Dune Documentation

Jérémie Dimino

Jun 18, 2024

CONTENTS

1 Getting Started and Core Concepts **3**
- 1.1 Overview . . . 3
- 1.2 Quickstart . . . 5
- 1.3 Command-Line Interface . . . 12

2 How-to Guides **21**
- 2.1 How to Install Dune . . . 21
- 2.2 How to Set up Automatic Formatting . . . 22
- 2.3 How to Generate Opam Files from `dune-project` . . . 23
- 2.4 Cross-Compilation . . . 25
- 2.5 Dealing with Foreign Libraries . . . 26
- 2.6 Generating Documentation . . . 32
- 2.7 How to Load Additional Files at Runtime . . . 34
- 2.8 Instrumentation . . . 39
- 2.9 JavaScript Compilation With Js_of_ocaml . . . 40
- 2.10 JavaScript Compilation With Melange . . . 41
- 2.11 Virtual Libraries . . . 45
- 2.12 Writing and Running Tests . . . 47
- 2.13 How to Bundle Resources . . . 57
- 2.14 How to Load a Project in a Toplevel . . . 57
- 2.15 Using Rule Generation . . . 58

3 Reference Manual **63**
- 3.1 dune . . . 63
- 3.2 dune-project . . . 88
- 3.3 dune-workspace . . . 98
- 3.4 config . . . 100
- 3.5 Lexical Conventions . . . 102
- 3.6 Actions . . . 104
- 3.7 Ordered Set Language . . . 110
- 3.8 Boolean Language . . . 111
- 3.9 Predicate Language . . . 111
- 3.10 Library Dependencies . . . 111
- 3.11 Preprocessing Specification . . . 112
- 3.12 Cram Tests . . . 114
- 3.13 Scopes . . . 117
- 3.14 Variables . . . 118
- 3.15 Dependency Specification . . . 121
- 3.16 OCaml Flags . . . 123
- 3.17 Sandboxing . . . 123

3.18 Locks . . . 124
3.19 Diffing and Promotion . . . 125
3.20 Package Specification . . . 126
3.21 Aliases . . . 128
3.22 Foreign Sources, Archives, and Objects . . . 131
3.23 Command Line Interface . . . 134
3.24 Dune Libraries . . . 137
3.25 Dune Cache . . . 139
3.26 Coq . . . 141
3.27 Dune RPC . . . 153
3.28 Packages . . . 154
3.29 Findlib Integration . . . 155

4 Explanation **157**
4.1 How Preprocessing Works . . . 157
4.2 The OCaml Ecosystem . . . 158
4.3 How Dune integrates with opam . . . 159
4.4 How Dune Uses Dune to Build Dune . . . 161
4.5 The Dune Mental Model . . . 162
4.6 A Tour of the Dune Codebase . . . 165

5 Advanced Topics **173**
5.1 Dynamic Loading of Packages with Findlib . . . 173
5.2 Profiling Dune . . . 174
5.3 Package Version . . . 174
5.4 OCaml Syntax . . . 174
5.5 Variables for Artifacts . . . 174
5.6 Building an Ad Hoc `.cmxs` . . . 175

6 Miscellaneous **177**
6.1 FAQ . . . 177
6.2 Goal of Dune . . . 181
6.3 Working on the Dune Codebase . . . 183

Index **195**

Dune is a build system for OCaml projects. Using it, you can build executables, libraries, run tests, and much more.

CHAPTER

ONE

GETTING STARTED AND CORE CONCEPTS

These documents should be the first ones read by new Dune users. They explain what Dune is, how it works, and how to use it.

1.1 Overview

1.1.1 Introduction

Dune is a build system for OCaml (with support for Reason and Coq). It is not intended as a completely generic build system that's able to build any project in any language. On the contrary, it makes lots of choices in order to encourage a consistent development style.

This scheme is inspired from the one used inside Jane Street and adapted to the opam world. It has matured over a long time and is used daily by hundreds of developers, which means that it is highly tested and productive.

When using Dune, you give very little, high-level information to the build system, which in turn takes care of all the low-level details from the compilation of your libraries, executables, and documentation to the installation, setting up of tests, and setting up development tools such as Merlin, etc.

In addition to the normal features expected from an OCaml build system, Dune provides a few additional ones that separate it from the crowd:

- You never need to tell Dune the location of things such as libraries. Dune will discover them automatically. In particular, this means that when you want to reorganise your project, you need nothing other than to rename your directories, Dune will do the rest.
- Things always work the same whether your dependencies are local or installed on the system. In particular, this means that you can insert the source for a project dependency in your working copy, and Dune will start using it immediately. This makes Dune a great choice for multi-project development.
- Cross-platform: as long as your code is portable, Dune will be able to cross-compile it. Read more in the *Cross-Compilation* section.
- Release directly from any revision: Dune needs no setup stage. To release your project, simply point to a specific Git tag (named revision). Of course, you can add some release steps if you'd like, but it isn't necessary. For more information, please refer to dune-release.

The first section below defines some terms used in this manual. The second section specifies the Dune metadata format, and the third one describes how to use the `dune` command.

1.1.2 Terminology

root
The top-most directory in a GitHub repo, workspace, and project, differentiated by variables such as `%{workspace_root}` and `%{project_root}`. Dune builds things from this directory. It knows how to build targets that are descendants of the root. Anything outside of the tree starting from the root is considered part of the *installed world*. Refer to *Finding the Root* to learn how the workspace root is determined.

workspace
The subtree starting from each root. It can contain any number of projects that will be built simultaneously by Dune, and it must contain a `dune-workspace` file.

project
A collection of source files that must include a `dune-project` file. It may also contain one or more packages. A project consists in a hierarchy of directories. Every directory (at the root, or a subdirectory) can contain a `dune` file that contains instructions to build files in that directory. Projects can be shared between different applications.

package
A set of libraries and executables that opam builds and installs as one.

installed world
Anything outside of the workspace. Dune doesn't know how to build things in the installed world.

installation
The action of copying build artifacts or other files from the `<root>/_build` directory to the *installed world*.

scope
Defined by any directory that contains at least one *<package>.opam* file. Typically, every project defines a single scope that is a subtree starting from this directory. Moreover, scopes are separate from your project's dependencies. The scope also determines where private items are visible. Private items include libraries or binaries that will not be installed. See *Scopes* for more details.

build context
A specific configuration written in a *dune-workspace* file, which has a corresponding subdirectory in the `<root>/_build` directory. It contains all the workspace's build artifacts. Without this specific configuration from the user, there is always a `default` build context that corresponds to the executed Dune environment.

build context root
The root of a build context named `foo` is `<root>/_build/<foo>`.

build target
Specified on the command line, e.g., `dune build <target_path.exe>`. All targets that Dune knows how to build live in the `_build` directory.

alias
A build target that doesn't produce any file and has configurable dependencies. Targets starting with `@` on the command line are interpreted as aliases (e.g., `dune build @src/runtest`). Aliases are per-directory. See *Aliases*.

environment
Determines the default values of various parameters, such as the compilation flags. In Dune, each directory has an environment attached to it. Inside a scope, each directory inherits the environment from its parent. At the root of every scope, a default environment is used. At any point, the environment can be altered using an *env* stanza.

build profile
A global setting that influences various defaults. It can be set from the command line using `--profile <profile>` or from `dune-workspace` files. The following profiles are standard:

- `release` which is the profile used for opam releases
- `dev` which is the default profile when none is set explicitly, it has stricter warnings than the `release` one

dialect
: An alternative frontend to OCaml (such as ReasonML). It is described by a pair of file extensions, one corresponding to interfaces and one to implementations. It can use the standard OCaml syntax, or it can specify an action to convert from a custom syntax to a binary OCaml abstract syntax tree. It can also specify a custom formatter.

placeholder substitution
: A build step in which placeholders such as `%%VERSION%%` in source files are replaced by concrete values such as `1.2.3`. It is performed by *dune subst* for development versions and dune-release for releases.

1.1.3 Project Layout

A typical Dune project will have a `dune-project` and one or more `<package>.opam` files at the root as well as `dune` files wherever interesting things are: libraries, executables, tests, documents to install, etc.

We recommended organising your project to have exactly one library per directory. You can have several executables in the same directory, as long as they share the same build configuration. If you'd like to have multiple executables with different configurations in the same directory, you will have to make an explicit module list for every executable using `modules`.

1.1.4 History

Dune started as `jbuilder` in late 2016. When its 1.0.0 version was released in 2018, the name has been changed to `dune`. It used to be configured with `jbuild` and `jbuild-workspace` files with a slightly different syntax. After a transition period, this syntax is not supported anymore.

1.2 Quickstart

This document gives simple usage examples of Dune. You can also look at examples for complete examples of projects using Dune with CRAM stanzas.

To try these examples, you will need to have Dune installed. See *How to Install Dune*.

1.2.1 Initializing Projects

The following subsections illustrate basic usage of the `dune init proj` subcommand. For more documentation, see *Initializing Components* and the inline help available from `dune init --help`.

Initializing an Executable

To initialize a project that will build an executable program, run the following (replacing `project_name` with the name of your project):

```
$ dune init proj project_name
```

This creates a project directory that includes the following contents:

```
project_name/
├── dune-project
├── test
```

(continues on next page)

(continued from previous page)

```
    ├── dune
    └── test_project_name.ml
├── lib
    └── dune
├── bin
    ├── dune
    └── main.ml
└── project_name.opam
```

Now, enter your project's directory:

```
$ cd project_name
```

Then, you can build your project with:

```
$ dune build
```

You can run your tests with:

```
$ dune test
```

You can run your program with:

```
$ dune exec project_name
```

This simple project will print "Hello World" in your shell.

The following itemization of the generated content isn't necessary to review at this point. But whenever you are ready, it will provide jump-off points from which you can dive deeper into Dune's capabilities:

- The `dune-project` file specifies metadata about the project, including its name, packaging data (including dependencies), and information about the authors and maintainers. Open this in your editor to fill in the placeholder values. See *dune-project* for details.
- The `test` directory contains a skeleton for your project's tests. Add to the tests by editing `test/test_project_name.ml`. See *Writing and Running Tests* for details on testing.
- The `lib` directory will hold the library you write to provide your executable's core functionality. Add modules to your library by creating new `.ml` files in this directory. See *library* for details on specifying libraries manually.
- The `bin` directory holds a skeleton for the executable program. Within the modules in this directory, you can access the modules in your `lib` under the namespace `project_name.Mod`, where `project_name` is replaced with the name of your project and `Mod` corresponds to the name of the file in the `lib` directory. You can run the executable with `dune exec project_name`. See *Building a Hello World Program From Scratch* for an example of specifying an executable manually and *executable* for details.
- The `project_name.opam` file will be freshly generated from the `dune-project` file whenever you build your project. You shouldn't need to worry about this, but you can see *How Dune integrates with opam* for details.
- The `dune` files in each directory specify the component to be built with the files in that directory. For details on `dune` files, see *dune*.

Initializing a Library

To initialize a project for an OCaml library, run the following (replacing `project_name` with the name of your project):

```
$ dune init proj --kind=lib project_name
```

This creates a project directory that includes the following contents:

```
project_name/
├── dune-project
├── lib
│   └── dune
├── test
│   ├── dune
│   └── test_project_name.ml
└── project_name.opam
```

Now, enter your project's directory:

```
$ cd project_name
```

Then, you can build your project with:

```
$ dune build
```

You can run your tests with:

```
$ dune test
```

All of the subcomponents generated are the same as those described in *Initializing an Executable*, with the following exceptions:

- There is no `bin` directory generated.
- The `dune` file in the `lib` directory specifies that the library should be *public*. See *library* for details.

1.2.2 Building a Hello World Program From Scratch

Create a new directory within a Dune project (*Initializing an Executable*). Since OCaml is a compiled language, first create a `dune` file in Nano, Vim, or your preferred text editor. Declare the `hello_world` executable by including the following stanza (shown below). Name this initial file `dune` and save it.

```
(executable
 (name hello_world))
```

Create a second file containing the following code and name it `hello_world.ml` (including the .ml extension). It will implement the executable stanza in the `dune` file when built.

```
print_endline "Hello, world!"
```

Next, build your new program in a shell using this command:

```
$ dune build hello_world.exe
```

This will create a directory called `_build` and build the program: `_build/default/hello_world.exe`. Note that native code executables will have the `.exe` extension on all platforms (including non-Windows systems).

Finally, run it with the following command to see that it worked. In fact, the executable can both be built and run in a single step:

```
$ dune exec -- ./hello_world.exe
```

Voila! This should print "Hello, world!" in the command line.

1.2.3 Building a Hello World Program Using Lwt

Lwt is a concurrent library in OCaml.

In a directory of your choice, write this `dune` file:

```
(executable
 (name hello_world)
 (libraries lwt.unix))
```

This `hello_world.ml` file:

```
Lwt_main.run (Lwt_io.printf "Hello, world!\n")
```

And build it with:

```
$ dune build hello_world.exe
```

The executable will be built as `_build/default/hello_world.exe`

1.2.4 Building a Hello World Program Using Core and Jane Street PPXs

Write this `dune` file:

```
(executable
 (name hello_world)
 (libraries core)
 (preprocess (pps ppx_jane)))
```

This `hello_world.ml` file:

```
open Core

let () =
  Sexp.to_string_hum [%sexp ([3;4;5] : int list)]
  |> print_endline
```

And build it with:

```
$ dune build hello_world.exe
```

The executable will be built as `_build/default/hello_world.exe`

1.2.5 Defining a Library Using Lwt and `ocaml-re`

Write this `dune` file:

```
(library
 (name        mylib)
 (public_name mylib)
 (libraries re lwt))
```

The library will be composed of all the modules in the same directory. Outside of the library, module `Foo` will be accessible as `Mylib.Foo`, unless you write an explicit `mylib.ml` file.

You can then use this library in any other directory by adding `mylib` to the `(libraries ...)` field.

1.2.6 Building a Hello World Program in Bytecode

In a directory of your choice, write this `dune` file:

```
;; This declares the hello_world executable implemented by hello_world.ml
;; to be build as native (.exe) or bytecode (.bc) version.
(executable
 (name hello_world)
 (modes byte exe))
```

This `hello_world.ml` file:

```
print_endline "Hello, world!"
```

And build it with:

```
$ dune build hello_world.bc
```

The executable will be built as `_build/default/hello_world.bc`. The executable can be built and run in a single step with `dune exec ./hello_world.bc`. This bytecode version allows the usage of `ocamldebug`.

1.2.7 Setting the OCaml Compilation Flags Globally

Write this `dune` file at the root of your project:

```
(env
 (dev
  (flags (:standard -w +42)))
 (release
  (ocamlopt_flags (:standard -O3))))
```

dev and *release* correspond to build profiles. The build profile can be selected from the command line with `--profile foo` or from a *dune-workspace* file by writing:

```
(profile foo)
```

1.2.8 Using Cppo

Add this field to your `library` or `executable` stanzas:

```
(preprocess (action (run %{bin:cppo} -V OCAML:%{ocaml_version} %{input-file})))
```

Additionally, if you want to include a `config.h` file, you need to declare the dependency to this file via:

```
(preprocessor_deps config.h)
```

Using the `.cppo.ml` Style Like the `ocamlbuild` Plugin

Write this in your dune file:

```
(rule
 (targets foo.ml)
 (deps    (:first-dep foo.cppo.ml) <other files that foo.ml includes>)
 (action  (run %{bin:cppo} %{first-dep} -o %{targets})))
```

1.2.9 Defining a Library with C Stubs

Assuming you have a file called `mystubs.c`, that you need to pass `-I/blah/include` to compile it and `-lblah` at link time, write this dune file:

```
(library
 (name            mylib)
 (public_name     mylib)
 (libraries       re lwt)
 (foreign_stubs
  (language c)
  (names mystubs)
  (flags -I/blah/include))
 (c_library_flags (-lblah)))
```

1.2.10 Defining a Library with C Stubs using `pkg-config`

Same context as before, but using `pkg-config` to query the compilation and link flags. Write this dune file:

```
(library
 (name            mylib)
 (public_name     mylib)
 (libraries       re lwt)
 (foreign_stubs
  (language c)
  (names mystubs)
  (flags (:include c_flags.sexp)))
 (c_library_flags (:include c_library_flags.sexp)))

(rule
 (targets c_flags.sexp c_library_flags.sexp)
 (action  (run ./config/discover.exe)))
```

Then create a `config` subdirectory and write this `dune` file:

```
(executable
 (name discover)
 (libraries dune-configurator))
```

as well as this `discover.ml` file:

```
module C = Configurator.V1

let () =
C.main ~name:"foo" (fun c ->
let default : C.Pkg_config.package_conf =
  { libs   = ["-lgst-editing-services-1.0"]
  ; cflags = []
  }
in
let conf =
  match C.Pkg_config.get c with
  | None -> default
  | Some pc ->
     match (C.Pkg_config.query pc ~package:"gst-editing-services-1.0") with
     | None -> default
     | Some deps -> deps
in


C.Flags.write_sexp "c_flags.sexp"         conf.cflags;
C.Flags.write_sexp "c_library_flags.sexp" conf.libs)
```

1.2.11 Using a Custom Code Generator

To generate a file `foo.ml` using a program from another directory:

```
(rule
 (targets foo.ml)
 (deps    (:gen ../generator/gen.exe))
 (action  (run %{gen} -o %{targets})))
```

1.2.12 Defining Tests

Write this in your `dune` file:

```
(test (name my_test_program))
```

And run the tests with:

```
$ dune runtest
```

It will run the test program (the main module is `my_test_program.ml`) and error if it exits with a nonzero code.

In addition, if a `my_test_program.expected` file exists, it will be compared to the standard output of the test program and the differences will be displayed. It is possible to replace the `.expected` file with the last output using:

```
$ dune promote
```

1.2.13 Building a Custom Toplevel

A toplevel is simply an executable calling `Topmain.main ()` and linked with the compiler libraries and `-linkall`. Moreover, currently toplevels can only be built in bytecode.

As a result, write this in your dune file:

```
(executable
 (name       mytoplevel)
 (libraries  compiler-libs.toplevel mylib)
 (link_flags (-linkall))
 (modes      byte))
```

And write this in `mytoplevel.ml`:

```
let () = exit (Topmain.main ())
```

1.3 Command-Line Interface

This section describes using dune from the shell.

1.3.1 Initializing Components

NOTE: The `dune init` command is still under development and subject to change.

Dune's `init` subcommand provides limited support for generating Dune file stanzas and folder structures to define components. The `dune init` command can be used to quickly add new projects, libraries, tests, and executables without having to manually create Dune files in a text editor, or it can be composed to programmatically generate parts of a multi-component project.

Initializing a Project

You can run the following command to initialize a new Dune project that uses the `base` and `cmdliner` libraries and supports inline tests:

```
$ dune init proj myproj --libs base,cmdliner --inline-tests --ppx ppx_inline_test
```

This creates a new directory called `myproj`, including subdirectories and dune files for library, executable, and test components. Each component's dune file will also include the declarations required for the given dependencies.

This is the quickest way to get a basic dune project up and building.

Initializing an Executable

To add a new executable to a dune file in the current directory (creating the file if necessary), run

```
$ dune init exe myexe --libs base,containers,notty --ppx ppx_deriving
```

This will add the following stanza to the dune file:

```
(executable
 (name main)
 (libraries base containers notty)
 (preprocess
  (pps ppx_deriving)))
```

Initializing a Library

Run the following command to create a new directory src, initialized as a library:

```
$ dune init lib mylib src --libs core --inline-tests --public
```

This will ensure the file ./src/dune contains the below stanza (creating the file and directory, if necessary):

```
(library
 (public_name mylib)
 (inline_tests)
 (name mylib)
 (libraries core)
 (preprocess
  (pps ppx_inline_tests)))
```

Initializing Components in a Specified Directory

All init subcommands take an optional PATH argument, which should be a path to a directory. When supplied, the component will be created in the specified directory. E.g., to initialize a project in the current working directory, run

```
$ dune init proj my_proj .
```

To initialize a project in a directory in some nested path, run

```
$ dune init proj my_proj path/to/my/project
```

If the specified directory does not already exist, it will be created.

Learning More About the `init` Commands

Consult the manual page using the `dune init --help` command for more details.

1.3.2 Finding the Root

The root of the current workspace is determined by looking up a `dune-workspace` or `dune-project` file in the current directory and its parent directories. Dune requires at least one of these two files to operate.

If it isn't in the current directory, Dune prints out the root when starting:

```
$ dune runtest
Entering directory '/home/jdimino/code/dune'
...
```

This message can be suppressed with the `--no-print-directory` command line option (as in GNU make).

More precisely, Dune will choose the outermost ancestor directory containing a `dune-workspace` file, which is used to mark the root of the current workspace. If no `dune-workspace` file is present, the same strategy applies with `dune-project` files.

In case of a mix of *dune-workspace* and *dune-project* files, workspace files take precedence over project files in the sense that if a `dune-workspace` file is found, only parent `dune-workspace` files will be considered when looking for the root; however, if a *dune-project* file is found both parent `dune-workspace` and `dune-project` files will be considered.

A `dune-workspace` file is also a configuration file. Dune will read it unless the `--workspace` command line option is used. See *dune-workspace* for the syntax of this file. The scope of `dune-project` files is wider than the scope `dune-workspace` files. For instance, a `dune-project` file may specify the name of the project which is a universal property of the project, while a `dune-workspace` file may specify an opam switch name which is valid only on a given machine. For this reason, it is common and recommended to commit `dune-project` files in repositories, while it is less common to commit `dune-workspace` files.

Current Directory

If the previous rule doesn't apply, i.e., no ancestor directory has a file named `dune-workspace`, then the current directory will be used as root.

Forcing the Root (for Scripts)

You can pass the `--root` option to `dune` to select the root explicitly. This option is intended for scripts to disable the automatic lookup.

Note that when using the `--root` option, targets given on the command line will be interpreted relative to the given root, not relative to the current directory, as this is normally the case.

1.3.3 Interpretation of Targets

This section describes how Dune interprets the targets provided on the command line. When no targets are specified, Dune builds the *default alias*.

Resolution

All targets that Dune knows how to build live in the `_build` directory. Although, some are sometimes copied to the source tree for the need of external tools. These includes `<package>.install` files when either `-p` or `--promote-install-files` is passed on the command line.

