dune pkg lock` computes the transitive dependency closure of the project, including *concrete* version numbers for each dependency, and this information is stored in files in a directory (typically `dune.lock`) which can be checked into version control and thus shared between the contributors to a project. This makes it very likely that if a project builds on one machine, then it will build on another machine, and freezes the versions of all dependencies, preventing unexpected API changes from breaking the project as new package releases are published.

3.1 Solving Dependencies

OCaml does not support multiple different versions of the same library being included in a build artifact. Dune must choose at most one version of each package such that a project's dependency constraints are satisfied, which is equivalent to a satisfiability problem. Opam's built-in constraint solver is optimized for making minimal updates to an existing set of installed packages, whereas Dune generates dependency solutions from scratch. Thus rather than using opam's solver we elected to base our solver off *0install*[6] as it's more suitable for Dune's use case.

3.2 Portable Lock Directories

Opam packages may have different dependencies depending on properties of the machine where they will be installed, such as its CPU architecture or operating system. In order for it to be safe to check lock directories into version control, it must be the case that generation of the lock directory is agnostic to the machine where the lock directory is being generated. Otherwise collaborators on a project with different types of system will not be able to share a single checked-in lock directory effectively.

Dune's dependency solver is currently unsophisticated with regard to generating portable lock directories, opting to solve the entire dependency problem once for each of a prescribed list of platforms, and

[5] `https://github.com/ocaml-dune/dune-bin-install`
[6] `https://github.com/0install/0install`

then merging the results. The list of platforms can be controlled by an entry in a config file. We find that in practice the performance cost of this repeated work is acceptable, though we may investigate alternative approaches to generating portable lock directories in the future. Of particular note is the Python package manager *uv*[7] which solves a similar problem in the Python package ecosystem.

4 Installing the Compiler

In the opam ecosystem the OCaml compiler is a mostly-regular package, as the system is designed in a way where other languages could be supported as well. However, if the OCaml compiler is necessary when building an opam package (as is most often the case), the compiler should be among that package's dependencies.

The OCaml compiler is implemented in such a way that the absolute path to its install location is included in its executable. This means that once installed, the compiler's executable and related files cannot be moved to another location. This poses a particular problem due to the way Dune builds packages.

Dune maintains a shared cache of built artifacts that can be reused across projects to speed up builds. For artifacts to be safely cached, it's necessary that the only files consumed when generating them are their stated build inputs, and not other files from the project or wider file system. To this end Dune builds each artifact (including packages) in a transient sandbox environment where only the artifact's dependencies are present, before moving the artifacts into the final directory for built artifacts (and also copying them into the shared cache). The compiler cannot be built in this manner because the paths included in its executable would refer to the transient sandbox where the compiler was built.

As a workaround, Dune treats the compiler package specially, patching its package metadata at build-time so that it is installed to a location within the user's home directory (typically `$HOME/.cache/dune/toolchains`). Multiple versions of the compiler can be co-installed to this directory, and Dune reuses compilers when possible rather than building them anew.

It's ongoing work to make it possible to relocate the compiler after installation[8], after which point this workaround can be removed from Dune.

[7] `https://github.com/astral-sh/uv`

[8] `https://www.youtube.com/watch?v=5JDSUCx-tPw`

5 Developer Tools

Developers typically require tools besides the build system and compiler to be productive. Dune contains a mechanism for installing and running developer tools. Each developer tool corresponds to an opam package, and Dune installs tools using the same package management mechanism as for regular project dependencies.

Developer tools installed by Dune are not installed in the same package environment as the project or as each other, thus the dependencies of developer tools do not have to be compatible with the dependency closure of the project itself, nor with the dependency closure of other developer tools. This is a major improvement over the previous opam workflow where all developer tools were installed into the same package environment as the project's dependencies which sometimes created conflicts between the dependencies of the projects and tools.

Developer tools managed by Dune are installed locally within the project, similar to installing tools into a local opam switch. This allows Dune to ensure that tools will be compatible with the project. For example OCaml's LSP server must be compiled with the same OCaml compiler as the code which it will be analyzing. Dune can use its knowledge of a project's dependencies to choose the appropriate compiler with which to compile the LSP server for that project.

6 System Packages

Opam contains a mechanism for installing non-opam packages using the system's package manager (`apt`, `brew`, `cygwin`, etc). An opam package may declare an external dependency (*depext*) by specifying the name of the external package in as many package ecosystems as possible, as packages may be referred to by different names by different package managers. Opam knows how to interface with many package managers to install depexts automatically.

Dune doesn't install depexts directly, however a project can be queried for a list of all depexts among its transitive dependency closure. Dune detects which system it is running on when computing this list allowing depext names to be tailored to be compatible with the current machine's system package manager.

In time Dune may be extended to print the command which would install all the depexts of a project using the current machine's package manager. An-

other possible extension would be for Dune to *run* this command. This would reduce a possible point of friction for users, however mutating the global state of a user's computer requires a great deal of care so this functionality is intentionally omitted for the time being.

7 Future Work

There are several features currently missing from Dune's implementation of package management that either prevent its use for some projects, or require an installation of opam:

- **Dune cannot build projects with circular dependencies via test-only dependencies.** This affects a small number of widely used low-level packages including Dune itself. Dune uses several external packages for its tests (such as *ppx_expect*[9]), and some of these packages depend on Dune. This should be a benign circular dependency as Dune does not require these packages at build-time, however due to implementation details Dune cannot currently handle such a case.

- **Portable lock directories are disabled by default.** Dune's default behaviour is currently to generate lock directories that are specialized to the platform where they were generated, making them unsafe to check into version control. Work is ongoing to stabilize this feature so it can be enabled by default.

- **Dune lacks commands for querying package repositories**. The only way to search for a package or print out a package's metadata from the command-line is to use `opam`.

[9] `https://github.com/janestreet/ppx_expect`