As a result, if you want to ask Dune to produce a particular `.exe` file you would have to type:

```
$ dune build _build/default/bin/prog.exe
```

However, for convenience, when a target on the command line doesn't start with `_build`, Dune expands it to the corresponding target in all the build contexts that Dune knows how to build. When using `--verbose`, it prints out the actual set of targets upon starting:

```
$ dune build bin/prog.exe --verbose
...
Actual targets:
- _build/default/bin/prog.exe
- _build/4.03.0/bin/prog.exe
- _build/4.04.0/bin/prog.exe
```

If a target starts with the `@` sign, it is interpreted as an *alias*. See *Aliases*.

Variables for Artifacts

It's possible to build specific artifacts by using the corresponding variable on the command line. For example:

```
dune build '%{cmi:foo}'
```

See *Variables for Artifacts* for more information.

1.3.4 Finding External Libraries

When a library isn't available in the workspace, Dune will search for it in the installed world and expect it to be already compiled.

It looks up external libraries using a specific list of search paths, and each build context has a specific list of search paths.

When running inside an opam environment, Dune will look for installed libraries in `$OPAM_SWITCH_PREFIX/lib`. This includes both opam build context configured via the `dune-workspace` file and the default build context when the variable `$OPAM_SWITCH_PREFIX` is set.

Otherwise, Dune takes the directory where `ocamlc` was found and appends *../lib* ` to it. For instance, if `ocamlc` is found in `/usr/bin`, Dune looks for installed libraries in `/usr/lib`.

In addition to the two above rules, Dune always inspects the `OCAMLPATH` environment variable and uses the paths defined in this variable. `OCAMLPATH` always has precedence and can have different values in different build contexts. For instance, you can set it manually in a specific build context via the `dune-workspace` file.

1.3.5 Running Tests

There are two ways to run tests:

- `dune build @runtest`
- `dune test` (or the more explicit `dune runtest`)

The two commands are equivalent, and they will run all the tests defined in the current directory and its children directories recursively. You can also run the tests in a specific sub-directory and its children by using:

- `dune build @foo/bar/runtest`
- `dune test foo/bar` (or `dune runtest foo/bar`)

1.3.6 Watch Mode

The `dune build` and `dune runtest` commands support a `-w` (or `--watch`) flag. When it's passed, Dune will perform the action as usual and then wait for file changes and rebuild (or rerun the tests). This feature requires `inotifywait` or `fswatch` to be installed.

1.3.7 Launching the Toplevel (REPL)

Dune supports launching a utop instance with locally defined libraries loaded.

```
$ dune utop <dir> -- <args>
```

Where `<dir>` is a directory under which Dune searches (recursively) for all libraries that will be loaded. `<args>` will be passed as arguments to the `utop` command itself. For example, `dune utop lib -- -implicit-bindings` will start `utop`, with the libraries defined in `lib` and implicit bindings for toplevel expressions.

Dune also supports loading individual modules unsealed by their signatures into the toplevel. This is accomplished by launching a toplevel and then asking dune to return the toplevel directives needed to evaluate the module:

```
$ utop
# use_output "dune ocaml top-module path/to/module.ml";;
```

Requirements & Limitations

- Utop version >= 2.0 is required for this to work.
- This subcommand only supports loading libraries. Executables aren't supported.
- Libraries that are dependencies of utop itself cannot be loaded. For example Camomile.
- Loading libraries that are defined in different directories into one `utop` instance isn't possible.

1.3.8 Restricting the Set of Packages

Restrict the set of packages from your workspace that Dune can see with the `--only-packages` option:

```
$ dune build --only-packages pkg1,pkg2,... @install
```

This option acts as if you went through all the Dune files and commented out the stanzas referring to a package that isn't in the list given to `dune`.

1.3.9 Distributing Projects

Dune provides support for building and installing your project; however, it doesn't provide helpers for distributing it. It's recommended to use dune-release for this purpose.

The common defaults are that your projects include the following files:

- `README.md`
- `CHANGES.md`
- `LICENSE.md`

If your project contains several packages, all the package names must be prefixed by the shortest one.

1.3.10 `dune subst`

One of the features `dune-release` provides is watermarking; it replaces various strings of the form `%%ID%%` in all your project files before creating a release tarball or when the opam user pins the package.

This is especially interesting for the `VERSION` watermark, which gets replaced by the version obtained from the Version-Control System (VCS). For instance, if you're using Git, `dune-release` invokes this command to find out the version:

```
$ git describe --always --dirty --abbrev=7
1.0+beta9-79-g29e9b37
```

If no VCS is detected, `dune subst` will do nothing.

Projects using Dune usually only need `dune-release` for creating and publishing releases. However, they may still substitute the watermarks when the user pins the package. To help with this, Dune provides the `subst` sub-command.

`dune subst` performs the same substitution that `dune-release` does with the default configuration, i.e., calling `dune subst` at the root of your project will rewrite all your project files.

More precisely, it replaces the following watermarks in the source files:

- `NAME`, the name of the project
- `VERSION`, output of `git describe --always --dirty --abbrev=7`
- `VERSION_NUM`, same as `VERSION` but with a potential leading `v` or `V` dropped
- `VCS_COMMIT_ID`, commit hash from the vcs
- `PKG_MAINTAINER`, contents of the `maintainer` field from the opam file
- `PKG_AUTHORS`, contents of the `authors` field from the opam file
- `PKG_HOMEPAGE`, contents of the `homepage` field from the opam file
- `PKG_ISSUES`, contents of the `issues` field from the opam file
- `PKG_DOC`, contents of the `doc` field from the opam file

- PKG_LICENSE, contents of the `license` field from the opam file
- PKG_REPO, contents of the `repo` field from the opam file

The project name is obtained by reading the `dune-project` file in the directory where `dune subst` is called. The `dune-project` file must exist and contain a valid `(name ...)` field.

Note that `dune subst` is meant to be called from the opam file and behaves a bit different to other Dune commands. In particular it doesn't try to detect the root of the workspace and must be called from the root of the project.

1.3.11 Custom Build Directory

By default Dune places all build artifacts in the `_build` directory relative to the user's workspace. However, one can customize this directory by using the `--build-dir` flag or the `DUNE_BUILD_DIR` environment variable.

```
$ dune build --build-dir _build-foo

# this is equivalent to:
$ DUNE_BUILD_DIR=_build-foo dune build

# Absolute paths are also allowed
$ dune build --build-dir /tmp/build foo.exe
```

1.3.12 Installing a Package

Via opam

When releasing a package using Dune in opam, there's nothing special to do. Dune generates a file called `<package-name>.install` at the root of the project. This contains a list of files to install, and opam reads it in order to perform the installation.

Manually

When not using opam, or when you want to manually install a package, you can ask Dune to perform the installation via the `install` command:

```
$ dune install [PACKAGE]...
```

This command takes a list of package names to install. If no packages are specified, Dune will install all available packages in the workspace. When several build contexts are specified via a *dune-workspace* file, Dune performs the installation in all the build contexts.

Destination Directory

For a given build context, the installation directories are determined with a single scheme for all installation sections. Taking the `lib` installation section as an example, the priorities of this scheme are as follows:

1. if an explicit `--lib <path>` argument is passed, use this path
2. if an explicit `--prefix <path>` argument is passed, use `<path>/lib`
3. if `--lib <path>` argument is passed before during dune compilation to `./configure`, use this paths
4. if `OPAM_SWITCH_PREFIX` is present in the environment use `$OPAM_SWITCH_PREFIX/lib`

5. otherwise, fail

Relocation Mode

The installation can be done in specific mode (`--relocation`) for creating a directory that can be moved. In that case, the installed executables will look up the package sites (cf *How to Load Additional Files at Runtime*) relative to its location. The *–prefix* directory should be used to specify the destination.

If you're using plugins that depend on installed libraries and aren't executable dependencies, like libraries that need to be loaded at runtime, you must copy the libraries manually to the destination directory.

1.3.13 Querying Merlin Configuration

Since Version 2.8, Dune no longer promotes `.merlin` files to the source directories. Instead, Dune stores these configurations in the *_build* folder, and Merlin communicates directly with Dune to obtain its configuration via the *ocaml-merlin* subcommand. The Merlin configuration is now stanza-specific, allowing finer control. The following commands aren't needed for normal Dune and Merlin use, but they can provide insightful information when debugging or configuring non-standard projects.

Printing the Configuration

It's possible to manually query the generated configuration for debugging purposes:

```
$ dune ocaml merlin dump-config
```

This command prints the distinct configuration of each module present in the current directory. This directory must be in a Dune workspace and the project must be already built. The configuration will be encoded as s-expressions, which are used to communicate with Merlin.

Printing an Approximated `.merlin`

It's also possible to print the current folder's configuration in the Merlin configuration syntax by running the following command:

```
$ dune ocaml dump-dot-merlin > .merlin
```

In that case, Dune prints only one configuration: the result of the configuration's coarse merge in the current folder's various modules. This folder must be in a Dune workspace, and the project must be already built. Preprocessing directives and other flags will be commented out and must be un-commented afterward. This feature doesn't aim at writing exact or correct `.merlin` files. Its sole purpose is to lessen the burden of writing the configuration from scratch.

Non-Standard Filenames

Merlin configuration loading is based on filenames, so if you have files that are preprocessed by custom rules before they are built, they should respect the following naming convention: the unprocessed file should start with the name of the resulting processed file followed by a dot. The rest does not matter. Dune uses only the name before the first dot to match with available configurations.

For example, if you use the `cppo` preprocessor to generate the file `real_module_name.ml`, then the source file could be named `real_module_name.cppo.ml`.

1.3.14 Running a Coq Toplevel

See *Running a Coq Toplevel*.

CHAPTER

TWO

HOW-TO GUIDES

These guides will help you use Dune's features in your project.

2.1 How to Install Dune

Dune is available as an Opam package. First, make sure that Opam is installed:

```
$ opam --version
2.1.5
```

Any version higher than 2.0.0 is supported, though preferably at least 2.1.0.

If Opam is not available, follow the official instructions on the Opam website to install it and then run its global setup with `opam init`.

Note: Opam requires a "shell hook" to work properly. Make sure to set it up correctly during `opam init`. Otherwise you will have to run `eval $(opam env)` every time you create an Opam switch or change directory.

Then, you can install Dune in an Opam switch using the following command:

```
$ opam install dune
```

After the command completes, the following should display a version number:

```
$ dune --version
3.12.1
```

Note: In most cases, when using Opam you will not need to install Dune by hand. Installing the project's dependencies will install it in the Opam switch.

2.2 How to Set up Automatic Formatting

This guide will show you how to configure Dune so that it can check the formatting of your source code.

Formatting is defined per project. This ensures that if a project is reused elsewhere, its formatting configuration will not interfere.

2.2.1 Setting Up the Environment

First, let's open the `dune-project` file. Make sure that the version specified in `(lang dune X.Y)` is at least `2.0`. Most formatting configuration happens in that file. If you want to format OCaml sources and `dune` files, you don't have anything to add. Otherwise, refer to the *formatting* stanza.

Next we need to install some code formatting tools. For OCaml code, this means installing OCamlFormat with `opam install ocamlformat`. Formatting `dune` files is built into Dune and does not require any extra tools. For Reason code, this uses the `refmt` tool which is already installed if you are using Reason syntax in your project. If your project uses a *dialect*, a specific tool might be required.

Using OCamlFormat requires some configuration. Take note of the version returned by `ocamlformat --version` (let's name that `X.Y.Z`) and create an `.ocamlformat` file in the same directory as `dune-project` with the following contents:

```
version=X.Y.Z
profile=default
```

The `version` line is checked by OCamlFormat and ensures that everybody contributing to the project uses the same version.

Note that you do not have to add `ocamlformat` to your opam files.

2.2.2 Running the Formatters

Run the `dune build @fmt` command. It will format the source files in the corresponding project and display the differences:

```
$ dune build @fmt
--- hello.ml
+++ hello.ml.formatted
@@ -1,3 +1 @@
-let () =
-  print_endline
-    "hello, world"
+let () = print_endline "hello, world"
```

Then it's possible to accept the correction by calling `dune promote` to replace the source files with the corrected versions.

```
$ dune promote
Promoting _build/default/hello.ml.formatted to hello.ml.
```

As usual with promotion, it's possible to combine these two steps by running `dune build @fmt --auto-promote`. This command can also be shortened to `dune fmt`. See *Diffing and Promotion* for more details.

2.2.3 Setting Up Your CI

To check formatting in CI, the precise set up depends on the CI system used, but in general it is easier to set up a dedicated job that just installs `dune` and the formatting tools, rather than doing that as part of the jobs that run tests.

If you use ocaml-ci, you have nothing to do: a formatting job is set up automatically.

If you use setup-ocaml, you can use the *lint-fmt* extend listed in the README file.

2.3 How to Generate Opam Files from `dune-project`

This guide will show you how to configure Dune so that it generates opam files.

2.3.1 Declaring Package Dependencies

The goal of this first step is to add `(package)` stanzas in your `dune-project` file. These stanzas declare the metadata that your package uses in the language of opam packages. See *Declaring a Package*.

The next step depends on whether you are starting from a clean slate (new package) or adapting an existing opam file.

For a New Package (No Existing Opam File)

If your project does not have any opam files, you will have to find your package dependencies. In the simple case, collect all the libraries that appear in the `(libraries)` fields of your project and put this list in the `(depends)` field of the corresponding `(package)`. See *The OCaml Ecosystem* for the difference between libraries and packages.

Example: you have a library that looks like:

```
(library
 (public_name frobnitz)
 (libraries lwt fmt))
```

You can declare the package as:

```
(package
 (name frobnitz)
 (depends lwt fmt))
```

Also add common metadata using `(authors)`, `(maintainers)`, `(license)`, `(source)`, as well as a `(synopsis)` and a `(description)` for

For an Existing Package

If you already have an opam file (or several of them), you can convert it by following the rules in *package*.

For example, if your opam file looks like:

```
opam-version: 2.0
authors: ["Anil Madhavapeddy" "Rudi Grinberg"]
maintainer: ["team@mirage.org"]
name: "cohttp-async"
synopsis: "HTTP client and server for the Async library"
```

(continues on next page)

(continued from previous page)

```
description: "A _really_ long description"
license: "ISC"
bug-reports: "https://github.com/mirage/ocaml-cohttp/issues"
homepage: "https://github.com/mirage/ocaml-cohttp/"
dev-repo: "git+https://github.com/mirage/ocaml-cohttp.git"
build: [
  ["dune" "subst"] {dev}
  [
    "dune"
    "build"
    "-p"
    name
    "-j"
    jobs
    "@install"
    "@runtest" {with-test}
    "@doc" {with-doc}
  ]
]
depends: [
  "dune" { >= "3.4" }
  "odoc" { with-doc }
  "cohttp" { >= "1.0.2" }
  "conduit-async" { >= "1.0.3" }
  "async" { >= "v0.10.0" }
]
```

You can express this as:

```
(source (github mirage/ocaml-cohttp))
(license ISC)
(authors "Anil Madhavapeddy" "Rudi Grinberg")
(maintainers "team@mirage.org")

(package
 (name cohttp-async)
 (synopsis "HTTP client and server for the Async library")
 (description "A _really_ long description")
 (depends
  (cohttp (>= 1.0.2))
  (conduit-async (>= 1.0.3))
  (async (>= v0.10.0))))
```

General Notes and Tips

- Do not declare a dependency on the `dune` and `odoc` packages. Dune will generate them with the right constraints.
- For fields that are common between packages (like `(authors)` or `(license)`), you can use a global one rather than replicate it between packages.
- If you use a platform such as GitHub you can use `(source)` as a shorthand instead of specifying `(bug_reports)`, `(homepage)`, etc.
- `(package)` stanzas do not support all opam fields or complete syntax for dependency specifications. If the package you are adapting requires this, keep the corresponding opam fields in a `pkg.opam.template` file. See *Packages*.
- It is not necessary to specify `(version)`, this will be added at release time if you use dune-release.

2.3.2 Generating Opam Files

If you have existing `*.opam` files, make a backup of them because the instructions in this section will overwrite them.

Now that you have declared package metadata in `dune-project`, you can add `(generate_opam_files)` in `(dune-project)`.

From now on, commands like `dune build` and `dune runtest` are going to regenerate the contents of opam files from the metadata in `(package)` stanzas. If you only want to generate the opam file, run `dune build <project_name>.opam`.

Run `dune build` once and observe that the opam files have been created or updated. Make sure to add these changes to your version control system.

2.4 Cross-Compilation

Dune allows for cross-compilation by defining build contexts with multiple targets. Targets are specified by adding a `targets` field to the build context definition.

`targets` takes a list of target name. It can be either:

- `native`, the native tools that can build binaries to run on the machine doing the build
- the name of an alternative toolchain

Note that at the moment, there is no official support for cross-compilation in OCaml. Dune supports the *opam-cross-<x>* repositories from the OCaml-cross organization on GitHub, such as:

- opam-cross-windows
- opam-cross-android
- opam-cross-ios

In particular:

- to build Windows binaries using opam-cross-windows, write `windows` in the list of targets
- to build Android binaries using opam-cross-android, write `android` in the list of targets
- to build IOS binaries using opam-cross-ios, write `ios` in the list of targets

For example, the following workspace file defines three different targets for the `default` build context:

```
(context (default (targets native windows android)))
```

This configuration defines three build contexts:

- `default`
- `default.windows`
- `default.android`

Note that the `native` target is always implicitly added when not present; however, `dune build @install` will skip this context, i.e., `default` will only be used for building executables needed by the other contexts.

With such a setup, calling `dune build @install` will build all the packages three times.

Note that instead of writing a `dune-workspace` file, you can also use the `-x` command line option. Passing `-x foo` to `dune` without having a `dune-workspace` file is the same as writing the following `dune-workspace` file:

```
(context (default (targets foo)))
```

If you have a `dune-workspace` and pass a `-x foo` option, `foo` will be added as target of all context stanzas.

2.4.1 How Does it Work?

In such a setup, binaries that need to be built and executed in the `default.windows` or `default.android` contexts as part of the build will no longer be executed. Instead, all the binaries that will be executed come from the `default` context. One consequence of this is that all preprocessing (PPX or otherwise) will be done using binaries built in the `default` context.

To clarify this with an example, let's assume that you have the following `src/dune` file:

```
(executable (name foo))
(rule (with-stdout-to blah (run ./foo.exe)))
```

When building `_build/default/src/blah`, dune will resolve `./foo.exe` to `_build/default/src/foo.exe` as expected. However, for `_build/default.windows/src/blah` dune will resolve `./foo.exe` to `_build/default/src/foo.exe`

Assuming that the right packages are installed or that your workspace has no external dependencies, Dune will be able to cross-compile a given package without doing anything special.

Some packages might still have to be updated to support cross-compilation. For instance if the `foo.exe` program in the previous example was using `Sys.os_type`, it should instead take it as a command line argument:

```
(rule (with-stdout-to blah (run ./foo.exe -os-type %{os_type})))
```

2.5 Dealing with Foreign Libraries

The OCaml programming language can interface with libraries written in foreign languages such as C. This section explains how to do this with Dune. Note that it does not cover how to write the C stubs themselves, but this is covered by the OCaml manual.

More precisely, this section covers:

- How to add C/C++ stubs to an OCaml library
- How to pass specific compilation flags for compiling the stubs

- How to build a library with a foreign build system

In general, Dune has limited support for building source files written in foreign languages. This support is suitable for most OCaml projects containing C stubs, but it is too limited for building complex libraries written in C or other languages. For such cases, Dune can integrate a foreign build system into a normal Dune build.

2.5.1 Adding C/C++ Stubs to an OCaml Library

To add C stubs to an OCaml library, simply list the C files without the `.c` extension in the *Foreign Stubs* field. For instance:

```
(library
 (name mylib)
 (foreign_stubs (language c) (names file1 file2)))
```

You can also add C++ stubs to an OCaml library by specifying `(language cxx)` instead.

Dune is currently not flexible regarding the extension of the C/C++ source files. They have to be `.c` for C files and `.cpp`, `.cc` or `.cxx` for C++ files. If you have source files with other extensions and you want to build them with Dune, you need to rename them first. Alternatively, you can use the *foreign build sandboxing* method described below.

Header Files

C/C++ source files may include header files in the same directory as the C/C++ source files or in the same directory group when using *include_subdirs*.

The header files must have the `.h` extension.

Installing Header Files

It is sometimes desirable to install header files with the library. For that you have two choices: install them explicitly with an *install* stanza or use the `install_c_headers` field of the *library* stanza. This field takes a list of header files names without the `.h` extension. When a library installs header files, they are made visible to users of the library via the include search path.

2.5.2 Stub Generation with Dune Ctypes

Beginning in Dune 3.0, it's possible to use the ctypes field to generate bindings for C libraries without writing any C code.

Note that Dune support for this feature is experimental and is not subject to backward compatibility guarantees.

To use Dune ctypes stub generation, you must provide two OCaml modules: a "type description" module for describing the C library types and constants, and a "function description" module for describing the C library functions. Additionally, you must list any C headers and a method for resolving build and link flags.

If you're binding a library distributed by your OS, you can use the pkg-config utility to resolve any build and link flags. Alternatively, if you're using a locally installed library or a vendored library, you can provide the flags manually.

The "type description" module must define a functor named `Types` with signature `Ctypes.TYPE`. The "function description" module must define a functor named `Functions` with signature `Ctypes.FOREIGN`.

A Toy Example

To begin, you must declare the `ctypes` extension in your `dune-project` file:

```
(lang dune 3.17)
(using ctypes 0.3)
```

Next, here is a `dune` file you can use to define an OCaml program that binds a C system library called `libfoo`, which offers `foo.h` in a standard location.

```
(executable
 (name foo)
 (libraries core)
 ; ctypes backward compatibility shims warn sometimes; suppress them
 (flags (:standard -w -9-27))
 (ctypes
  (external_library_name libfoo)
  (build_flags_resolver pkg_config)
  (headers (include "foo.h"))
  (type_description
   (instance Types)
   (functor Type_description))
  (function_description
   (concurrency unlocked)
   (instance Functions)
   (functor Function_description))
  (generated_types Types_generated)
  (generated_entry_point C)))
```

This field will introduce a module named `C` into your project, with the sub-modules `Types` and `Functions` that will have your fully-bound C types, constants, and functions.

Given `libfoo` with the C header file `foo.h`:

```
#define FOO_VERSION 1

int foo_init(void);

int foo_fnubar(char *);

void foo_exit(void);
```

Your example `type_description.ml` file is:

```
open Ctypes

module Types (F : Ctypes.TYPE) = struct
  open F

  let foo_version = constant "FOO_VERSION" int
end
```

Your example `function_description.ml` file is:

```
open Ctypes

(* This Types_generated module is an instantiation of the Types
   functor defined in the type_description.ml file. It's generated by
   a C program that Dune creates and runs behind the scenes. *)
module Types = Types_generated

module Functions (F : Ctypes.FOREIGN) = struct
  open F

  let foo_init = foreign "foo_init" (void @-> returning int)

  let foo_fnubar = foreign "foo_fnubar" (string_opt @-> returning int)

  let foo_exit = foreign "foo_exit" (void @-> returning void)
end
```

Finally, the entry point of your executable named above, `foo.ml`, demonstrates how to access the bound C library functions and values:

```
let () =
  if (C.Types.foo_version <> 1) then
    failwith "foo only works with libfoo version 1";

  match C.Functions.foo_init () with
  | 0 ->
    C.Functions.foo_fnubar "fnubar!";
    C.Functions.foo_exit ()
  | err_code ->
    Printf.eprintf "foo_init failed: %d" err_code;
;;
```

From here, one only needs to run `dune build ./foo.exe` to generate the stubs and build and link the example `foo.exe` program.

Complete information about the `ctypes` combinators used above is available at the ctypes project.

Ctypes Field Reference

The `ctypes` field can be used in any `executable(s)` or `library` stanza.

```
((executable|library)
  ...
  (ctypes
    (external_library_name <package-name>)
    (type_description
      (instance <module-name>)
      (functor <module-name>))
    (function_description
      (instance <module-name>)
      (functor <module-name>)
      <optional-function-description-fields>)
    (generated_entry_point <module-name>)
```

(continues on next page)

(continued from previous page)

```
    <optional-ctypes-fields>)
 )
```

- `type_description`: the `functor` module is a description of the C library types and constants written in the `ctypes` domain-specific language you wish to bind. The `instance` module is the name of the instantiated functor, inserted into the top-level of the `generated_entry_point` module.
- `function_description`: the `functor` module is a description of the C library functions written in the `ctypes` domain-specific language you wish to bind. The `instance` module is the name of the instantiated functor, inserted into the top-level of the `generated_entry_point` module. The `function_description` field can be repeated. This is useful if you need to specify sets of functions with different concurrency policies (see below).

The instantiated types described above can be accessed from the function descriptions by referencing them as the module specified in optional `generated_types` field.

<optional-ctypes-fields> are:

- `(build_flags_resolver <pkg_config|vendored-field>)` tells Dune how to compile and link your foreign library. Specifying `pkg_config` will use the pkg-config tool to query the compilation and link flags for `external_library_name`. For vendored libraries, provide the build and link flags using `vendored` field. If `build_flags_resolver` is not specified, the default of `pkg_config` will be used.
- `(generated_types <module-name>)` is the name of an intermediate module. By default, it's named `Types_generated`. You can use this module to access the types defined in `Type_description` from your `Function_description` module(s).
- `(generated_entry_point <module-name>)` is the name of a generated module that your instantiated `Types` and `Functions` modules will instantiated under. We suggest calling it `C`.
- Headers can be added to the generated C files:
  - `(headers (include "include1" "include2" ...))` adds `#include <include1>`, `#include <include2>`. It uses the *Ordered Set Language*.
  - `(headers (preamble <preamble>)` adds directly the preamble. Variables can be used in `<preamble>` such as `%{read:  }`.
- Since the Dune's `ctypes` feature is still experimental, it could be useful to add additional dependencies in order to make sure that local headers or libraries are available: `(deps <deps-conf list>)`. See *Dependency Specification* for more details.

<optional-function-description-fields> are:

- `(concurrency <sequential|unlocked|lwt_jobs|lwt_preemptive>)` tells `ctypes stubgen` whether to call your C functions with the runtime lock held or released. These correspond to the `concurrency_policy` type in the `ctypes` library. If `concurrency` is not specified, the default of `sequential` will be used.
- `(errno_policy <ignore_errno|return_errno>)` specifies the errno_policy passed to the code generator. With `ignore_errno`, the errno variable is not accessed or returned by function calls. With `return_errno`, all functions will return the tuple `(retval, errno)`.

<vendored-field> is:

- `(vendored (c_flags <flags>) (c_library_flags <flags>))` provide the build and link flags for binding your vendored code. You must also provide instructions in your `dune` file on how to build the vendored foreign library; see the *foreign_library* stanza. Usually the `<flags>` should contain `:standard` in order to add the default flags used by the OCaml compiler for C files *use_standard_c_and_cxx_flags*.

2.5.3 Foreign Build Sandboxing

When the build of a C library is too complicated to express in the Dune language, it's possible to simply *sandbox* a foreign build. Note that this method can be used to build other things, not just C libraries.

To do that, follow the following procedure:

- Put all the foreign code in a sub-directory
- Tell Dune not to interpret configuration files in this directory via an *data_only_dirs* stanza
- Write a custom rule that:
  - depends on this directory recursively via *source_tree*
  - invokes the external build system
  - copies the generated files
  - the C archive `.a` must be built with `-fpic`
  - the `libfoo.so` must be copied as `dllfoo.so`, and no `libfoo.so` should appear, otherwise the dynamic linking of the C library will be attempted. However, this usually fails because the `libfoo.so` isn't available at the time of the execution.
- *Attach* the C archive files to an OCaml library via *Foreign Archives*.

For instance, let's assume that you want to build a C library `libfoo` using `libfoo`'s own build system and attach it to an OCaml library called `foo`.

The first step is to put the sources of `libfoo` in your project, for instance in `src/libfoo`. Then tell Dune to consider `src/libfoo` as raw data by writing the following in `src/dune`:

```
(data_only_dirs libfoo)
```

The next step is to setup the rule to build `libfoo`. For this, writing the following code `src/dune`:

```
(rule
 (deps (source_tree libfoo))
 (targets libfoo.a dllfoo.so)
 (action
 (no-infer
  (progn
   (chdir libfoo (run make))
   (copy libfoo/libfoo.a libfoo.a)
   (copy libfoo/libfoo.so dllfoo.so)))))
```

We copy the resulting archive files to the top directory where they can be declared as `targets`. The build is done in a *no-infer* action because `libfoo/libfoo.a` and `libfoo/libfoo.so` are dependencies produced by an external build system.

The last step is to attach these archives to an OCaml library as follows:

```
(library
 (name bar)
 (foreign_archives foo))
```

Then, whenever you use the `bar` library, you'll also be able to use C functions from `libfoo`.

Limitations

When using the sandboxing method, the following limitations apply:

- The build of the foreign code will be sequential
- The build of the foreign code won't be incremental

Both these points could be improved. If you're interested in helping make this happen, please let the Dune team know and someone will guide you.

Real Example

The re2 project uses this method to build the `re2` C library. You can look at the file `re2/src/re2_c/dune` in this project to see a full working example.

2.6 Generating Documentation

2.6.1 Prerequisites

Documentation in Dune is done courtesy of the odoc tool. Therefore, to generate documentation in Dune, you will need to install this tool. This should be done with opam:

```
$ opam install odoc
```

2.6.2 Writing Documentation

Documentation comments will be automatically extracted from your OCaml source files following the syntax described in the section `Text formatting` of the OCaml manual.

Additional documentation pages may be attached to a package using the *documentation* stanza.

2.6.3 Building Documentation

To generate documentation using the *@doc* alias, all that's required to is to build this alias:

```
$ dune build @doc
```

An index page containing links to all the opam packages in your project can be found in:

```
$ open _build/default/_doc/_html/index.html
```

Documentation for private libraries may also be built with *@doc-private*:

```
$ dune build @doc-private
```

But these libraries will not be in the main HTML listing above, since they don't belong to any particular package, but the generated HTML will still be found in `_build/default/_doc/_html/<library>`.

Documentation Stanza: Examples

The *documentation* stanza will attach all the `.mld` files in the current directory in a project with a single package.

```
(documentation)
```

This stanza will attach three `.mld` files to package `foo`. The `.mld` files should be named `foo.mld`, `bar.mld`, and `baz.mld`

```
(documentation
 (package foo)
  (mld_files foo bar baz))
```

This stanza will attach all `.mld` files to the inferred package, excluding `wip.mld`, in the current directory:

```
(documentation
 (mld_files :standard \ wip))
```

All `.mld` files attached to a package will be included in the generated `.install` file for that package. They'll be installed by opam.

Package Entry Page

The `index.mld` file (specified as `index` in `mld_files`) is treated specially by Dune. This will be the file used to generate the entry page for the package, linked from the main package listing.

To generate pleasant documentation, we recommend writing an `index.mld` file with at least short description of your package and possibly some examples.

If you do not write your own `index.mld` file, Dune will generate one with the entry modules for your package. But this generated file will not be installed.

2.6.4 Passing Options to odoc

```
(env
 (<profile>
  (odoc <optional-fields>)))
```

See *env* for more details on the `(env ...)` stanza. `<optional-fields>` are:

- `(warnings <mode>)` specifies how warnings should be handled. `<mode>` can be: `fatal` or `nonfatal`. The default value is `nonfatal`. This field is available since Dune 2.4.0 and requires odoc 1.5.0.

2.6.5 Local Documentation Search Using Sherlodoc

If Sherlodoc is installed, generated HTML documentation will include a search bar. It supports search by name, documentation and fuzzy type search.

In can be installed with:

```
$ opam install sherlodoc
```

2.7 How to Load Additional Files at Runtime

There are many ways for applications to load files at runtime and Dune provides a well-tested, key-in-hand portable system for doing so. The Dune model works by defining `sites` where files will be installed and looked up at runtime. At runtime, each site is associated to a list of directories which contain the files added in the site.

WARNING: This feature remains experimental and is subject to breaking changes without warning. It must be explicitly enabled in the `dune-project` file with `(using dune_site 0.1)`

2.7.1 Sites

Defining a Site

A site is defined in a package *package* in the `dune-project` file. It consists of a name and a *section* (e.g `lib`, `share`, `etc`) where the site will be installed as a sub-directory.

```
(lang dune 3.17)
(using dune_site 0.1)
(name mygui)

(package
 (name mygui)
 (sites (share themes)))
```

Adding Files to a Site

Here the package `mygui` defines a site named `themes` that will be located in the section `share`. This package can add files to this `site` using the *install stanza*:

```
(install
 (section (site (mygui themes)))
 (files
  (layout.css as default/layout.css)
  (ok.png  as default/ok.png)
  (ko.png  as default/ko.png)))
```

Another package `mygui_material_theme` can install files inside `mygui` directory for adding a new theme. Inside the scope of `mygui_material_theme` the `dune` file contains:

```
(install
 (section (site mygui themes))
 (files
  (layout.css as material/layout.css)
  (ok.png  as material/ok.png)
  (ko.png  as material/ko.png)))
```

The package `mygui` must be present in the workspace or installed.

> **Warning:** Two files should not be installed by different packages at the same destination.

Getting the Locations of a Site at Runtime

The executable `mygui` will be able to get the locations of the `themes` site using the *generate_sites_module stanza*.

```
(executable
 (name mygui)
 (modules mygui mysites)
 (libraries dune-site))

(generate_sites_module
 (module mysites)
 (sites mygui))
```

The generated module *mysites* depends on the library *dune-site* provided by Dune.

Then inside `mygui.ml` module the locations can be recovered and used:

```
(** Locations of the site for the themes *)
let themes_locations : string list = Mysites.Sites.themes

(** Merge the contents of the directories in [dirs] *)
let lookup_dirs dirs =
  List.filter Sys.file_exists dirs
  |> List.map (fun dir -> Array.to_list (Sys.readdir dir))
  |> List.concat

(** Get the available themes *)
let find_available_themes () = lookup_dirs themes_locations

(** [lookup_file name dirs] finds the first file called [name] in [dirs] *)
let lookup_file filename dirs =
  List.find_map
    (fun dir ->
      let filename' = Filename.concat dir filename in
      if Sys.file_exists filename' then Some filename' else None)
    dirs

(** [lookup_theme_file theme file] get the [file] of the [theme] *)
let lookup_theme_file file theme =
  lookup_file (Filename.concat theme file) themes_locations

let get_layout_css = lookup_theme_file "layout.css"
let get_ok_ico = lookup_theme_file "ok.png"
let get_ko_ico = lookup_theme_file "ko.png"
```

Tests

During tests, the files are copied into the sites through the dependency `(package mygui)` and `(package mygui_material_theme)` as for other files in install stanza.

Installation

Installation is done simply with `dune install`; however, if one wants to install this tool to make it relocatable, one can use `dune install --relocatable --prefix $dir`. The files will be copied to the directory `$dir` but the binary `$dir/bin/mygui` will find the site location relative to its location. So even if the directory `$dir` is moved, `themes_locations` will be correct.

For installation through opam, `dune install` must be invoked with the option `--create-install-files` which creates an install file `<pkg>.install` and copy the file that needs substitution to an intermediary directory. The `<pkg>.opam` file generated by Dune *generate_opam_files* does the right invocation.

Implementation Details

The main difficulty for sites is that their directories are found at different locations at different times:

- When the package is available locally, the location is inside `_build`
- When the package is installed, the location is inside the install prefix
- If a local package wants to install files to the site of another installed package the location is at the same time in `_build` and in the install prefix of the second package.

With the last example, we see that the location of a site is not always a single directory, but rather it can consist of a sequence of directories: `["dir1" ; "dir2"]`. So a lookup must first look into *dir1*, then into *dir2*.

2.7.2 Plugins and Dynamic Loading of Packages

Dune allows you to define and load plugins without having to deal with specific compilation, installation directories, dependencies, or the `Dynlink_` module.

To define a plugin:

- The package defining the plugin interface must define a *site* where the plugins must live. Traditionally, this is in `(lib plugins)`, but it's just a convention.
- Define a library that each plugin must use to register itself (or otherwise provide its functionality).
- Define the plugin in another package using the *plugin* stanza.
- Generate a module that may load all available plugins using the *generated_module* stanza.

Example

We demonstrate an example of the scheme above. The example consists of the following components:

Inside package *app*:

- An executable *app*, that we intend to extend with plugins
- A library *app.registration* which defines the plugin registration interface
- A generated module *Sites* which can load available plugins at runtime

- An executable *app* that will use the module *Sites* to load all the plugins

Inside package *Plugin1*, we declare a plugin using the *app.registration* api and the *plugin* stanza.

Directory structure

```
.
├── app.ml
├── dune
├── dune-project
├── plugin
│   ├── dune
│   ├── dune-project
│   └── plugin1_impl.ml
└── registration.ml
```

Main Executable (C)

- The `dune-project` file:

```
(lang dune 3.17)
(using dune_site 0.1)
(name app)

(package
  (name app)
  (sites (lib plugins)))
```

- The `dune` file:

```
(executable
  (public_name app)
  (modules sites app)
  (libraries app.register dune-site dune-site.plugins))

(library
  (public_name app.register)
  (name registration)
  (modules registration))

(generate_sites_module
(module sites)
(plugins (app plugins)))
```

The generated module *sites* depends here also on the library *dune-site.plugins* because the *plugins* optional field is requested.

If the executable being created is an OCaml toplevel, then the `libraries` stanza needs to also include the `dune-site.toplevel` library. This causes the loading to use the toplevel's normal loading mechanism rather than `Dynload.loadfile` (which is not allowed in toplevels).

- The module `registration.ml` of the library `app.registration`:

```
let todo : (unit -> unit) Queue.t = Queue.create ()
```

- The code of the executable `app.ml`:

```
(* load all the available plugins *)
let () = Sites.Plugins.Plugins.load_all ()

let () = print_endline "Main app starts..."
(* Execute the code registered by the plugins *)
let () = Queue.iter (fun f -> f ()) Registration.todo
```

The Plugin "plugin1"

- The `plugin/dune-project` file:

```
(lang dune 3.17)
(using dune_site 0.1)

(generate_opam_files true)

(package
  (name plugin1))
```

- The `plugin/dune` file:

```
(library
  (public_name plugin1.plugin1_impl)
  (name plugin1_impl)
  (modules plugin1_impl)
  (libraries app.register))

(plugin
  (name plugin1)
  (libraries plugin1.plugin1_impl)
  (site (app plugins)))
```

- The code of the plugin `plugin/plugin1_impl.ml`:

```
let () =
  print_endline "Registration of Plugin1";
  Queue.add (fun () -> print_endline "Plugin1 is doing something...") Registration.todo
```

Running the Example

```
$ dune build @install && dune exec ./app.exe
Registration of Plugin1
Main app starts...
Plugin1 is doing something...
```

2.8 Instrumentation

In this section, we'll explain how to define and use instrumentation backends (such as `bisect_ppx` or `landmarks`) so that you can enable and disable coverage via `dune-workspace` files or by passing a command-line flag or environment variable. In addition to providing an easy way to toggle instrumentation of your code, this setup avoids creating a hard dependency on the precise instrumentation backend in your project.

2.8.1 Specifying What to Instrument

When an instrumentation backend is activated, Dune will only instrument libraries and executables for which the user has requested instrumentation.

To request instrumentation, one must add the following field to a library or executable stanza:

```
(library
 (name ...)
 (instrumentation
  (backend <name> <args>)
  <optional-fields>))
```

The backend `<name>` can be passed into arguments using `<args>`.

This field can be repeated multiple times in order to support various backends. For instance:

```
(library
 (name foo)
 (modules foo)
 (instrumentation (backend bisect_ppx --bisect-silent yes))
 (instrumentation (backend landmarks)))
```

This will instruct Dune that when either the `bisect_ppx` or `landmarks` instrumentation is activated, the library should be instrumented with this backend.

By default, these fields are simply ignored; however, when the corresponding instrumentation backend is activated, Dune will implicitly add the relevant `ppx` rewriter to the list of `ppx` rewriters.

At the moment, it isn't possible to instrument code that's preprocessed via an action preprocessors. As these preprocessors are quite rare nowadays, there is no plan to add support for them in the future.

`<optional-fields>` are:

- `(deps <deps-conf list>)` specifies extra instrumentation dependencies, for instance, if it reads a generated file. The dependencies are only applied when the instrumentation is actually enabled. The specification of dependencies is described in *Dependency Specification*.

2.8.2 Enabling/Disabling Instrumentation

Activating an instrumentation backend can be done via the command line or the `dune-workspace` file.

Via the command line, it is done as follows:

```
$ dune build --instrument-with <names>
```

Here `<names>` is a comma-separated list of instrumentation backends. For example:

```
$ dune build --instrument-with bisect_ppx,landmarks
```

This will instruct Dune to activate the given backend globally, i.e., in all defined build contexts.

It's also possible to enable instrumentation backends via the `dune-workspace` file, either globally or for specific builds contexts.

To enable an instrumentation backend globally, type the following in your `dune-workspace` file:

```
(lang dune 3.17)
(instrument_with bisect_ppx)
```

or for each context individually:

```
(lang dune 3.17)
(context default)
(context (default (name coverage) (instrument_with bisect_ppx)))
(context (default (name profiling) (instrument_with landmarks)))
```

If both the global and local fields are present, the precedence is the same as the `profile` field: the per-context setting takes precedence over the command-line flag, which takes precedence over the global field.

2.8.3 Declaring an Instrumentation Backend

Instrumentation backends are libraries with the special field `(instrumentation.backend)`. This field instructs Dune that the library can be used as an instrumentation backend, and it also provides the parameters specific to this backend.

Currently, Dune will only support `ppx` instrumentation tools, and the instrumentation library must specify the `ppx` rewriters that instruments the code. This can be done as follows:

```
(library
 ...
 (instrumentation.backend
   (ppx <ppx-rewriter-name>)))
```

When such an instrumentation backend is activated, Dune will implicitly add the mentioned `ppx` rewriter to the list of `ppx` rewriters for libraries and executables that specify this instrumentation backend.

2.9 JavaScript Compilation With Js_of_ocaml

Js_of_ocaml is a compiler from OCaml to JavaScript. The compiler works by translating OCaml bytecode to JS files. The compiler can be installed with opam:

```
$ opam install js_of_ocaml-compiler
```

2.9.1 Compiling to JS

Dune has full support building Js_of_ocaml libraries and executables transparently. There's no need to customize or enable anything to compile OCaml libraries/executables to JS.

To build a JS executable, just define an executable as you would normally. Consider this example:

```
$ echo 'print_endline "hello from js"' > foo.ml
```

With the following dune file:

```
(executable (name foo) (modes js))
```

And then request the `.js` target:

```
$ dune build ./foo.bc.js
$ node _build/default/foo.bc.js
hello from js
```

Similar targets are created for libraries, but we recommend sticking to the executable targets.

If you're using the Js_of_ocaml syntax extension, you must remember to add the appropriate PPX in the `preprocess` field:

```
(executable
 (name foo)
 (modes js)
 (preprocess (pps js_of_ocaml-ppx)))
```

2.9.2 Separate Compilation

Dune supports two modes of compilation:

- Direct compilation of a bytecode program to JavaScript. This mode allows Js_of_ocaml to perform whole-program deadcode elimination and whole-program inlining.
- Separate compilation, where compilation units are compiled to JavaScript separately and then linked together. This mode is useful during development as it builds more quickly.

The separate compilation mode will be selected when the build profile is `dev`, which is the default. It can also be explicitly specified in an `env` stanza. See *env* for more information.

2.10 JavaScript Compilation With Melange

2.10.1 Introduction

Melange compiles OCaml to JavaScript. It produces one JavaScript file per OCaml module. Melange can be installed with opam:

```
$ opam install melange
```

Dune can build projects using Melange, and it allows the user to produce JavaScript files by defining a *melange.emit* stanza. Dune libraries can be used with Melange by adding `melange` to `(modes ...)` in the *library* stanza.

Melange support is still experimental in Dune and needs to be enabled in the *dune-project* file:

```
(using melange 0.1)
```

Once that's in place, you can use the Melange mode in *library* stanzas `melange.emit` stanzas.

2.10.2 Simple Project

Let's start by looking at a simple project with Melange and Dune. Subsequent sections explain the different concepts used here in further detail.

First, make sure that the *dune-project* file specifies at least version 3.8 of the Dune language, and the Melange extension is enabled:

```
(lang dune 3.17)
(using melange 0.1)
```

Next, write a *dune* file with a *melange.emit* stanza:

```
(melange.emit
 (target output))
```

Finally, add a source file to build:

```
$ echo 'Js.log "hello from melange"' > hello.ml
```

After running `dune build @melange` or just `dune build`, Dune produces the following file structure:

```
.
├── _build
│   └── default
│       └── output
│           └── hello.js
├── dune
├── dune-project
└── hello.ml
```

The resulting JavaScript can now be run:

```
$ node _build/default/output/hello.js
hello from melange
```

2.10.3 Libraries

Adding Melange support to Dune libraries is done as follows:

- `(modes melange)`: adding `melange` to `modes` is required. This field also supports the *Ordered Set Language*.
- `(melange.runtime_deps <deps>)`: optionally, define any runtime dependencies using `melange.runtime_deps`. This field is analog to the `runtime_deps` field used in `melange.emit` stanzas.

2.10.4 melange.emit

New in version 3.8.

The `melange.emit` stanza allows the user to produce JavaScript files from Melange libraries and entry-point modules. It's similar to the OCaml *executable* stanza, with the exception that there is no linking step.

```
(melange.emit
 (target <target>)
 <optional-fields>)
```

<target> is the name of the folder where resulting JavaScript artifacts will be placed. In particular, the folder will be placed under `_build/default/$path-to-directory-of-melange-emit-stanza`.

The result of building a `melange.emit` stanza will match the file structure of the source tree. For example, given the following source tree:

```
├── dune # (melange.emit (target output) (libraries lib))
├── app.ml
└── lib
    ├── dune # (library (name lib) (modes melange))
    └── helper.ml
```

The resulting layout in `_build/default/output` will be as follows:

```
output
├── app.js
└── lib
    ├── lib.js
    └── helper.js
```

<optional-fields> are:

- `(alias <alias-name>)` specifies an alias to which to attach the targets of the `melange.emit` stanza.
  - These targets include the `.js` files generated by the stanza modules, the targets for the `.js` files of any library that the stanza depends on, and any copy rules for runtime dependencies (see `runtime_deps` field below).
  - By default, all stanzas will have their targets attached to an alias `melange`. The behavior of this default alias is exclusive: if an alias is explicitly defined in the stanza, the targets from this stanza will be excluded from the `melange` alias.
  - The targets of `melange.emit` are also attached to the Dune default alias (*@all*), regardless of whether the `(alias ...)` field is present.
- `(module_systems <module_systems>)` specifies the JavaScript import and export format used. The values allowed for <module_systems> are `es6` and `commonjs`.
  - `es6` will follow JavaScript modules, and will produce `import` and `export` statements.
  - `commonjs` will follow CommonJS modules, and will produce *require* calls and export values with `module.exports`.
  - If no extension is specified, the resulting JavaScript files will use `.js`. You can specify a different extension with a pair `(<module_system> <extension>)`, e.g. `(module_systems (es6 mjs))`.
  - Multiple module systems can be used in the same field as long as their extensions are different. For example, `(module_systems commonjs (es6 mjs))` will produce one set of JavaScript files using CommonJS and the `.js` extension, and another using ES6 and the `.mjs` extension.

- `(modules <modules>)` specifies what modules will be built with Melange. By default, if this field is not defined, Dune will use all the `.ml/.re` files in the same directory as the `dune` file. This includes module sources present in the file system as well as modules generated by user rules. You can restrict this list by using an explicit `(modules <modules>)` field. `<modules>` uses the *Ordered Set Language*, where elements are module names and don't need to start with an uppercase letter. For instance, to exclude module `Foo`, use `(modules :standard \ foo)`.
- `(libraries <library-dependencies>)` specifies Melange library dependencies. Melange libraries can only use the simple form, like `(libraries foo pkg.bar)`. Keep in mind the following limitations:
  - The `re_export` form is not supported.
  - All the libraries included in `<library-dependencies>` have to support the `melange` mode (see the section about libraries below).
- `(package <package>)` allows the user to define the JavaScript package to which the artifacts produced by the `melange.emit` stanza will belong.
- `(runtime_deps <paths-to-deps>)` specifies dependencies that should be copied to the build folder together with the `.js` files generated from the sources. These runtime dependencies can include assets like CSS files, images, fonts, external JavaScript files, etc. `runtime_deps` adhere to the formats in *Dependency Specification*. For example `(runtime_deps ./path/to/file.css (glob_files_rec ./fonts/*))`.
- `(emit_stdlib <bool>)` allows the user to specify whether the Melange standard library should be included as a dependency of the stanza or not. The default is `true`. If this option is `false`, the Melange standard library and runtime JavaScript files won't be produced in the target directory.
- `(promote <options>)` promotes the generated `.js` files to the source tree. The options are the same as for the *rule promote mode*. Adding `(promote (until-clean))` to a `melange.emit` stanza will cause Dune to copy the `.js` files to the source tree and `dune clean` to delete them.
- `(preprocess <preprocess-spec>)` specifies how to preprocess files when needed. The default is `no_preprocessing`. Additional options are described in the *Preprocessing Specification* section.
- `(preprocessor_deps (<deps-conf list>))` specifies extra preprocessor dependencies, e.g., if the preprocessor reads a generated file. The dependency specification is described in the *Dependency Specification* section.
- `(compile_flags <flags>)` specifies compilation flags specific to `melc`, the main Melange executable. `<flags>` is described in detail in the *Ordered Set Language* section. It also supports `(:include ...)` forms. The value for this field can also be taken from `env` stanzas. It's therefore recommended to add flags with e.g. `(compile_flags :standard <my options>)` rather than replace them.
- `(root_module <module>)` specifies a `root_module` that collects all listed dependencies in `libraries`. See the documentation for `root_module` in the *library* stanza.
- `(allow_overlapping_dependencies)` is the same as the corresponding field of *library*.
- `(enabled_if <blang expression>)` conditionally disables a melange emit stanza. The JavaScript files associated with the stanza won't be built. The condition is specified using the *Boolean Language*.

2.10.5 Recommended Practices

Keep Bundles Small by Reducing the Number of `melange.emit` Stanzas

It is recommended to minimize the number of `melange.emit` stanzas that a project defines: using multiple `melange.emit` stanzas will cause multiple copies of the JavaScript files to be generated if the same libraries are used across them. As an example:

```
(melange.emit
 (target app1)
 (libraries foo))

(melange.emit
 (target app2)
 (libraries foo))
```

The JavaScript artifacts for library `foo` will be emitted twice in the `_build` folder. They will be present under `_build/default/app1` and `_build/default/app2`.

This can have unexpected impact on bundle size when using tools like Webpack or Esbuild, as these tools will not be able to see shared library code as such, as it would be replicated across the paths of the different stanzas `target` folders.

Faster Builds With `subdir` and `dirs` Stanzas

Melange libraries might be installed from the `npm` package repository, together with other JavaScript packages. To avoid having Dune inspect unnecessary folders in `node_modules`, it is recommended to explicitly include only the folders that are relevant for Melange builds.

This can be accomplished by combining *subdir* and *subdir* stanzas in a `dune` file next to the `node_modules` folder. The *vendored_dirs* stanza can be used to avoid warnings in Melange libraries during the application build. The *data_only_dirs* stanza can be useful as well if you need to override the build rules in one of the packages.

```
(subdir
 node_modules
 (vendored_dirs reason-react)
 (dirs reason-react))
```

2.11 Virtual Libraries

Virtual libraries correspond to Dune's ability to compile parameterised libraries and delay the selection of concrete implementations until linking an executable.

The feature introduces two kinds of libraries: virtual and implementations. A *virtual library* corresponds to an interface (although it may contain partial implementation). An *implementation* of a virtual library fills in all unimplemented modules in the virtual library.

The benefit of this partition is that other libraries may depend on and compile against the virtual library, and they might only select concrete implementations for these virtual libraries when linking executables. An example where this might be useful would be a virtual, cross-platform, `clock` library. This library would have `clock.unix` and `clock.win` implementations. Executable using `clock` or libraries that use `clock` would conditionally select one of the implementations, depending on the target platform.

Note: This feature is sometimes informally called "variants". However, this term refers to a related feature that has been removed in Dune 2.6 in favor of default implementation, and the correct term for the mechanism as a whole is "virtual libraries".

2.11.1 Virtual Library

To define a virtual library, a `virtual_modules` field must be added to an ordinary library stanza, and the version of the Dune language must be at least 1.5. This field defines modules for which only an interface would be present (mli only):

```
(library
 (name clock)
 ;; clock.mli must be present, but clock.ml must not be
 (virtual_modules clock))
```

Apart from this field, the virtual library is defined just like a normal library and may use all the other fields. A virtual library may include other modules (with or without implementations), which is why it's not a pure "interface" library.

Note: the `virtual_modules` field is not merged in `modules`, which represents the total set of modules in a library. If a directory has more than one stanza and thus a `modules` field must be specified, virtual modules still need to be added in `modules`.

2.11.2 Implementation

An implementation for a library is defined as:

```
(library
 (name clock_unix)
 ;; clock.ml must be present, but clock.mli must not be
 (implements clock))
```

The `name` field is slightly different for an implementation than it is for a normal library. The `name` is just an internal name to refer to the implementation, it doesn't correspond to any particular module like it does in the virtual library.

Other libraries may then depend on the virtual library as if it was a regular library:

```
(library
 (name calendar)
 (libraries clock))
```

But when it comes to creating an executable, we must now select a valid implementation for every virtual library that we've used:

```
(executable
 (name birthday-reminder)
 (libraries
  clock_unix ;; leaving this dependency will make dune loudly complain
  calendar))
```

2.11.3 Default Implementation

A virtual library may select a default implementation, which is enabled after variant resolution if no suitable implementation has been found.

```
(library
 (name time)
 (virtual_modules time)
 (default_implementation time-js))
```

The default implementation must live in the same package as the virtual library. In the example above, that would mean that the `time-js` and `time` libraries must be in the same package

2.11.4 Limitations

The current implementation of virtual libraries suffers from a few limitations. Some of these are temporary.

- It's impossible to link more than one implementation for the same virtual library in one executable.
- It's not possible for implementations to introduce new public modules. That is, modules that aren't a part of the virtual library's CMI. Consequently, a module in an implementation either implements a virtual module or is private.
- It isn't possible to load virtual libraries into `utop`. As a result, any directory that contains a virtual library will not work with `$ dune utop`. This is an essential limitation, but it would be best to somehow skip these libraries or provide an implementation for them when loading a toplevel.
- Virtual libraries must be defined using Dune. It's not possible for Dune to implement virtual libraries created outside of Dune. On the other hand, virtual libraries and implementations defined using Dune should be usable with findlib-based build systems.
- It's impossible for a library to be both virtual and implement another library. This isn't very useful, but it could technically be used to create partial implementations. It is possible to lift this restriction if there's enough demand.

2.12 Writing and Running Tests

Dune tries to streamline the testing story as much as possible, so you can focus on the tests themselves and not bother with setting up various test frameworks.

In this section, we'll explain the workflow to deal with tests in Dune. In particular, we'll see how to run the test suite of a project, how to describe your tests to Dune, and how to promote test results as expectation.

We distinguish three kinds of tests:

- Inline tests - written directly inside the `.ml` files of a library
- Custom tests - run an executable, possibly followed by an action such as diffing the produced output.
- Cram tests - expect tests written in Cram style.

2.12.1 Running Tests

Whatever the tests of a project are, the usual way to run tests with Dune is to call `dune runtest` from the shell (or the command alias `dune test`). This will run all the tests defined in the current directory and any subdirectory recursively.

Note that in any case, `dune runtest` is simply shorthand for building the `runtest` alias, so you can always ask Dune to run the tests in conjunction with other targets by passing `@runtest` to `dune build`. For instance:

```
$ dune build @install @runtest
$ dune build @install @test/runtest
```

Running a Single Test

If you would only like to run a single test for your project, you may use `dune exec` to run the test executable (for the sake of this example, `project/tests/myTest.ml`):

```
$ dune exec project/tests/myTest.exe
```

To run *Cram Tests*, you can use the alias that is created for the test. The name of the alias corresponds to the name of the test without the `.t` extension. For directory tests, this is the name of the directory without the `.t` extension. Assuming a `cram-test.t` or `cram-test.t/run.t` file exists, it can be run with:

```
$ dune build @cram-test
```

Running Tests in a Directory

You can also pass a directory argument to run the tests from a subtree. For instance, `dune runtest test` will only run the tests from the `test` directory and any subdirectory of `test` recursively.

2.12.2 Inline Tests

There are several inline tests frameworks available for OCaml, such as ppx_inline_test and qtest. We will use ppx_inline_test as an example because it has the necessary setup to be used with Dune out of the box.

ppx_inline_test allows one to write tests directly inside `.ml` files as follows:

```
let rec fact n = if n = 1 then 1 else n * fact (n - 1)

let%test _ = fact 5 = 120
```

The file must be preprocessed with the `ppx_inline_test` PPX rewriter, so for instance the `dune` file might look like this:

```
(library
 (name foo)
 (preprocess (pps ppx_inline_test)))
```

In order to tell Dune that our library contains inline tests, we have to add an `inline_tests` field:

```
(library
 (name foo)
 (inline_tests)
 (preprocess (pps ppx_inline_test)))
```

We can now build and execute this test by running `dune runtest`. For instance, if we make the test fail by replacing `120` by `0` we get:

```
$ dune runtest
[...]
File "src/fact.ml", line 3, characters 0-25: <<(fact 5) = 0>> is false.

FAILED 1 / 1 tests
```

Note that in this case Dune knew how to build and run the tests without any special configuration. This is because `ppx_inline_test` defines an inline tests backend that's used by the library. Some other frameworks, such as qtest, don't have any special library or PPX rewriter. To use such a framework, you must tell Dune about it, as it cannot guess. You can do that by adding a `backend` field:

```
(library
 (name foo)
 (inline_tests (backend qtest.lib)))
```

In the example above, the name *qtest.lib* comes from the *public_name* field in *qtest*'s own *dune* file.

Note that using `ppx_inline_test` requires that the opam package `ppx_inline_test` be installed in your switch. If you use `ppx_inline_test` in a package then that package must *unconditionally* depend on `ppx_inline_test` (ie. `ppx_inline_test` can't be a `with-test` dependency).

Inline Expectation Tests

Inline expectation tests are a special case of inline tests where written OCaml code prints something followed by what you expect this code to print. For instance, using ppx_expect:

```
let%expect_test _ =
  print_endline "Hello, world!";
  [%expect{|
    Hello, world!
  |}]
```

The test procedure consist of executing the OCaml code and replacing the contents of the `[%expect]` extension point by the real output. You then get a new file that you can compare to the original source file. Expectation tests are a neat way to write tests as the following test elements are clearly identified:

- The code of the test
- The test expectation
- The test outcome

You can have a look at this blog post to find out more about expectation tests. To Dune, the workflow for expectation tests is always as follows:

- Write the test with some empty expect nodes in it
- Run the tests
- Check the suggested correction and promote it as the original source file if you are happy with it

Dune makes this workflow very easy. Simply add `ppx_expect` to your list of PPX rewriters as follows:

```
(library
 (name foo)
 (inline_tests)
 (preprocess (pps ppx_expect)))
```

Then calling `dune runtest` will run these tests, and in case of mismatch, Dune will print a diff of the original source file and the suggested correction. For instance:

```
$ dune runtest
[...]
-src/fact.ml
+src/fact.ml.corrected
File "src/fact.ml", line 5, characters 0-1:
let rec fact n = if n = 1 then 1 else n * fact (n - 1)

let%expect_test _ =
  print_int (fact 5);
-  [%expect]
+  [%expect{| 120 |}]
```

In order to accept the correction, simply run:

```
$ dune promote
```

You can also make Dune automatically accept the correction after running the tests by typing:

```
$ dune runtest --auto-promote
```

Finally, some editor integration can make the editor do the promotion, which in turn makes the workflow even smoother.

Running a Subset of the Test Suite

You may also run a group of tests located under a directory with:

```
$ dune runtest mylib/tests
```

The above command will run all tests defined in tests and its subdirectories.

Running Tests in Bytecode or JavaScript

By default, Dune runs inline tests in native mode, unless native compilation isn't available. In which case, it runs them in bytecode. You can change this setting to choose the modes that tests should run in. To do this, add a `modes` field to the `inline_tests` field. Available modes are:

- `byte` for running tests in byte code
- `native` for running tests in native mode
- `best` for running tests in native mode with fallback to byte code, if native compilation is not available
- `js` for running tests in JavaScript using Node.js

For instance:

```
(library
 (name foo)
 (inline_tests (modes byte best js))
 (preprocess (pps ppx_expect)))
```

Specifying Inline Test Dependencies

If your tests are reading files, you must tell Dune by adding a `deps` field the `inline_tests` field. The argument of this `deps` field follows the usual *Dependency Specification*. For instance:

```
(library
 (name foo)
 (inline_tests (deps data.txt))
 (preprocess (pps ppx_expect)))
```

Passing Special Arguments to the Test Runner

Under the hood, a test executable is built by Dune. Depending on the backend used, this runner might take useful command line arguments. You can specify such flags by using a `flags` field, such as:

```
(library
 (name foo)
 (inline_tests (flags (-foo bar)))
 (preprocess (pps ppx_expect)))
```

The argument of the `flags` field follows the *Ordered Set Language*.

Passing Special Arguments to the Test Executable

To control how the test executable is built, it's possible to customize a subset of compilation options for an executable using the `executable` field. Dune gives you this ability by simply specifying command line arguments as flags. You can specify such flags by using `flags` field. For instance:

```
(library
 (name foo)
 (inline_tests
  (flags (-foo bar)
  (executable
   (flags (-foo bar))))
  (preprocess (pps ppx_expect))))
```

The argument of the `flags` field follows the *Ordered Set Language*.

Using Additional Libraries in the Test Runner

When tests are not part of the library code, it's possible that tests require additional libraries than the library being tested. This is the case with qtest, as tests are written in comments. You can specify such libraries using a `libraries` field, such as:

```
(library
 (name foo)
 (inline_tests
  (backend qtest)
  (libraries bar)))
```

Changing the Flags of the Linking Step of the Test Runner

You can use the `link_flags` field to change the linker flags passed to `ocamlopt` when building the test runner. By default, the linking flags are `-linkall`. You probably want to keep `-linkall` as one of the new list of flags (unless you know what you are doing), forcing the linker to load your test module, since the test runner doesn't depend on anything itself. This field supports `(:include ...)` forms.

```
(library
 (name foo)
 (inline_tests
  (executable
   (link_flags -linkall -noautolink -cclib -Wl,-Bstatic -cclib -lm)))
 (preprocess (pps ppx_expect)))
```

Defining Your Own Inline Test Backend

If you are writing a test framework (or for other specific cases), you might want to define your own inline tests backend. If your framework is naturally implemented by a library or PPX rewriter that's necessary to write tests, you should define this library as a backend. Otherwise simply create an empty library with your chosen backend's name.

In order to define a library as an inline tests backend, simply add an `inline_tests.backend` field to the library stanza. An inline tests backend is specified by four parameters:

1. How to create the test runner
2. How to build the test runner
3. How to run the test runner
4. Optionally how to run the test runner to list partitions

These four parameters can be specified inside the `inline_tests.backend` field, which accepts the following fields:

```
(generate_runner       <action>)
(runner_libraries     (<ocaml-libraries>))
(flags                 <flags>)
(list_partitions_flags <flags>)
(extends              (<backends>))
```

For instance:

`<action>` follows the *Actions* specification. It describes an action that should be executed in the library's directory using this backend for their tests. It's expected that the action will produce some OCaml code on its standard output. This code will constitute the test runner. The action can use the following additional variables:

- `%{library-name}` — the name of the library being tested
- `%{impl-files}` — the list of implementation files in the library, i.e., all the `.ml` and `.re` files
- `%{intf-files}` — the list of interface files in the library, i.e., all the `.mli` and `.rei` files

The `runner_libraries` field specifies what OCaml libraries the test runner uses. For instance, if the `generate_runner` actions generates something like `My_test_framework.runtests ()`, then you should probably put `my_test_framework` in the `runner_libraries` field.

If your test runner needs specific flags, you should pass them in the `flags` field. You can use the `%{library-name}` variable in this field.

If your test runner supports test partitions, you should pass the flags necessary for listing partitions in the `list_partitions_flags` field. In such scenario, the `flags` field will also accepts a `%{partition}` variable.

Finally, a backend can be an extension of another backend. In this case, you must specify this in the `extends` field. For instance, ppx_expect is an extension of ppx_inline_test. It's possible to use a backend with several extensions in a library; however, there must be exactly one *root backend*, i.e., exactly one backend that isn't an extension of another one.

When using a backend with extensions, the various fields are simply concatenated. The order in which they are concatenated is unspecified; however, if a backend `b` extends a backend `a`, then `a` will always come before `b`.

Example of Backend

In this example, we put tests in comments of the form:

```
(*TEST: assert (fact 5 = 120) *)
```

The backend for such a framework looks like this:

```
(library
 (name simple_tests)
 (inline_tests.backend
  (generate_runner (run sed "s/(\\*TEST:\\(.*\\)\\*)/let () = \\1;;/" %{impl-files}))))
```

Now all you have to do is write `(inline_tests ((backend simple_tests)))` wherever you want to write such tests. Note that this is only an example. We don't recommend using `sed` in your build, as this would cause portability problems.

2.12.3 Custom Tests

We said in *Running tests* that to run tests, Dune simply builds the `runtest` alias. As a result, you simply need to add an action to this alias in any directory in order to define custom tests. For instance, if you have a binary `tests.exe` that you want to run as part of running your test suite, simply add this to a `dune` file:

```
(rule
 (alias  runtest)
 (action (run ./tests.exe)))
```

Hence to define a test, a pair of alias and executable stanzas are required. To simplify this common pattern, Dune provides a *tests* stanza to define multiple tests and their aliases at once:

```
(tests (names test1 test2))
```

Diffing the Result

It's often the case that we want to compare the actual output of a test to an expected one. For that, Dune offers the `diff` command, which in essence is the same as running the `diff` tool, except that it's more integrated in Dune, especially with the `promote` command. For instance, let's consider this test:

```
(rule
(with-stdout-to tests.output (run ./tests.exe)))

(rule
 (alias runtest)
 (action (diff tests.expected tests.output)))
```

After having run `tests.exe` and dumping its output to `tests.output`, Dune will compare the latter to `tests.expected`. In case of mismatch, Dune will print a diff and then the `dune promote` command can be used to copy over the generated `test.output` file to `tests.expected` in the source tree.

Alternatively, the *tests* also supports this style of tests.

```
(tests (names tests))
```

Dune expects the existence of a `tests.expected` file to infer that this is an expected test.

This provides a nice way of dealing with the usual *write code*, *run*, and *promote* cycle of testing. For instance:

```
$ dune runtest
[...]
-tests.expected
+tests.output
File "tests.expected", line 1, characters 0-1:
-Hello, world!
+Good bye!
$ dune promote
Promoting _build/default/tests.output to tests.expected.
```

Note that if available, the diffing is done using the patdiff tool, which displays nicer looking diffs than the standard `diff` tool. You can change that by passing `--diff-command CMD` to Dune.

2.12.4 Cram Tests

Cram tests are expectation tests written in a shell-like syntax. They are ideal for testing binaries. Cram tests are automatically discovered from files or directories with a `.t` extension. By default, this has been enabled since Dune 3.0. For older versions, it must be manually enabled in the `dune-project` file:

```
(lang dune 2.7)
(cram enable)
```

File Tests

To define a standalone test, we create a `.t` file. For example, `foo.t`:

```
Simplest possible Cram test
  $ echo "testing"
```

This simple example demonstrates two components of Cram tests: comments and commands. See *Cram Tests* for a description of the syntax.

To run the test and promote the results:

```
$ dune runtest
$ dune promote
```

We now see the output of the command:

```
Simplest possible cram test
  $ echo "testing"
  testing
```

This is the main advantage of expect tests. We don't need to write assertions manually; instead we detect failure when the command produces a different output than what is recorded in the test script.

For example, here's an example of how we'd test the `wc` utility. `wc.t`:

```
We create a test artifact called "foo"
  $ cat >foo <<EOF
  > foo
  > bar
  > baz
  > EOF

After creating the fixture, we want to verify that ``wc`` gives us the right
result:
  $ wc -l foo | awk '{ print $1 }'
  4
```

The above example uses the doc syntax, piping the subsequent lines to `cat`. This is convenient for creating small test artifacts.

Directory Tests

In the above example we used `cat` to create the test artifact, but what if there are too many artifacts to comfortably fit in test file? Or some of the artifacts are binary?

It's possible to include the artifacts as normal files or directories, provided the test is defined as a directory. The name of the test directory must end with `.t` and must include a `run.t` as the test script. Everything else in that directory is treated as raw data for the test. It's not possible to define rules using `dune` files in such a directory.

We convert the `wc` test above into a directory test `wc.t`:

```
$ ls wc.t
run.t foo.txt bar/
```

This defines a directory test `wc.t` which must include a `run.t` file as the test script, with `fool.txt` and `bar` are test artifacts. We may then access their contents in the test script `run.t`:

```
Testing wc:
  $ wc -l foo | awk '{ print $1 }'
  4
  $ wc -l $(ls bar) | awk '{ print $1 }'
  1231
```

See also:

(cram) stanza reference

Testing an OCaml Program

The most common testing situation involves testing an executable that is defined in Dune. For example:

```
(executable
 (name wc)
 (public_name wc))
```

To use this binary in the Cram test, we should depend on the binary in the test:

```
(cram
 (deps %{bin:wc}))
```

Sandboxing

Since Cram tests often create intermediate artifacts, it's important that Cram tests are executed in a clean environment. This is why all Cram tests are sandboxed. To respect sandboxing, every test should specify dependency on any artifact that might rely on using the `deps` field.

See *Sandboxing* for details about the sandboxing mechanism.

Test Output Sanitation

In some situations, Cram tests emit non portable or non-deterministic output. We recommend sanitising such outputs using pipes. For example, we can scrub the OCaml magic number using `sed` as follows:

```
$ ocamlc -config | grep "cmi_magic_number:" | sed 's/Caml.*/$SPECIAL_CODE/'
cmi_magic_number: $SPECIAL_CODE
```

By default, Dune will scrub some paths from the output of the tests. The default list of paths is:

- The `PWD` of the test will be replaced by `$TESTCASE_ROOT`
- The temporary directory for the current script will be replaced by `$TMPDIR`

To add additional paths to this sanitation mechanism, it's sufficient to modify the standard BUILD_PATH_PREFIX_MAP environment variable. For example:

```
$ export BUILD_PATH_PREFIX_MAP="HOME=$HOME:$BUILD_PATH_PREFIX_MAP"
$ echo $HOME
$HOME
```

Note: Unlike Dune's version of Cram, the original specification for Cram supports regular expression and glob filtering for matching output. We chose not to implement this feature because it breaks the test, diff, and accept cycle. With regex or glob matching, the output must now be manually inspected and possibly updated. We consider the postprocessing approach described here as superior and will not introduce output matchers.

2.13 How to Bundle Resources

This guide will show you how to configure Dune to generate modules with string resources from other files in your project.

2.13.1 Folder Structure

```
$ tree src
src
└── lib
    └── my_lib
        ├── dune
        └── resources
            └── site.css
```

2.13.2 Dune Configuration

See *progn* and *with-<outputs>-to*.

```
(rule
 (with-stdout-to
  css.ml
  (progn
   (echo "let css = {|")
   (cat resources/site.css)
   (echo "|}"))))
```

2.13.3 Using the Bundled Resource

```
let () = Printf.printf "%s" Css.css
```

2.14 How to Load a Project in a Toplevel

It is possible to use OCaml code in an interactive way, by typing an expression, which gets evaluated and its result printed. Such a program is called a *toplevel*, or REPL (Read-Eval-Print Loop).

The compiler distribution comes with a small REPL called simply `ocaml`, and the community has developed enhanced versions such as UTop.

2.14.1 Building a Specialized UTop Executable

It is possible to generate a specialized version of UTop that embeds the current project. To do so, use the following command:

```
$ dune utop
```

The interactive session will start with all the modules loaded.

If some of the libraries are PPX rewriters, the phrases you type in the toplevel will be rewritten with these PPX rewriters. Similarly, PPX derivers defined in the project will be available.

2.14.2 Loading the Project in a Toplevel

It is also possible to load Dune projects in any toplevel. To do that, simply execute the following in your toplevel:

```
# #use_output "dune ocaml top";;
```

`dune ocaml top` is a Dune command that builds all the libraries in the current directory and subdirectories and outputs the relevant toplevel directives (`#directory` and `#load`) to make the various modules available in the toplevel.

2.14.3 Loading a Single Module in a Toplevel

It's also possible to load individual modules for interactive development. Use the following dune command:

```
# #use_output "dune ocaml top-module foo.ml";;
```

This will print directives that will load `foo.ml` without sealing it behind `foo.mli`. This is particularly useful for peeking and prodding at a module's internals.

2.15 Using Rule Generation

Sometimes it can be useful to generate Dune rules that depend on the file system layout or on the content of configuration files. This often happens for integration tests.

In this document, we will see two ways to encode this behavior.

We suppose that we are testing an executable named `tool`. There are some input files named `*.input`, output files named `*.output`, and we want to ensure that when running `tool` on `x.input`, the standard output corresponds to `x.output`.

2.15.1 The Generate-Include-Commit Pattern

Note: This is the most common way to do this. It has a couple drawbacks listed below, but you should start with this pattern.

What we are going to do is:

- generate a `dune.inc` file;
- include it in our main `dune` file;

- commit the generated code in the source repository.

This creates a loop: a program (the "generator") looks at the file system and creates a `dune.inc` file. Changes to the file system (for example, if a test is added) mean that a change in `dune.inc` will be promoted. These generated rules are included in the main `dune` file, so `dune runtest` will run the tests. Finally, the generated file is part of the source repository, so it is not necessary to run several commands to run the test suite.

Let's expand a bit on how to achieve this.

Generating a `dune.inc` File

Create a `gen` subdirectory and create a `gen.ml` file in it:

gen/gen.ml

```
let generate_rules base =
  Printf.printf
    {|
        (rule
         (with-stdout-to %s.gen
          (run %%{bin:tool} %s.input)))

        (rule
         (alias runtest)
         (action
          (diff %s.output %s.gen)))
        |}
    base base base base

let () =
  Sys.readdir "."
  |> Array.to_list
  |> List.sort String.compare
  |> List.filter_map (Filename.chop_suffix_opt ~suffix:".input")
  |> List.iter generate_rules
```

Create a `dune` file in that directory:

gen/dune

```
(executable
 (name gen))
```

This defines an executable that lists `*.input` files in the current directory and outputs rules on its standard output.

Note: It is important to sort the input files to ensure that the output is independent from the order in which `Sys.readdir` returns the files .

For each input file, we output two rules:

- The first one creates a `x.gen` file that corresponds to the actual output.
- The second uses a *diff* action to compare the actual output to the expected output. If it is different, `dune runtest` will display the difference, which can be accepted by `dune promote`.

Note: It is possible to have more complicated logic here. For example, to pass different arguments to `tool` depending on the presence of a `*.args` file. To do that, check if `*.args` exists in `generate_rules` and emit a different `(run ...)` action.

Including it in the Main `dune` File

Our main test `dune` file contains the following:

dune

```
(include dune.inc)

(rule
 (deps (source_tree .))
 (with-stdout-to
  dune.inc.gen
  (run gen/gen.exe)))

(rule
 (alias runtest)
 (action
  (diff dune.inc dune.inc.gen)))
```

In addition to including the contents of `dune.inc`, we use the same pattern as before: `dune.inc.gen` is the actual output of the generator, and `dune.inc` is the expected output. At runtime, the generator will read the contents of the current directory (where the `*.input` and `*.output` files are located), so we record `(source_tree .)` as a dependency to make it run again if a file is created, for example.

Commit the Generated Code In The Source Repository

To make this work, we have a final step to do. We have to add the generated file to our source tree. But since it is generated, we will have to first create an empty file, run the test, and promote the result.

```
$ touch dune.inc
$ dune runtest
+ (rule
+  (with-stdout-to a.gen
+   (run %{bin:tool} a.input)))
+
+ (rule
+  (alias runtest)
+  (action
+   (diff a.output a.gen)))
$ dune promote dune.inc
$ git add dune.inc
```

Now, running `dune runtest` will run the test suite.

Notes

This pattern is "correct": it will execute all tests and make sure the list of tests is up to date. But when adding a test, it is necessary to first run `dune runtest`, promote the result, and then re-run `dune runtest` to actually run the test (and possibly promote the result of the test itself).

There is a variant of this pattern which will promote the output automatically instead of using a manual promotion step. This variant can be used either for the test list or for the individual tests.

To use it in the test list, replace the `dune` file by this version:

dune (alternative version)

```
(include dune.inc)

(rule
 (mode promote)
 (alias runtest)
 (deps (source_tree .))
 (with-stdout-to
  dune.inc
  (run gen/gen.exe)))
```

Using this version, `dune runtest` will directly replace `dune.inc` with an updated version.

Another caveat of this approach is that the generator needs to emit the same output on all systems. For example, if some tests should be skipped on Linux, the generator can not just filter the corresponding tests depending on `Sys.os_type`. It has to consistently emit a `(enabled_if)` field for the rules.

2.15.2 Using `(dynamic_include)`

New in version 3.14.

This technique relies on *dynamic_include*, which is more flexible than *include*. The difference is that the intermediate `dune.inc` file does not need to be part of the source tree. It will only be generated by a rule and be present in the `_build` directory.

At first it looks like it would be possible to reuse the same pattern as above: change `include` to `dynamic_include` and delete the `dune.inc` file. However, it is not possible. The reason is that rules are loaded per directory, and there needs to be a strict order (no cycles) between directories for this to work.

So, instead we are going to:

- generate `dune.inc` in a subdirectory named `generate`, and
- include these rules in a subdirectory named `run`.

These subdirectories do not need to be actual directories. They can be emulated through *subdir*.

To do this, we can create the following `dune` file in the same directory as the `*.input` and `*.output` files.

dune

```
(executable
 (name gen))

(subdir run
 (dynamic_include ../generate/dune.inc))
```

(continues on next page)

(continued from previous page)

```
(subdir generate
 (rule
  (deps (glob_files ../*.input))
  (action
   (with-stdout-to dune.inc
    (run ../gen.exe)))))
```

Then create the following `gen.ml` file. Note that here we can define it in the same directory.

gen.ml

```
let generate_rules base =
  Printf.printf
    {|
        (rule
         (with-stdout-to %s.gen
          (run %%{bin:tool} ../%s.input)))

        (rule
         (alias runtest)
         (action
          (diff ../%s.output %s.gen)))
        |}
    base base base base

let () =
  Sys.readdir ".." |> Array.to_list |> List.sort String.compare
  |> List.filter_map (Filename.chop_suffix_opt ~suffix:".input")
  |> List.iter generate_rules
```

There are a few differences from the generator above because this one is going to be invoked from subdirectories, so it is necessary to refer to the `..` directory both in the input (which files to read) and in the output (how the rules are executed).

These two files are enough. `dune runtest` is going to generate the rules and interpret them in a single command.

Notes

This approach is shorter, but it might be more difficult to debug because changes to the generated rules will not be visible. Also, it works in that case, but it is not possible to generate all kinds of stanzas with that pattern. See *dynamic_include* for more information about the limitations.

CHAPTER

THREE

REFERENCE MANUAL

These documents specify the various features and languages present in Dune.

3.1 dune

dune files are the main part of Dune. They are used to describe libraries, executables, tests, and everything Dune needs to know about.

The syntax of dune files is described in *Lexical Conventions*.

dune files are composed of stanzas, as shown below:

```
(library
 (name mylib)
 (libraries base lwt))

(rule
 (target foo.ml)
 (deps   generator/gen.exe)
 (action (run %{deps} -o %{target})))
```

The following pages describe the available stanzas and their meanings.

3.1.1 executable

The `executable` stanza must be used to describe an executable. The format of executable stanzas is as follows:

```
(executable
 (name <name>)
 <optional-fields>)
```

`<name>` is a module name that contains the executable's main entry point. There can be additional modules in the current directory; you only need to specify the entry point. Given an `executable` stanza with `(name <name>)`, Dune will know how to build `<name>.exe`. If requested, it will also know how to build `<name>.bc` and `<name>.bc.js` (Dune 2.0 and up also need specific configuration (see the `modes` optional field below)).

`<name>.exe` is a native code executable, `<name>.bc` is a bytecode executable which requires `ocamlrun` to run, and `<name>.bc.js` is a JavaScript generated using `js_of_ocaml`.

Please note: in case native compilation is not available, `<name>.exe` will be a custom bytecode executable, in the sense of `ocamlc -custom`. This means it's a native executable that embeds the `ocamlrun` virtual machine as well as the bytecode, so you can always rely on `<name>.exe` being available. Moreover, it is usually preferable to use `<name>.exe`

in custom rules or when calling the executable by hand because running a bytecode executable often requires loading shared libraries that are locally built. This requires additional setup, such as setting specific environment variables, which Dune doesn't do at the moment.

Native compilation isn't available when there is no `ocamlopt` binary at the same place as `ocamlc` was found.

Executables can also be linked as object or shared object files. See *linking modes* for more information.

Starting from Dune 3.0, it's possible to automatically generate empty interface files for executables. See *executables_implicit_empty_intf*.

`<optional-fields>` are:

- `(public_name <public-name>)` specifies that the executable should be installed under this name. It's the same as adding the following stanza to your `dune` file:

  ```
  (install
   (section bin)
   (files (<name>.exe as <public-name>)))
  ```

  As a special case, `(public_name -)` is the same as if the field was absent.

- `(package <package>)` if there is a `(public_name ...)` field, this specifies the package the executables are part of it.

- `(libraries <library-dependencies>)` specifies the library dependencies. See *Library Dependencies* for more details.

- `(link_flags <flags>)` specifies additional flags to pass to the linker. This field supports `(:include ...)` forms.

- `(link_deps (<deps-conf list>))` specifies the dependencies used only by the linker, i.e., when using a version script. See *Dependency Specification* for more details.

- `(modules <modules>)` specifies which modules in the current directory Dune should consider when building this executable. Modules not listed here will be ignored and cannot be used inside the executable described by the current stanza. It is interpreted in the same way as the `(modules ...)` field of *library*.

- `(root_module <module>)` specifies a `root_module` that collects all listed dependencies in `libraries`. See the documentation for `root_module` in the library stanza.

- `(modes (<modes>))` sets the *linking modes*. The default is `(exe)`. Before Dune 2.0, it formerly was `(byte exe)`.

- `(preprocess <preprocess-spec>)` is the same as the `(preprocess ...)` field of *library*.

  `(preprocessor_deps ...)` field of *library*.

- `(preprocessor_deps (<deps-conf list>))` is the same as the

- `js_of_ocaml`: See the section about *js_of_ocaml*

- `flags`, `ocamlc_flags`, and `ocamlopt_flags`: See *OCaml Flags*.

- `(modules_without_implementation <modules>)` is the same as the corresponding field of *library*.

- `(allow_overlapping_dependencies)` is the same as the corresponding field of *library*.

- `(optional)` is the same as the corresponding field of *library*.

- `(enabled_if <blang expression>)` is the same as the corresponding field of *library*.

- `(promote <options>)` allows promoting the linked executables to the source tree. The options are the same as for the *rule promote mode*. Adding `(promote (until-clean))` to an `executable` stanza will cause Dune to copy the `.exe` files to the source tree and use `dune clean` to delete them.

- `(foreign_stubs <foreign-stubs-spec>)` specifies foreign source files, e.g., C or C++ stubs, to be linked into the executable. See *Foreign Sources, Archives, and Objects* for more details.
- `(foreign_archives <foreign-archives-list>)` specifies archives of foreign object files to be linked into the executable. See the section *Foreign Archives* for more details.
- `(forbidden_libraries <libraries>)` ensures that the given libraries are not linked in the resulting executable. If they end up being pulled in, either through a direct or transitive dependency, Dune fails with an error message explaining how the library was pulled in. This field has been available since Dune 2.0.
- `(embed_in_plugin_libraries <library-list>)` specifies a list of libraries to link statically when using the `plugin` linking mode. By default, no libraries are linked in. Note that you may need to also use the `-linkall` flag if some of the libraries listed here are not referenced from any of the plugin modules.
- `(ctypes <ctypes field>)` instructs Dune to use ctypes stubgen to process your type and function descriptions for binding system libraries, vendored libraries, or other foreign code. See *Stub Generation with Dune Ctypes* for a full reference. This field is available since the 3.0 version of the Dune language.
- `(empty_module_interface_if_absent)` causes the generation of empty interfaces for every module that does not have an interface file already. Useful when modules are used solely for their side-effects. This field is available since the 3.0 version of the Dune language.

Linking Modes

The `modes` field allows selecting which linking modes will be used to link executables. Each mode is a pair `(<compilation-mode> <binary-kind>)`, where `<compilation-mode>` describes whether the bytecode or native code backend of the OCaml compiler should be used and `<binary-kind>` describes what kind of file should be produced.

`<compilation-mode>` must be `byte`, `native`, or `best`, where `best` is `native` with a fallback to bytecode when native compilation isn't available.

`<binary-kind>` is one of:

- `c` for producing OCaml bytecode embedded in a C file
- `exe` for normal executables
- `object` for producing static object files that can be manually linked into C applications
- `shared_object` for producing object files that can be dynamically loaded into an application. This mode can be used to write a plugin in OCaml for a non-OCaml application.
- `js` for producing JavaScript from bytecode executables, see *explicit_js_mode*.
- `plugin` for producing a plugin (`.cmxs` if native or `.cma` if bytecode).

For instance the following `executables` stanza will produce bytecode executables and native shared objects:

```
(executables
  (names a b c)
  (modes (byte exe) (native shared_object)))
```

Additionally, you can use the following shorthands:

- `c` for `(byte c)`
- `exe` for `(best exe)`
- `object` for `(best object)`
- `shared_object` for `(best shared_object)`

- `byte` for `(byte exe)`
- `native` for `(native exe)`
- `js` for `(byte js)`
- `plugin` for `(best plugin)`

For instance, the following `modes` fields are all equivalent:

```
(modes (exe object shared_object))
(modes ((best exe)
        (best object)
        (best shared_object)))
```

Lastly, use the special mode `byte_complete` for building a bytecode executable as a native self-contained executable, i.e., an executable that doesn't require the `ocamlrun` program to run and doesn't require the C stubs to be installed as shared object files.

The extensions for the various linking modes are chosen as follows:

`%{ext_obj}` and `%{ext_dll}` are the extensions for object and shared object files. Their value depends on the OS. For instance, on Unix `%{ext_obj}` is usually `.o` and `%{ext_dll}` is usually `.so`, while on Windows `%{ext_obj}` is `.obj` and `%{ext_dll}` is `.dll`.

Up to version 3.0 of the Dune language, when `byte` is specified but none of `native`, `exe`, or `byte_complete` are specified, Dune implicitly adds a linking mode that's the same as `byte_complete`, but it uses the extension `.exe`. `.bc` files require additional files at runtime that aren't currently tracked by Dune, so they don't run `.bc` files during the build. Run the `.bc.exe` or `.exe` ones instead, as these are self-contained.

Lastly, note that `.bc` executables cannot contain C stubs. If your executable contains C stubs you may want to use `(modes exe)`.

js_of_ocaml

In `library` and `executables` stanzas, you can specify `js_of_ocaml` options using `(js_of_ocaml (<js_of_ocaml-options>))`.

`<js_of_ocaml-options>` are all optional:

- `(flags <flags>)` to specify flags passed to `js_of_ocaml compile`. This field supports `(:include ...)` forms
- `(build_runtime_flags <flags>)` to specify flags passed to `js_of_ocaml build-runtime`. This field supports `(:include ...)` forms
- `(link_flags <flags>)` to specify flags passed to `js_of_ocaml link`. This field supports `(:include ...)` forms
- `(javascript_files (<files-list>))` to specify `js_of_ocaml` JavaScript runtime files.

`<flags>` is specified in the *Ordered Set Language*.

The default value for `(flags ...)` depends on the selected build profile. The build profile `dev` (the default) will enable sourcemap and the pretty JavaScript output.

See *JavaScript Compilation With Js_of_ocaml* for more information.

3.1.2 executables

There is a very subtle difference in the naming of these stanzas. One is `executables`, plural, and the other is `executable`, singular. The `executables` stanza is very similar as the `executable` stanza but can be used to to describe several executables sharing the same configuration, so the plural `executables` stanza is used to describe more than one executable.

It shares the same fields as the `executable` stanza, except that instead of `(name ...)` and `(public_name ...)` you must use the plural versions as well:

- `(names <names>)` where `<names>` is a list of entry point names. Compare with `executable`, where you only need to specify the modules containing the entry point of each executable.
- `(public_names <names>)` describes under what name to install each executable. The list of names must be of the same length as the list in the `(names ...)` field. Moreover, you can use `-` for executables that shouldn't be installed.

However, using `executables` the executables defined in the stanza are allowed to share modules.

Given modules `Foo`, `Bar` and `Baz` the usage of `executables` can simplify the code:

```
(executables
  (names foo bar))
```

Instead of the more complex

```
(library
  (name baz)
  (modules baz))

(executable
  (name foo)
  (modules foo)
  (libraries baz))

(executable
  (name bar)
  (modules bar)
  (libraries baz))
```

3.1.3 library

The `library` stanza must be used to describe OCaml libraries. The format of library stanzas is as follows:

```
(library
 (name <library-name>)
 <optional-fields>)
```

`<library-name>` is the real name of the library. It determines the names of the archive files generated for the library as well as the module name under which the library will be available, unless `(wrapped false)` is used (see below). It must be a valid OCaml module name, but it doesn't need to start with an uppercase letter.

For instance, the modules of a library named `foo` will be available as `Foo.XXX`, outside of `foo` itself; however, it is allowed to write an explicit `Foo` module, which will be the library interface. You are free to expose only the modules you want.

Please note: by default, libraries and other things that consume OCaml/Reason modules only consume modules from the directory where the stanza appear. In order to declare a multi-directory library, you need to use the *include_subdirs* stanza.

<optional-fields> are:

- (public_name <name>) - the name under which the library can be referred as a dependency when it's not part of the current workspace, i.e., when it's installed. Without a (public_name ...) field, the library won't be installed by Dune. The public name must start with the package name it's part of and optionally followed by a dot, then anything else you want. The package name must also be one of the packages that Dune knows about, as determined by the logic described in *Packages*.
- (package <package>) installs a private library under the specified package. Such a library is now usable by public libraries defined in the same project. The Findlib name for this library will be <package>.__private__.<name>; however, the library's interface will be hidden from consumers outside the project.
- (synopsis <string>) should give a one-line description of the library. This is used by tools that list installed libraries
- (modules <modules>) specifies what modules are part of the library. By default, Dune will use all the .ml/.re files in the same directory as the dune file. This includes ones present in the file system as well as ones generated by user rules. You can restrict this list by using a (modules <modules>) field. <modules> uses the *Ordered Set Language*, where elements are module names and don't need to start with an uppercase letter. For instance, to exclude module Foo, use (modules (:standard \ foo)). Starting in Dune 3.13, one can also use special forms (:include <file>) and variables such as %{read-lines:<file>} in this field to customize the list of modules using Dune rules. The dependencies introduced in this way *must live in a different directory that the stanza making use of them*.
- (libraries <library-dependencies>) specifies the library's dependencies. See *Library Dependencies* for more details.
- (wrapped <boolean>) specifies whether the library modules should be available only through the top-level library module, or if they should all be exposed at the top level. The default is true, and it's highly recommended to keep it this way. Because OCaml top-level modules must all be unique when linking an executables, polluting the top-level namespace will make your library unusable with other libraries if there is a module name clash. This option is only intended for libraries that manually prefix all their modules by the library name and to ease porting of existing projects to Dune.
- (wrapped (transition <message>)) is the same as (wrapped true), except it will also generate unwrapped (not prefixed by the library name) modules to preserve compatibility. This is useful for libraries that would like to transition from (wrapped false) to (wrapped true) without breaking compatibility for users. The deprecation notices for the unwrapped modules will include <message>.
- (preprocess <preprocess-spec>) specifies how to preprocess files when needed. The default is no_preprocessing, and other options are described in *Preprocessing Specification*.
- (preprocessor_deps (<deps-conf list>)) specifies extra preprocessor dependencies preprocessor, i.e., if the preprocessor reads a generated file. The specification of dependencies is described in *Dependency Specification*.
- (optional) - if present, it indicates that the library should only be built and installed if all the dependencies are available, either in the workspace or in the installed world. Use this to provide extra features without adding hard dependencies to your project
- (foreign_stubs <foreign-stubs-spec>) specifies foreign source files, e.g., C or C++ stubs, to be compiled and packaged together with the library. See the section *Foreign Sources, Archives, and Objects* for more details. This field replaces the now-deleted fields c_names, c_flags, cxx_names, and cxx_flags.
- (foreign_archives <foreign-archives-list>) specifies archives of foreign object files to be packaged with the library. See the section *Foreign Archives* for more details. This field replaces the now-deleted field

`self_build_stubs_archive`.

- `(install_c_headers (<names>))` - if your library has public C header files that must be installed, you must list them in this field, without the `.h` extension. You should favor the `public_headers` field starting from 3.8.
- `(public_headers (<files>))` - if your library has public C header files that must be installed, you must list them in this field. This field accepts globs in the form of `(glob_files_rec <glob>)` and `(glob_files <glob>)` fields to specify multiple files.

  The advantage of this field over `install_c_headers` is that it preserves the directory structures of the headers relative to the library stanza. Additionally, it allows to specify the extensions of the header files, which allows alternative extensions such as `.hh` or `.hpp`.
- `(modes <modes>)` is for modes which should be built by default. The most common use for this feature is to disable native compilation when writing libraries for the OCaml toplevel. The following modes are available: `byte`, `native` and `best`. `best` is `native` or `byte` when native compilation isn't available.
- `(no_dynlink)` disables dynamic linking of the library. This is for advanced use only. By default, you shouldn't set this option.
- `(kind <kind>)` sets the type of library. The default is `normal`, but other available choices are `ppx_rewriter` and `ppx_deriver`. They must be set when the library is intended to be used as a PPX rewriter or a `[@@deriving ...]` plugin. The reason `ppx_rewriter` and `ppx_deriver` are split is historical, and hopefully we won't need two options soon. Both PPX kinds support an optional field: `(cookies <cookies>)`, where `<cookies>` is a list of pairs `(<name> <value>)` with `<name>` being the cookie name and `<value>` a string that supports *Variables* evaluated by each preprocessor invocation (note: libraries that share cookies with the same name should agree on their expanded value).
- `(ppx_runtime_libraries (<library-names>))` is for when the library is a `ppx rewriter` or a `[@@deriving ...]` plugin, and has runtime dependencies. You need to specify these runtime dependencies here.
- `(virtual_deps (<opam-packages>)`. Sometimes opam packages enable a specific feature only if another package is installed. For instance, the case of `ctypes` will only install `ctypes.foreign` if the dummy `ctypes-foreign` package is installed. You can specify such virtual dependencies here, but you don't need to do so unless you use Dune to synthesize the `depends` and `depopts` sections of your opam file.
- `js_of_ocaml` sets options for JavaScript compilation, see *js_of_ocaml*.
- For `flags`, `ocamlc_flags`, and `ocamlopt_flags`, see *OCaml Flags*.
- `(library_flags (<flags>))` is a list of flags passed to `ocamlc` and `ocamlopt` when building the library archive files. You can use this to specify `-linkall`, for instance. `<flags>` is a list of strings supporting *Variables*.
- `(c_library_flags <flags>)` specifies the flags passed to the C compiler when constructing the library archive file for the C stubs. `<flags>` uses the *Ordered Set Language* and supports `(:include ...)` forms. When you write bindings for a C library named `bar`, you should typically write `-lbar` here, or whatever flags are necessary to link against this library.
- `(modules_without_implementation <modules>)` specifies a list of modules that have only a `.mli` or `.rei` but no `.ml` or `.re` file. Such modules are usually referred as *mli only modules*. They are not officially supported by the OCaml compiler; however, they are commonly used. Such modules must only define types. Since it isn't reasonably possible for Dune to check this is the case, Dune requires the user to explicitly list such modules to avoid surprises. Note that the `modules_without_implementation` field isn't merged in `modules`, which represents the total set of modules in a library. If a directory has more than one stanza, and thus a `modules` field must be specified, `<modules>` still needs to be added in `modules`.
- `(private_modules <modules>)` specifies a list of modules that will be marked as private. Private modules are inaccessible from outside the libraries they are defined in. Note that the `private_modules` field is not merged

in `modules`, which represents the total set of modules in a library. If a directory has more than one stanza and thus a `modules` field must be specified, `<modules>` still need to be added in `modules`.

- `(allow_overlapping_dependencies)` allows external dependencies to overlap with libraries that are present in the workspace.
- `(enabled_if <blang expression>)` conditionally disables a library. A disabled library cannot be built and will not be installed. The condition is specified using the *Boolean Language*, and the field allows for the `%{os_type}` variable, which is expanded to the type of OS being targeted by the current build. Its value is the same as the value of the `os_type` parameter in the output of `ocamlc -config`.
- `(inline_tests)` enables inline tests for this library. They can be configured through options using `(inline_tests <options>)`. See *Inline Tests* for a reference of corresponding options.
- `(root_module <module>)` this field instructs Dune to generate a module that will contain module aliases for every library specified in dependencies. This is useful whenever a library is shadowed by a local module. The library may then still be accessible via this root module
- `(ctypes <ctypes field>)` instructs Dune to use ctypes stubgen to process your type and function descriptions for binding system libraries, vendored libraries, or other foreign code. See *Stub Generation with Dune Ctypes* for a full reference. This field is available since the 3.0 version of the Dune language.
- `(empty_module_interface_if_absent)` causes the generation of empty interfaces for every module that does not have an interface file already. Useful when modules are used solely for their side-effects. This field is available since the 3.0 version of the Dune language.

Note that when binding C libraries, Dune doesn't provide special support for tools such as `pkg-config`; however, it integrates easily with *Configurator* by using `(c_flags (:include ...))` and `(c_library_flags (:include ...))`.

3.1.4 foreign_library

The `foreign_library` stanza describes archives of separately compiled foreign object files that can be packaged with an OCaml library or linked into an OCaml executable. See *Foreign Sources, Archives, and Objects* for further details and examples.

3.1.5 deprecated_library_name

The `deprecated_library_name` stanza enables redirecting an old deprecated name after a library has been renamed. It's syntax is as follows:

```
(deprecated_library_name
 (old_public_name <name>)
 (new_public_name <name>))
```

When a developer uses the old public name in a list of library dependencies, it will be transparently replaced by the new name. Note that it's not necessary for the new name to exist at definition time, as it is only resolved at the point where the old name is used.

The `old_public_name` can also be one of the names declared in the `deprecated_package_names` field of the package declaration in the `dune-project` file. In this case, the "old" library is understood to be a library whose name is not prefixed by the package name. Such a library cannot be defined in Dune, but other build systems allow it. This feature is meant to help migration from those systems.

3.1.6 generate_sites_module

New in version 2.8.

Dune proposes some facilities for dealing with *sites* in a program. The `generate_sites_module` stanza will generate code for looking up the correct locations of the sites' directories and for loading plugins. It works after installation with or without the relocation mode, inside Dune rules, and when using Dune executables. For promotion, it works only if the generated modules are solely in the executable (or library statically linked) promoted; generated modules in plugins won't work.

```
(generate_sites_module
 (module <name>)
 <facilities>)
```

The module's code is generated in the directory with the given name. The code is populated according to the requested facilities.

The available <facilities> are:

- `sourceroot` adds a value `val sourceroot: string option` in the generated module, which contains the value of `%{workspace_root}`, if the code has been built locally. It could be used to keep the tool's configuration file locally when executed with `dune exec` or after promotion. The value is `None` once it has been installed.
- `relocatable` adds a value `val relocatable: bool` in the generated module, which indicates if the binary has been installed in the relocatable mode.
- `(sites <package>)` adds a value `val <site>: string list` for each <site> of <package> in the submodule *Sites* of the generated module. The identifier <site> isn't capitalized.
- `(plugins (<package> <site>) ...)` adds a submodule <site> with the following signature `S` in the submodule `Plugins` of the generated module . The identifier <site> is capitalized.

```
module type S = sig
  val paths: string list
  (** return the locations of the directory containing the plugins *)

  val list: unit -> string list
  (** return the list of available plugins *)

  val load_all: unit -> unit
  (** load all the plugins and their dependencies *)

  val load: string -> unit
  (** load the specified plugin and its dependencies *)
end
```

The generated module is a dependency on the library `dune-site`, and if the facilities `(plugins ...)` are used, it is a dependency on the library `dune-site.plugins`. Those dependencies are not automatically added to the library or executable which use the module (cf. *Plugins and Dynamic Loading of Packages*).

3.1.7 test

The `test` stanza is the singular form of `tests`. The only difference is that it's of the form:

```
(test
 (name foo)
 <optional fields>)
```

The `name` field is singular, and the same optional fields are supported.

3.1.8 tests

The `tests` stanza allows one to easily define multiple tests. For example, we can define two tests at once with:

```
(tests
 (names mytest expect_test)
 <optional fields>)
```

This defines an executable named `mytest.exe` that will be executed as part of the `runtest` alias. If the directory also contains an `expect_test.expected` file, then `expect_test` will be used to define an expect test. That is, the test will be executed and its output will be compared to `expect_test.expected`.

The optional fields supported are a subset of the alias and executables fields. In particular, all fields except for `public_names` are supported from the *executables stanza*. Alias fields apart from `name` are allowed.

The `(enabled_if)` field has special semantics: when present, it only applies to running the tests. The test executable is always built by default. If you need to restrict building the test executable, use `(build_if)` instead.

By default, the test binaries are run without options. The `action` field can override the test binary invocation, i.e., if you're using Alcotest and wish to see all the test failures on the standard output. When running Dune `runtest` you can use the following stanza:

```
(tests
 (names mytest)
 (libraries alcotest mylib)
 (action (run %{test} -e)))
```

Starting from Dune 2.9, it's possible to automatically generate empty interface files for test executables. See *executables_implicit_empty_intf*.

3.1.9 Cram

`(cram ...)`

Configure Cram tests in the current directory (and subdirectories).

A single test may be configured by more than one `cram` stanza. In such cases, the values from all applicable `cram` stanzas are merged together to get the final values for all the fields.

See also:

Cram Tests

`(deps <dep-spec>)`

Specify the dependencies of the test.

When testing binaries, it's important to to specify a dependency on the binary for two reasons:

- Dune must know to re-run the test when a dependency changes
- The dependencies must be specified to guarantee that they're visible to the test when running it.

The following introduces a dependency on `foo.exe` on all Cram tests in this directory:

```
(cram
 (deps ../foo.exe))
```

See also:

Dependency Specification.

(applies_to <predicate-lang>)

Specify the scope of this `cram` stanza. By default it applies to all the Cram tests in the current directory. The special `:whole_subtree` value will apply the options to all tests in all subdirectories (recursively). This is useful to apply common options to an entire test suite.

The following will apply the stanza to all tests in this directory, except for `foo.t` and `bar.t`:

```
(cram
 (applies_to * \ foo bar)
 (deps ../foo.exe))
```

See also:

Predicate Language

(enabled_if <blang>)

Control whether the tests are enabled.

See also:

Boolean Language, *Variables*

(alias <name>)

Alias that can be used to run the test. In addition to the user alias, every test `foo.t` is attached to the *@runtest* alias and gets its own `@foo` alias to make it convenient to run individually.

(locks <lock-names>)

Specify that the tests must be run while holding the following locks.

See also:

Locks

(package <name>)

Attach the tests selected by this stanza to the specified package.

(runtest_alias <true|false>)

New in version 3.12.

When set to `false`, do not add the tests to the `runtest` alias. The default is to add every Cram test to `runtest`, but this is not always desired.

3.1.10 toplevel

The `toplevel` stanza allows one to define custom toplevels. Custom toplevels automatically load a set of specified libraries and are runnable like normal executables. Example:

```
(toplevel
 (name tt)
 (libraries str))
```

This will create a toplevel with the `str` library loaded. We may build and run this toplevel with:

```
$ dune exec ./tt.exe
```

`(preprocess (pps ...))` is the same as the `(preprocess (pps ...))` field of *library*. Currently, `action` and `future_syntax` are not supported in the toplevel.

3.1.11 documentation

Additional manual pages may be attached to packages using the `documentation` stanza. These `.mld` files must contain text in the same syntax as OCamldoc comments.

```
(documentation (<optional-fields>))
```

Where <optional-fields> are:

- `(package <name>)` defines the package this documentation should be attached to. If this is absent, Dune will try to infer it based on the location of the stanza.
- `(mld_files <arg>)`: the <arg> field follows the *Ordered Set Language*. This is a set of extensionless MLD file basenames attached to the package, where `:standard` refers to all the `.mld` files in the stanza's directory.

For more information, see *Generating Documentation*.

3.1.12 install

Dune supports installing packages on the system, i.e., copying freshly built artifacts from the workspace to the system. The `install` stanza takes three pieces of information:

- The list of files or directories to install
- The package to attach these files. This field is optional if your project contains a single package.
- The section in which the files will be installed

For instance:

```
(install
 (files hello.txt)
 (section share)
 (package mypackage))
```

Indicate that the file `hello.txt` in the current directory is to be installed in <prefix>/share/mypackage.

The following sections are available:

Section	Target	Remarks
lib	<prefix>/lib/<pkgname>/	
lib_root	<prefix>/lib/	
libexec	<prefix>/lib/<pkgname>/	executable bit is set
libexec_root	<prefix>/lib/	executable bit is set
bin	<prefix>/bin/	executable bit is set
sbin	<prefix>/sbin/	executable bit is set
toplevel	<prefix>/lib/toplevel/	
share	<prefix>/share/<pkgname>/	
share_root	<prefix>/share/	
etc	<prefix>/etc/<pkgname>/	
stublibs	<prefix>/lib/stublibs/	executable bit is set
doc	<prefix>/doc/<pkgname>/	
man	<prefix>/man/manX/	(see below)
misc	absolute destination	(see below)
(site (<package> <site>))	<site> directory of <package>	(see below)

Additional remarks:

- For man, the exact destination is inferred from the file extension. For example, foo.1 is installed as <prefix>/man/man1/foo.1.
- misc only works when using opam. In that case, the user will be prompted before installation. This mechanism is deprecated.
- In the case of (site), if the prefix isn't the same as the one used when installing <package>, <package> won't find the files.

Normally, Dune uses the file's basename to determine the file's name once installed; however, you can change that by using the form (<filename> as <destination>) in the files field. For instance, to install a file mylib.el as <prefix>/emacs/site-lisp/mylib.el, you must write the following:

```
(install
 (section share_root)
 (files   (mylib.el as emacs/site-lisp/mylib.el)))
```

The mode of installed files is fully determined by the section they are installed in. If the section above is documented as with the executable bit set, they are installed with mode 0o755 (rwxr-xr-x); otherwise they are installed with mode 0o644 (rw-r--r--).

Note that all files in the install stanza must be specified by relative paths only. It is an error to specify files by absolute paths.

Also note that as of dune-lang 3.11 (i.e., (lang dune 3.11) in dune-project) it is deprecated to use the as keyword to specify a destination beginning with ... Dune intends for files associated with a package to only be installed under specific directories in the file system implied by the installation section (e.g., share, bin, doc, etc.) and the package name. Starting destination paths with .. allows packages to install files to arbitrary locations on the file system. In 3.11, this behaviour is still supported (as some projects may depend on it) but will generate a warning and will be removed in a future version of Dune.

Including Files in the Install Stanza

You can include external files from the `files` and `dirs` fields of the install stanza:

```
(install
 (files (include foo.sexp))
 (section share))
```

Here the file `foo.sexp` must contain a single S-expression list, whose elements will be included in the list of files or directories to install. That is, elements may be of the form:

- `<filename>`
- `(<filename> as <destination>)`
- `(include <filename>)`

Included files may be generated by rules. Here is an example of a rule which generates a file by listing all the files in a subdirectory `resources`:

```
(rule
 (deps (source_tree resources))
 (action
  (with-stdout-to foo.sexp
  (system "echo '(' resources/* ')'"))))
```

Globs in the Install Stanza

You can use globs to specify files to install by using the terms `(glob_files <glob>)` and `(glob_files_rec <glob>)` inside the `files` field of the install stanza (but not inside the `dirs` field). See the *glob* for details of the glob syntax. The `(glob_files <glob>)` term will expand its argument within a single directory, whereas the `(glob_files_rec <glob>)` term will recursively expand its argument within all subdirectories.

For example:

```
(install
 (files
  (glob_files style/*.css)
  (glob_files_rec content/*.html))
 (section share))
```

This example will install:

- All files matching `*.css` in the `style` directory.
- All files matching `*.html` in the `content` directory, or any of its descendant subdirectories.

Note that the paths to files are preserved after installation. Suppose the source directory contained the files `style/foo.css` and `content/bar/baz.html`. The example above will place these files in `share/<package>/style/foo.css` and `share/<package>/content/bar/baz.html` respectively where `<package>` is the name of the package (ie. `dune-project` would contain `(package (name <package>))`).

The `with_prefix` keyword can be used to change the destination path of files matched by a glob, similar to the `as` keyword in the `(files ...)` field. `with_prefix` changes the prefix of a path before the component matched by the `*` to some new value. For example:

```
(install
 (files
  (glob_files (style/*.css with_prefix web/stylesheets))
  (glob_files_rec (content/*.html with_prefix web/documents)))
 (section share))
```

Continuing the example above, this would result in the source file at `style/foo.css` being installed to `share/<package>/web/stylesheets/foo.css` and `content/bar/baz.html` being installed to `share/<package>/web/documents/bar/baz.html`. Note in the latter case `with_prefix` only replaced the `content` component of the path and not the `bar` component since since it replaces the prefix of the glob - not the prefix of paths matching the glob.

Installing Globs from Parent Directories

The default treatment of paths in globs creates a complication where referring to globs in a parent directory such as `(glob_files ../*.txt)` would attempt to install the matched files outside the designated install directory. For example writing:

```
(install
 (files (glob_files ../*.txt))
 (section share))
```

…would cause Dune to attempt to install the matching files to `share/<package>/../`, ie. `share` where `<package>` is the name of the package (i.e., `dune-project` would contain `(package (name <package>))`). This is probably not what the user intends, and installing files to relative paths beginning with `..` is deprecated from version 3.11 of Dune and will become an error in a future version.

The solution is to use `with_prefix` to replace the `..` with some other path. For example:

```
(install
 (files (glob_files (../*.txt with_prefix .)))
 (section share))
```

…would install the matched files to `share/<package>/` instead.

Handling of the .exe Extension on Windows

Under Microsoft Windows, executables must be suffixed with `.exe`. Dune tries to ensure that executables are always installed with this extension on Windows.

More precisely, when installing a file via an `(install ...)` stanza, Dune implicitly adds the `.exe` extension to the destination, if the source file has extension `.exe` or `.bc` and if it's not already present

Installing Source Directories

To install entire source directories, the `source_tree` field can be used:

```
(install
 (section doc)
 (source_trees manual))
```

This example results in the contents of the `manual` directory being installed under `<prefix>/doc/<package>/manual/`.

As with `(files ...)` the destination can be changed with the `as` keyword. For example if you want to install all the files in the `manual` directory directly into `<prefix>/doc/<package>/` you can write:

```
(install
 (section doc)
 (source_trees (manual as .)))
```

It's also possible to specify multiple directories:

```
(install
 (section doc)
 (source_trees manual examples))
```

This would result in the local directories `manual` and `examples` being installed to `<prefix>/doc/<package>/manual/` and `<prefix>/doc/<package>/examples/` respectively.

Unlike with `(files ...)` it is an error to begin the destination (the right-hand side of `as`) with `...` (This is because support for installing source directories was added to Dune after destinations beginning with `..` were deprecated.)

3.1.13 plugin

New in version 2.8.

Plugins are a way to load OCaml libraries at runtime. The `plugin` stanza allows you to declare the plugin's name, which *sites* should be present and which libraries it will load.

```
(plugin
 (name <name>)
 (libraries <libraries>)
 (site (<package> <site name>))
 (<optional-fields>))
```

`<optional-fields>` are:

- `(package <package>)` if there is more than one package defined in the current scope, this specifies which package the plugin will install. A plugin can be installed by one package in the site of another package.

- `(optional)` will not declare the plugin if the libraries are not available.

The loading of the plugin is done using the facilities generated by *generate_sites_module*.

3.1.14 rule

The `rule` stanza is used to create custom user rules. It tells Dune how to generate a specific set of files from a specific set of dependencies.

The syntax is as follows:

```
(rule
 (action <action>)
 <optional-fields>)
```

`<action>` is what you run to produce the targets from the dependencies. See *Actions* for more details.

`<optional-fields>` are:

- `(target <filename>)` or `(targets <filenames>)` ``<filenames>`` is a list of filenames (if defined with `targets`) or exactly one filename (if defined with `target`). Dune needs to statically know targets of each rule. `(targets)` can be omitted if it can be inferred from the action. See *inferred rules*.
- `(deps <deps-conf list>)` specifies the dependencies of the rule. See *Dependency Specification* for more details.
- `(mode <mode>)` specifies how to handle the targets. See *modes* for details.
- `(fallback)` is deprecated and is the same as `(mode fallback)`.
- `(locks (<lock-names>))` specifies that the action must be run while holding the following locks. See *Locks* for more details.
- `(alias <alias-name>)` specifies this rule's alias. Building this alias means building the targets of this rule.
- `(aliases <alias-name list>)` specifies many aliases for this rule.
- `(package <package>)` specifies this rule's package. This rule will be unavailable when installing other packages in release mode.
- `(enabled_if <blang expression>)` specifies the Boolean condition that must be true for the rule to be considered. The condition is specified using the *Boolean Language*, and the field allows for *Variables* to appear in the expressions.

Please note: contrary to makefiles or other build systems, user rules currently don't support patterns, such as a rule to produce `%.y` from `%.x` for any given `%`. This might be supported in the future.

Modes

By default, a rule's target must not exist in the source tree because Dune will error out when this is the case; however, it's possible to change this behavior using the `mode` field. The following modes are available:

- `standard` - the standard mode.
- `fallback` - in this mode, when the targets are already present in the source tree, Dune will ignore the rule. It's an error if only a subset of the targets are present in the tree. Fallback rules are commonly used to generate default configuration files that may be generated by a configure script.
- `promote` or `(promote <options>)` - in this mode, the files in the source tree will be ignored. Once the rule has been executed, the targets will be copied back to the source tree. The following options are available:

- `(until-clean)` means that `dune clean` will remove the promoted files from the source tree.
- `(into <dir>)` means that the files are promoted in `<dir>` instead of the current directory. This feature has been available since Dune 1.8.
- `(only <predicate>)` means that only a subset of the targets should be promoted. The argument is similar to the argument of *subdir*, specified using the *Predicate Language*. This feature has been available since Dune 1.10.

There are two use cases for `promote` rules. The first one is when the generated code is easier to review than the generator, so it's easier to commit the generated code and review it. The second is to cut down dependencies during releases. By passing `--ignore-promoted-rules` to Dune, rules with `(mode promote)` will be ignored, and the source files will be used instead. The `-p/--for-release-of-packages` flag implies `--ignore-promote-rules`. However, rules that promote only a subset of their targets via `(only ...)` are never ignored.

Inferred Rules

When using the action DSL (see *Actions*), the dependencies and targets are usually obvious.

For instance:

```
(rule
 (target b)
 (deps   a)
 (action (copy %{deps} %{target})))
```

In this example, the dependencies and targets are obvious by inspecting the action. When this is the case, you can use the following shorter syntax and have Dune infer dependencies and targets for you:

```
(rule <action>)
```

For instance:

```
(rule (copy a b))
```

Note that in Dune, targets must always be known statically. For instance, this `(rule ...)` stanza is rejected by Dune:

```
(rule (copy a b.%{read:file}))
```

Directory targets

Note that at this time, Dune officially only supports user rules with targets in the current directory. However, starting from Dune 3.0, we provide an experimental support for *directory targets*, where an action can produce a whole tree of build artifacts. To specify a directory target, you can use the `(dir <dirname>)` syntax. For example, the following stanza describes a rule with a file target `foo` and a directory target `bar`.

```
(rule
 (targets foo (dir bar))
 (action  <action>))
```

To enable this experimental feature, add `(using directory-targets 0.1)` to your `dune-project` file. However note that currently rules with a directory target are always rebuilt. We are working on fixing this performance bug.

3.1.15 alias

The `alias` stanza adds dependencies to an alias or specifies an action to run to construct the alias.

The syntax is as follows:

```
(alias
 (name    <alias-name>)
 (deps    <deps-conf list>)
 <optional-fields>)
```

<name> is an alias name such as `runtest`.

<deps-conf list> specifies the dependencies of the alias. See *Dependency Specification* for more details.

<optional-fields> are:

- <action>, an action for constructing the alias. See *Actions* for more details. Note that this is removed in Dune 2.0, so users must port their code to use the `rule` stanza with the `alias` field instead.
- `(package <name>)` indicates that this alias stanza is part of package <name> and should be filtered out if <name> is filtered out from the command line, either with `--only-packages <pkgs>` or `-p <pkgs>`.
- `(locks (<lock-names>))` specifies that the action must be run while holding the following locks. See *Locks* for more details.
- `(enabled_if <blang expression>)` specifies the Boolean condition that must be true for the tests to run. The condition is specified using the *Boolean Language*, and the field allows for *Variables* to appear in the expressions.

The typical use of the `alias` stanza is to define tests:

```
(rule
 (alias   runtest)
 (action (run %{exe:my-test-program.exe} blah)))
```

See the section about *Running Tests* for details.

Please note: if your project contains several packages, and you run the tests from the opam file using a `build-test` field, all your `runtest` alias stanzas should have a `(package ...)` field in order to partition the set of tests.

3.1.16 copy_files

The `copy_files` and `copy_files#` stanzas specify that files from another directory could be copied to the current directory, if needed.

The syntax is as follows:

```
(copy_files
 <optional-fields>
 (files <glob>))
```

<glob> represents the set of files to copy. See the *glob* for details.

<optional-fields> are:

- `(alias <alias-name>)` specifies an alias to which to attach the targets.
- `(mode <mode>)` specifies how to handle the targets. See *Modes* for details.

- `(enabled_if <blang expression>)` conditionally disables this stanza. The condition is specified using the *Boolean Language*.
- `(only_sources <blang expression>)` specifies that the glob in `files` gets applied over the source tree, and not the build tree.

The short form:

```
(copy_files <glob>)
```

is equivalent to:

```
(copy_files (files <glob>))
```

The difference between `copy_files` and `copy_files#` is the same as the difference between the `copy` and `copy#` actions. See *Actions* section for more details.

3.1.17 include

The `include` stanza allows including the contents of another file in the current `dune` file. Currently, the included file cannot be generated and must be present in the source tree. This feature is intended for use in conjunction with promotion, when parts of a `dune` file are to be generated.

For instance:

```
(include dune.inc)

(rule (with-stdout-to dune.inc.gen (run ./gen-dune.exe)))

(rule
 (alias  runtest)
 (action (diff dune.inc dune.inc.gen)))
```

With this `dune` file, running Dune as follows will replace the `dune.inc` file in the source tree by the generated one:

```
$ dune build @runtest --auto-promote
```

3.1.18 dynamic_include

The `dynamic_include` stanza allows including the contents of another file in the current dune file like the `include` stanza. However, the `dynamic_include` stanza allows the included file to be the target of a rule and disallows generating some stanzas.

For instance:

```
(subdir b
 (dynamic_include ../a/foo.inc))
(subdir a
 (rule
  (write-file
   foo.inc
   "(rule (write-file file bar))")))
```

In the example above, the dynamic rule loading and generation are split into different directories to avoid rule loading cycles as rules are loaded per directory.

The following stanzas cannot be dynamically generated:

- Libraries, coq theories, library redirects
- Public executables or install section with the `bin` section
- Plugin stanzas

3.1.19 env

The `env` stanza allows one to modify the environment. The syntax is as follows:

```
(env
 (<profile1> <settings1>)
 (<profile2> <settings2>)
 ...
 (<profilen> <settingsn>))
```

The first form `(<profile> <settings>)` that corresponds to the selected build profile will be used to modify the environment in this directory. You can use `_` to match any build profile.

Fields supported in `<settings>` are:

- any OCaml flags field. See *OCaml Flags* for more details.
- `(link_flags <flags>)` specifies flags to OCaml when linking an executable. See *executables stanza*.
- `(c_flags <flags>)` and `(cxx_flags <flags>)` specify compilation flags for C and C++ stubs, respectively. See *library* for more details.
- `(env-vars (<var1> <val1>) .. (<varN> <valN>))` will add the corresponding variables to the environment where the build commands are executed and are used by `dune exec`.
- `(menhir_flags <flags>))` specifies flags for Menhir stanzas. This flag was replaced by the `(menhir)` field (see below) starting in version 3.0 of the Menhir extension.
- `(menhir (flags <flags) (explain <blang expression>))` specifies the Menhir settings. See *menhir* for more details. This field was introduced in version 3.0 of the Menhir extension.
- `(js_of_ocaml (flags <flags>)(build_runtime <flags>)(link_flags <flags>))` specifies `js_of_ocaml` flags. See *js_of_ocaml* for more details.
- `(js_of_ocaml (compilation_mode <mode>))` controls whether to use separate compilation or not where `<mode>` is either `whole_program` or `separate`.
- `(js_of_ocaml (runtest_alias <alias-name>))` specifies the alias under which *Inline Tests* and tests (*tests*) run for the *js* mode.
- `(binaries <binaries>)`, where `<binaries>` is a list of entries of the form `(<filepath> as <name>)`. `(<filepath> as <name>)` makes the binary `<filepath>` available in the command search as just `<name>`. For instance, in a `(run <name> ...)` action, `<name>` will resolve to this file path. You can also write just the file path, in which case the name will be inferred from the basename of `<filepath>` by dropping the `.exe` suffix, if it exists. For example, `(binaries bin/foo.exe (bin/main.exe as bar))` would add the commands `foo` and `bar` to the search path.
- `(inline_tests <state>)`, where `<state>` is either `enabled`, `disabled`, or `ignored`. This field has been available since Dune 1.11. It controls the variable's value `%{inline_tests}`, which is read by the inline test framework. The default value is `disabled` for the `release` profile and `enabled` otherwise.
- `(odoc <fields>)` allows passing options to `odoc`. See *Passing Options to odoc* for more details.
- `(coq <coq_fields>)` allow passing options to Coq. See *Coq Environment Fields* for more details.

- `(formatting <settings>)` allows the user to set auto-formatting in the current directory subtree (see *formatting*).
- `(bin_annot <bool>)` allows the user to specify whether to generate **.cmt* and **.cmti* in the current directory subtree.

3.1.20 dirs

New in version 1.6.

The `dirs` stanza allows specifying the subdirectories Dune will include in a build. The syntax is based on Dune's *Predicate Language* and allows the following operations:

- The special value `:standard` which refers to the default set of used directories. These are the directories that don't start with `.` or `_`.
- Set operations. Differences are expressed with backslash: `* \ bar`; unions are done by listing multiple items.
- Sets can be defined using globs.

Examples:

```
(dirs *) ;; include all directories
(dirs :standard \ ocaml) ;; include all dirs except ocaml
(dirs :standard \ test* foo*) ;; exclude all dirs that start with test or foo
```

Dune will not scan a directory that isn't included in this stanza. Any contained `dune` (or other special) files won't be interpreted either and will be treated as raw data. It is however possible to depend on files inside ignored subdirectories.

3.1.21 data_only_dirs

New in version 1.6.

Dune allows the user to treat directories as *data only*. `dune` files in these directories won't be evaluated for their rules, but the contents of these directories will still be usable as dependencies for other rules.

The syntax is the same as for the `dirs` stanza except that `:standard` is empty by default.

Example:

```
;; dune files in fixtures_* dirs are ignored
(data_only_dirs fixtures_*)
```

3.1.22 ignored_subdirs

Deprecated since version 1.6.

One may also specify *data only* directories using the `ignored_subdirs` stanza, meaning it's the same as `data_only_dirs`, but the syntax isn't as flexible and only accepts a list of directory names. It's advised to switch to the new `data_only_dirs` stanza.

Example:

```
(ignored_subdirs (<sub-dir1> <sub-dir2> ...))
```

All of the specified `<sub-dirn>` will be ignored by Dune. Note that users should rely on the `dirs` stanza along with the appropriate set operations instead of this stanza. For example:

```
(dirs :standard \ <sub-dir1> <sub-dir2> ...)
```

3.1.23 include_subdirs

The `include_subdirs` stanza is used to control how Dune considers subdirectories of the current directory. The syntax is as follows:

```
(include_subdirs <mode>)
```

Where <mode> maybe be one of:

- `no`, the default
- `unqualified`
- `qualified`

When the `include_subdirs` stanza isn't present or <mode> is `no`, Dune considers subdirectories independent. When <mode> is `unqualified`, Dune will assume that the current directory's subdirectories are part of the same group of directories. In particular, Dune will simultaneously scan all these directories when looking for OCaml/Reason files. This allows you to split a library between several directories. `unqualified` means that modules in subdirectories are seen as if they were all in the same directory. In particular, you cannot have two modules with the same name in two different directories. When <mode> is `qualified`, each subdirectory's files will be grouped into submodules of the library module, mirroring the directory structure.

Note that subdirectories are included recursively; however, the recursion will stop when encountering a subdirectory that contains another `include_subdirs` stanza. Additionally, it's not allowed for a subdirectory of a directory with `(include_subdirs <x>)` where <x> is not `no` to contain one of the following stanzas:

- `library`
- `executable(s)`
- `test(s)`

3.1.24 vendored_dirs

New in version 1.11.

Dune supports vendoring other Dune-based projects natively, since simply copying a project into a subdirectory of your own project will work. Simply doing that has a few limitations though. You can workaround those by explicitly marking such directories as containing vendored code.

Example:

```
(vendored_dirs vendor)
```

Dune will not resolve aliases in vendored directories. By default, it won't build all installable targets, run the tests, format, or lint the code located in such a directory while still building your project's dependencies. Libraries and executables in vendored directories will also be built with a `-w -a` flag to suppress all warnings and prevent pollution of your build output.

3.1.25 subdir

The `subdir` stanza can be used to evaluate stanzas in subdirectories. This is useful for generated files or to override stanzas in vendored directories without editing vendored `dune` files.

In this example, a `bar` target is created in the `foo` directory, and a bar target will be created in `a/b/bar`:

```
(subdir foo (rule (with-stdout-to bar (echo baz))))
(subdir a/b (rule (with-stdout-to bar (echo baz))))
```

3.1.26 cinaps

A `cinaps` stanza is available to support the `cinaps` tool. See the cinaps website for more details.

3.1.27 coq.theory

See the documentation on the *coq.theory*, *coq.extraction*, *coq.pp*, and related stanzas.

3.1.28 mdx

New in version 2.4.

MDX is a tool that helps you keep your markdown documentation up-to-date by checking that its code examples are correct. When setting an MDX stanza, the MDX checks are automatically attached to the `runtest` alias of the stanza's directory.

See MDX's repository for more details.

You can define an MDX stanza to specify which files you want checked.

Note that this feature is still experimental and needs to be enabled in your `dune-project` with the following `using` stanza:

```
(using mdx 0.4)
```

Note: Version `0.2` of the stanza requires mdx `1.9.0`. Version `0.4` of the stanza requires mdx `2.3.0`.

The syntax is as follows:

```
(mdx <optional-fields>)
```

Where <optional-fields> are:

- `(files <globs>)` are the files that you want MDX to check, described as a list of globs (see the *Glob language specification*). It defaults to `*.md *.mld` as of version `0.4` of the stanza and `*.md` before.
- `(deps <deps-conf list>)` to specify the dependencies of your documentation code blocks. See *Dependency Specification* for more details.
- `(preludes <files>)` are the prelude files you want to pass to MDX. See MDX's documentation for more details on preludes.
- `(libraries <libraries>)` are libraries that should be statically linked in the MDX test executable.
- `(enabled_if <blang expression>)` is the same as the corresponding field of *library*.

- `(package <package>)` specifies which package to attach this stanza to (similarly to when `(package)` is attached to a `(rule)` stanza). When `-p` is passed, `(mdx)` stanzas with another package will be ignored. Note that this feature is completely separate from `(packages)`, which specifies some dependencies.
- `(locks <lock-names>)` specifies that the action of running the tests holds the specified locks. See *Locks* for more details.

Upgrading from Version 0.1

- The 0.2 version of the stanza requires at least MDX 1.9.0. If you encounter an error such as, `ocaml-mdx: unknown command `dune-gen'`, then you should upgrade MDX.
- The field `(packages <packages>)` is deprecated in version 0.2. You can use package items in the generic `deps` field instead: `(deps (package <package>) ... (package <package>))`
- Use the new `libraries` field to directly link libraries in the test executable and remove the need for `#require` directives in your documentation code blocks.

3.1.29 menhir

A `menhir` stanza is available to support the Menhir parser generator.

To use Menhir in a Dune project, the language version should be selected in the `dune-project` file. For example:

```
(using menhir 2.0)
```

This will enable support for Menhir stanzas in the current project. If the language version is absent, Dune will automatically add this line with the latest Menhir version once a Menhir stanza is used anywhere.

The basic form for defining menhir-git parsers (analogous to *ocamlyacc*) is:

```
(menhir
 (modules <parser1> <parser2> ...)
 <optional-fields>)
```

<optional-fields> are:

- `(merge_into <base_name>)` is used to define modular parsers. This correspond to the `--base` command line option of `menhir`. With this option, a single parser named `base_name` is generated.
- `(flags <option1> <option2> ...)` is used to pass extra flags to Menhir.
- `(infer <bool>)` is used to enable Menhir with type inference. This option is enabled by default with Menhir language 2.0.

Menhir supports writing the grammar and automation to the `.cmly` file. Therefore, if this is flag is passed to Menhir, Dune will know to introduce a `.cmly` target for the module.

- `(explain <blang expression>)` is used to control the generation of the `.conflicts` file explaining conflicts found while generating the parser. The condition is specified using the *Boolean Language*. This field was introduced in version 3.0 of the Menhir extension.

Note that starting in version 3.0 of the Menhir extension, the `.conflicts` file is generated by default. If this is not desired, it needs to be disabled explicitly by using the `(explain)` field.

3.1.30 ocamllex

`(ocamllex <names>)` is essentially a shorthand for:

```
(rule
 (target <name>.ml)
 (deps   <name>.mll)
 (action (chdir %{workspace_root}
          (run %{bin:ocamllex} -q -o %{target} %{deps}))))
```

To use a different rule mode, use the long form:

```
(ocamllex
 (modules <names>)
 (mode    <mode>))
```

3.1.31 ocamlyacc

`(ocamlyacc <names>)` is essentially a shorthand for:

```
(rule
 (targets <name>.ml <name>.mli)
 (deps    <name>.mly)
 (action  (chdir %{workspace_root}
           (run %{bin:ocamlyacc} %{deps}))))
```

To use a different rule mode, use the long form:

```
(ocamlyacc
 (modules <names>)
 (mode    <mode>))
```

3.1.32 jbuild_version

Deprecated. This *jbuild_version* stanza is no longer used and will be removed in the future.

3.2 dune-project

These files are used to mark the root of projects as well as define project-wide parameters. The first line of `dune-project` must be a `lang` stanza with no extra whitespace or comments. The `lang` stanza controls the names and contents of all configuration files read by Dune and looks like:

```
(lang dune 3.14)
```

Additionally, they can contains the following stanzas.

3.2.1 accept_alternative_dune_file_name

(accept_alternative_dune_file_name ...)

New in version 3.0.

Specify that the alternative filename `dune-file` is accepted in addition to `dune`.

This may be useful to avoid problems with `dune` files that have the executable permission in a directory in the `PATH`, which can unwittingly happen on Windows.

Note that `dune` continues to be accepted even after enabling this option, but if a file named `dune-file` is found in a directory, it will take precedence over `dune`.

3.2.2 cram

(cram <status>)

Define whether Cram-style tests are enabled for the project.

<*status*> can be either `enable` or `disable`. The default is `enable` starting from the language version 3.0.

See also:

Cram Tests

3.2.3 dialect

(dialect ...)

Declare a new *dialect*.

(name <name>)

The name of the dialect being defined. It must be unique in a given project.

This field is required.

(implementation ...)

Details related to the implementation files (corresponding to **.ml*).

Changed in version 3.9: This field is made optional.

(extension <string>)

Specify the file extension used for this dialect.

The extension string must not start with a period and be unique in a given project (so that a given extension can be mapped back to a corresponding dialect). In Dune 3.9 and later, the extension string may contain periods (e.g., *cppo.ml*).

This field is required.

(preprocess <action>)

Run <*action*> to produce a valid OCaml abstract syntax tree.

This action is expected to read the file given in the variable named `%{input-file}` and output a *binary* abstract syntax tree on its standard output.

If the field is not present, it is assumed that the corresponding source code is already valid OCaml code and can be passed to the OCaml compiler as-is.

See also:

Preprocessing With Actions

`(format <action>)`

Run *<action>* to format source code for this dialect.

The action is expected to read the file given in the variable named `%{input-file}` and output the formatted source code on its standard output.

If the field is not present, the behavior depends on the presence of `(preprocess)`: if it is also not present (that is, the dialect consists of valid OCaml code), then the dialect will be formatted as any other OCaml code. Otherwise no special formatting will be done.

See also:

How to Set up Automatic Formatting

`(interface ...)`

Details related to the interface files (corresponding to **.mli*).

This field supports the same sub-fields as `implementation`.

Changed in version 3.9: This field is made optional.

`(merlin_reader <program> <args>...)`

Configure Merlin to use *<program> <args>...* as READER. Merlin's READER is a mechanism to extend Merlin to support OCaml dialects by providing a program that transforms a dialect AST into an OCaml AST.

See also:

merlin/src/extend/extend_protocol.ml for the protocol specification.

This field is optional.

New in version 3.16.

3.2.4 Default dialects

Dune ships with two dialects pre-configured and enabled:

- `ocaml` for the default OCaml syntax which consumes *.ml* and *.mli* files and uses `ocamlformat` for formatting.
- `reason` for the Reason syntax and enabled in *.re*/*.rei* files. `refmt` is used for formatting.

A third dialect, `rescript`, is added when Melange support (see *JavaScript Compilation With Melange*) is enabled in the project.

3.2.5 executables_implicit_empty_intf

`(executables_implicit_empty_intf ...)`

New in version 2.9.

Automatically generate empty interface files for executables and tests that do not already have them.

By default, executables defined via `(executables(s) ...)` or `(test(s) ...)` stanzas are compiled with the interface file provided (e.g., `.mli` or `rei`). Since these modules cannot be used as library dependencies, it is common to give them empty interface files to strengthen the compiler's ability to detect unused values in these modules.

This option, when enabled, will generate an empty **.mli* file.

Example:

```
(executables_implicit_empty_intf true)
```

This option is enabled by default starting with Dune lang 3.0.

3.2.6 expand_aliases_in_sandbox

(expand_aliases_in_sandbox ...)

When a sandboxed action depends on an alias, copy the expansion of the alias inside the sandbox. For instance, in the following example:

```
(alias
 (name foo)
 (deps ../x))

(cram
 (deps (alias foo)))
```

File *x* will be visible inside the Cram test if and only if this option is enabled. This option is a better default in general; however, it currently causes Cram tests to run noticeably slower. So it is disabled by default until the performance issue with Cram test is fixed.

3.2.7 explicit_js_mode

(explicit_js_mode ...)

Do not implicitly add `js` to the `(modes ...)` field of executables.

In projects that use dune lang 1.x, JavaScript targets are defined for every bytecode executable. This is not very precise and does not interact well with the *@all* alias.

It is possible to opt out of this behavior by using:

```
(explicit_js_mode)
```

When this is enabled, an explicit `js` mode needs to be added to the `(modes ...)` field of executables in order to trigger the JavaScript compilation. Explicit JS targets declared like this will be attached to the *@all* alias.

Starting with Dune 2.0, this behavior is the default, and there is no way to disable it.

3.2.8 formatting

(formatting ...)

New in version 2.0.

Control automatic formatting. Several forms are accepted:

- To disable automatic formatting completely (equivalent to the behavior in language 1.x):

```
(formatting disabled)
```

- To restrict the languages that are considered for formatting:

```
(formatting
 (enabled_for <languages>))
```

The list of <*languages*> can be either `dune` (formatting of `dune` files) or a *dialect* name.

See also:

How to Set up Automatic Formatting

3.2.9 generate_opam_files

`(generate_opam_files ...)`

Use metadata specified in the `dune-project` file to generate `.opam` files.

To enable this integration, add the following field to the `dune-project` file:

```
(generate_opam_files)
```

See also:

How to Generate Opam Files from dune-project

Dune uses the following global fields to set the metadata for all packages defined in the project:

`(license <strings>)`

Specify the license of the project, ideally as an identifier from the SPDX License List.

Example:

```
(license MIT)
```

Multiple licenses may be specified.

`(authors <strings>)`

Specify authors.

Example:

```
(authors
 "Jane Doe <jane.doe@example.com>"
 "John Doe <john.doe@example.com>")
```

`(maintainers <strings>)`

Specify maintainers.

Example:

```
(maintainers
 "Jane Doe <jane.doe@example.com>"
 "John Doe <john.doe@example.com>")
```

`(source ...)`

Specify where the source for the package can be found.

It can be specified as `(uri <uri>)` or using shortcuts for some hosting services:

Service	Syntax
Github	`(github user/repo)`
Bitbucket	`(bitbucket user/repo)`
Gitlab	`(gitlab user/repo)`
Sourcehut	`(sourcehut user/repo)`

Examples:

```
(source
 (github ocaml/dune))
```

```
(source
 (uri https://dev.example.com/project.git))
```

(bug_reports <url>)

Where bugs should be reported.

If a hosting service is used in `(source)`, a default value is provided.

Example:

```
(bug_reports https://dev.example.com/project/issues)
```

(homepage <url>)

The homepage of the project.

If a hosting service is used in `(source)`, a default value is provided.

Example:

```
(bug_reports https://example.com/)
```

(documentation <url>)

Where the documentation is hosted.

With these fields, every time one calls Dune to execute some rules (either via `dune build`, `dune runtest`, or something else), the opam files get generated.

Some or all of these fields may be overridden for each package of the project, see *package*.

3.2.10 implicit_transitive_deps

(implicit_transitive_deps ...)

Control whether transitive dependencies are made implicitly visible.

By default, Dune allows transitive dependencies of dependencies used when compiling OCaml. However, this can be disabled by specifying:

```
(implicit_transitive_deps false)
```

Then all dependencies directly used by a library or an executable must be added in the `libraries` field.

We recommend users experiment with this mode and report any problems.

Note that you must use `threads.posix` instead of `threads` when using this mode. This isn't an important limitation, as `threads.vm` is deprecated anyway.

In some situations, it can be desirable to selectively preserve the behavior of transitive dependencies' availability a library's users. For example, if we define a library `foo_more` that extends `foo`, we might want `foo_more` users to immediately have `foo` available as well. To do this, we must define the dependency on `foo` as re-exported:

```
(library
 (name foo_more)
 (libraries (re_export foo)))
```

3.2.11 map_workspace_root

(map_workspace_root <bool>)

Control references to the file system locations where the project has been built.

- with `(map_workspace_root true)`, dune rewrites references to the workspace root to `/workspace_root`. Note that when this mapping is enabled, the debug information produced by the bytecode compiler is incorrect, as the location information is lost.
- with `(map_workspace_root false)`, the references are not rewritten.

The default is `(map_workspace_root true)`.

New in version 3.0: Initial version with the mapping always enabled.

Changed in version 3.7: Add a way to disable the mapping.

3.2.12 name

(name <string>)

Set the name of the project.

It is used by *dune subst* and error messages.

3.2.13 opam_file_location

(opam_file_location <location>)

New in version 3.8.

Configure where generated `.opam` files are located. <*location*> can be one of the following:

- `relative_to_project`: the `.opam` files are generated in the project root directory. This is the default.
- `inside_opam_directory`: the `.opam` files are generated in a directory named `opam` in the project root directory.

See also:

How to Generate Opam Files from dune-project

3.2.14 package

(package ...)

Define package-specific metadata.

(name <string>)

The name of the package.

This must be specified.

(synopsis <string>)

A short package description.

(description <string>)

A longer package description.

(depends <dep-specification>)

Package dependencies, as *dep_specification*.

(conflicts <dep-specification>)

Package conflicts, as *dep_specification*.

(depopts <dep-specification>)

Optional package dependencies, as *dep_specification*.

(tags <tags>)

A list of tags.

(deprecated_package_names <name list>)

A list of names that can be used with the *deprecated_library_name* stanza to migrate legacy libraries from other build systems that do not follow Dune's convention of prefixing the library's public name with the package name.

(license ...)

New in version 2.0.

The same as (and takes precedences over) the corresponding global field.

(authors ...)

New in version 2.0.

The same as (and takes precedences over) the corresponding global field.

(maintainers ...)

New in version 2.0.

The same as (and takes precedences over) the corresponding global field.

(source ...)

New in version 2.0.

The same as (and takes precedences over) the corresponding global field.

(bug_reports ...)

New in version 2.0.

The same as (and takes precedences over) the corresponding global field.

(homepage ...)

New in version 2.0.

The same as (and takes precedences over) the corresponding global field.

(documentation ...)

New in version 2.0.

The same as (and takes precedences over) the corresponding global field.

(sites ...)

Define a site.

`(sites (<section> <name>) ...)` defines a site named `<name>` in the section `<section>`.

Adding libraries to different packages is done via the `public_name` and `package` fields. See *library* section for details.

The list of dependencies *dep_specification* is modelled after opam's own language. The syntax is a list of the following elements:

```
op                ::=   '=' | '<' | '>' | '<>' | '>=' | '<='
filter            ::=   :dev | :build | :with-test | :with-doc | :post
constr            ::=   (<op> <version>)
logop             ::=   or | and
dep               ::=   <name>
                        (<name> <filter>)
                        (<name> <constr>)
                        (<name> (<logop> (<filter> | <constr>))*)
dep_specification ::=   <dep>+
```

Filters will expand to any opam variable name if prefixed by `:`, not just the ones listed in *`filter`*. This also applies to version numbers. For example, to generate `depends: [ pkg { = version } ]`, use `(depends (pkg (= :version)))`.

Note that the use of a `using` stanza (see *using*) doesn't automatically add the associated library or tool as a dependency. They have to be added explicitly.

3.2.15 subst

`(subst <bool>)`

New in version 3.0.

Control whether *dune subst* is enabled for this project.

- `(subst disabled)`, means that any call of `dune subst` in this project is forbidden and will result in an error. This line will be omitted from the build instructions when generating opam files.
- `(subst enabled)` allows substitutions explicitly. This is the default.

3.2.16 use_standard_c_and_cxx_flags

`(use_standard_c_and_cxx_flags ...)`

New in version 2.8.

Control how flags coming from `ocamlc -config` are passed to the C compiler command line.

Historically, they have been systematically prepended without a way to override them.

If the following is passed, the mechanism is slightly altered:

```
(use_standard_c_and_cxx_flags)
```

In this mode, Dune will populate the `:standard` set of C flags with the content of `ocamlc_cflags` and `ocamlc_cppflags`. These flags can be completed or overridden using the *Ordered Set Language*.

This is the default in the language version 3.0.

3.2.17 using

(using <plugin> <version>)

Enable a dune language extension.

The language of configuration files read by Dune can be extended to support additional stanzas (e.g., `menhir`, `coq.theory`, `mdx`).

<*plugin*> is the name of the plugin that defines this stanza and <*version*> describes the configuration language's version. Note that this version has nothing to do with the version of the associated tool or library. In particular, adding a `using` stanza will not result in a build dependency in the generated `.opam` file. See *generate_opam_files*.

Example:

```
(using mdx 0.3)
```

3.2.18 version

(version <version>)

Set the version of the project.

Example:

```
(version 1.2.3)
```

3.2.19 warnings

(warnings ...)

New in version 3.11.

Configure Dune warnings for the project.

(<name> <enabled | disabled>)

Enable or disable the warning <name> for the current project.

3.2.20 wrapped_executables

(wrapped_executables <bool>)

New in version 1.11.

Control wrapping of modules in executables.

Executables are made of compilation units whose names may collide with libraries' compilation units. To avoid this possibility, Dune prefixes these compilation unit names with `Dune__exe__`. This is entirely transparent to users except when such executables are debugged. In which case, the mangled names will be visible in the debugger.

- with `(wrapped_executables false)`, the original names are used.
- with `(wrapped_executables true)`, the names are mangled.

Starting in language version 2.0, the default value is `true`.

3.3 dune-workspace

By default, a workspace has only one build context named `default` which corresponds to the environment, in which `dune` is run. You can define more contexts by writing a `dune-workspace` file.

You can point Dune to an explicit `dune-workspace` file with the `--workspace` option. For instance, it's good practice to write a `dune-workspace.dev` in your project with all the OCaml versions your projects support, so developers can test that the code builds with all OCaml versions by simply running:

```
$ dune build --workspace dune-workspace.dev @all @runtest
```

The `dune-workspace` file uses the S-expression syntax. This is what a typical `dune-workspace` file looks like:

```
(lang dune 3.14)
(context (opam (switch 4.07.1)))
(context (opam (switch 4.08.1)))
(context (opam (switch 4.11.1)))
```

The rest of this section describe the stanzas available.

Note that an empty `dune-workspace` file is interpreted the same as one containing exactly:

```
(lang dune 3.2)
(context default)
```

This allows you to use an empty `dune-workspace` file to mark the root of your project.

3.3.1 `config` stanzas

Starting in Dune 3.0, any of the stanzas from the *config* file can be used in the `dune-workspace` file. In this case, the configuration stanza will only affect the current workspace.

3.3.2 context

The `(context ...)` stanza declares a build context. The argument can be either `default` or `(default)` for the default build context, or it can be the description of an opam switch, as follows:

```
(context (opam (switch <opam-switch-name>)
               <optional-fields>))
```

<optional-fields> are:

- `(name <name>)` is the subdirectory's name for `_build`, where this build's context artifacts will be stored.
- `(root <opam-root>)` is the opam root. By default, it will take the opam root defined by the environment in which `dune` is run, which is usually `~/.opam`.
- `(merlin)` instructs Dune to use this build context for Merlin.
- `(generate_merlin_rules)` instructs Dune to generate Merlin rules for this context, even if it is not the one selected via `(merlin)`.
- `(profile <profile>)` sets a different profile for a *build context*. This has precedence over the command-line option `--profile`.

- `(env <env>)` sets the environment for a particular context. This is of higher precedence than the root `env` stanza in the workspace file. This field has the same options as the *env* stanza.
- `(toolchain <findlib_toolchain>)` sets a `findlib` toolchain for the context.
- `(host <host_context>)` chooses a different context to build binaries that are meant to be executed on the host machine, such as preprocessors.
- `(paths (<var1> <val1>) .. (<varN> <valN>))` allows you to set the value of any `PATH`-like variables in this context. If `PATH` itself is modified in this way, its value will be used to resolve workspace binaries, including finding the compiler and related tools. These variables will also be passed as part of the environment to any program launched by Dune. For each variable, the value is specified using the *Ordered Set Language*. Relative paths are interpreted with respect to the workspace root. See *Finding the Root*.
- `(fdo <target_exe>)` builds this context with feedback-direct optimizations. It requires OCamlFDO. `<target_exe>` is a path-interpreted relative to the workspace root (see *Finding the Root*). `<target_exe>` specifies which executable to optimize. Users should define a different context for each target executable built with FDO. The context name is derived automatically from the default name and `<target-exe>`, unless explicitly specified using the `(name ...)` field. For example, if `<target_exe>` is *src/foo.exe* in a default context, then the name of the context is *default-fdo-foo* and the filename that contains execution counters is *src/fdo.exe.fdo-profile*. This feature is **experimental** and no backwards compatibility is implied.
- By default, Dune builds and installs dynamically-linked foreign archives (usually named `dll*.so`). It's possible to disable this by setting by including `(disable_dynamically_linked_foreign_archives true)` in the workspace file, so bytecode executables will be built with all foreign archives statically linked into the runtime system.

Both `(default ...)` and `(opam ...)` accept a `targets` field in order to setup cross compilation. See *Cross-Compilation* for more information.

Merlin reads compilation artifacts, and it can only read the compilation artifacts of a single context. Usually, you should use the artifacts from the `default` context, and if you have the `(context default)` stanza in your `dune-workspace` file, that is the one Dune will use.

For rare cases where this is not what you want, you can force Dune to use a different build contexts for Merlin by adding the field `(merlin)` to this context.

3.3.3 env

The `env` stanza can be used to set the base environment for all contexts in this workspace. This environment has the lowest precedence of all other `env` stanzas. The syntax for this stanza is the same as Dune's *env* stanza.

3.3.4 profile

The build profile can be selected in the `dune-workspace` file by write a `(profile ...)` stanza. For instance:

```
(profile release)
```

Note that the command line option `--profile` has precedence over this stanza.

3.4 config

This file is used to set Dune's global configuration, which is applicable across projects and workspaces.

The configuration file is normally `~/.config/dune/config` on Unix systems and `%LOCALAPPDATA%/dune/config` on Windows. However, for most Dune commands, it is possible to specify an alternative configuration file with the `--config-file` option. Command-line flags take precedence over the contents of the `config` file. If `--no-config` or `-p` is passed, Dune will not read this file.

The `config` file can contain the following stanzas:

3.4.1 action_stdout_on_success

Specifies how Dune should handle the standard output of actions when they succeed. This can be used to reduce the noise of large builds.

```
(action_stdout_on_success <setting>)
```

where <setting> is one of:

- `print` prints the output on the terminal. This is the default.
- `swallow` ignores the output and does not print it on the terminal.
- `must-be-empty` enforces that the output should be empty. If it is not, Dune will fail.

3.4.2 action_stderr_on_success

Same as *action_stdout_on_success*, but applies to standard error instead of standard output.

3.4.3 cache

Specifies whether Dune is allowed to store and fetch build targets from the Dune cache.

```
(cache <setting>)
```

where <setting> is one of:

- `enabled` enables Dune cache.
- `disabled` disables Dune cache.

3.4.4 cache-check-probability

While the main purpose of Dune cache is to speed up build times, it can also be used to check build reproducibility. It is possible to enable a probabilistic check, in which Dune will re-execute randomly chosen build rules and compare their results with those stored in the cache. If the results differ, the rule is not reproducible, and Dune will print out a corresponding warning.

```
(cache-check-probability <number>)
```

where <number> is a floating-point number between 0 and 1 (inclusive). 0 means never to check for reproducibility, and 1 means to always perform the check.

3.4.5 cache-storage-mode

Specify the mechanism used by the Dune cache for storage.

```
(cache-storage-mode <setting>)
```

where <setting> is one of:

- auto lets Dune decide the best mechanism to use.
- hardlink uses hard links for entries in the cache. If the cache is stored in a different partition than the one where the build is taking place, then this mode will not work and copy should be used instead.
- copy copies entries to the cache. This is less efficient than using hard links.

3.4.6 display

Specify the amount of Dune's verbosity.

```
(display <setting>)
```

where <setting> is one of:

- progress, Dune shows and updates a status line as build goals are being completed. This is the default value.
- verbose prints the full command lines of programs being executed by Dune, with some colors to help differentiate programs.
- short prints a line for each program executed with the binary name on the left and the targets of the action on the right.
- quiet only display errors.

3.4.7 jobs

Maximum number of concurrent jobs Dune is allowed to have.

```
(jobs <setting>)
```

where <setting> is one of:

- auto, auto-detect maximum number of cores. This is the default value.
- <number>, a positive integer specifying the maximum number of jobs Dune may use simultaneously.

3.4.8 sandboxing_preference

The preferred sandboxing setting. Individual rules may specify different preferences. Dune will try to utilize a setting satisfying both conditions.

```
(sandboxing_preference <setting> <setting> ...)
```

where each <setting> can be one of:

- none disables sandboxing.
- hardlink uses hard links for sandboxing. This is the default under Linux.

- `copy` copies files for sandboxing. This is the default under Windows.
- `symlink` uses symbolic links for sandboxing.

3.4.9 terminal-persistence

Specifies how Dune handles the terminal when a rebuild is triggered in watch mode.

```
(terminal-persistence <setting>)
```

where <setting> is one of:

- `preserve` does not clear the terminal screen between rebuilds.
- `clear-on-rebuild` clears the terminal screen between rebuilds.
- `clear-on-rebuild-and-flush-history` clears the terminal between rebuilds, and it also deletes everything in the scrollback buffer.

3.5 Lexical Conventions

All configuration files read by Dune use a simple syntax that's similar to S-expressions. The Dune language can represent three kinds of values: atoms, strings, and lists. By combining these, it's possible to construct arbitrarily complex project descriptions.

A Dune configuration file is a sequence of atoms, strings, or lists separated by spaces, newlines, and comments. The other sections of this manual describe how each configuration file is interpreted, and we illustrate the syntax below:

3.5.1 Comments

The Dune language only has end of line comments. A semicolon introduces end of line comments and span up to the end of the current line. The system ignores everything from the semicolon to the end of the line. For instance:

```
; This is a comment
```

3.5.2 Atoms

An atom is a non-empty contiguous sequences of character other than special characters. Special characters are:

- spaces, horizontal tabs, newlines and form feed
- opening and closing parenthesis
- double quotes
- semicolons

For instance `hello` or `+` are valid atoms.

Note that backslashes inside atoms have no special meaning and Dune always interprets them as plain backslash characters.

3.5.3 Strings

A string is a sequence of characters surrounded by double quotes. A string represent the exact text between the double quotes, except for escape sequences. A backslash character introduces escape sequences. Dune recognizes and interprets the following escape sequences:

- `\n` to represent a newline character
- `\r` to represent a carriage return (character with ASCII code 13)
- `\b` to represent ASCII character 8
- `\t` to represent a horizontal tab
- `\NNN`, a backslash followed by three decimal characters to represent the character with ASCII code `NNN`
- `\xHH`, a backslash followed by two hexadecimal characters to represent the character with ASCII code `HH` in hexadecimal
- `\\`, a double backslash to represent a single backslash
- `\%{` to represent `%{` (see *Variables*)

Additionally, you can use a backslash just before the end of the line. This skips the newline leading up to the next non-space character. For instance, the following two strings represent the same text:

```
"abcdef"
"abc\
   def"
```

In most places where Dune expects a string, it will also accept an atom. As a result, it's possible to write most Dune configuration files using very few double quotes. This is very convenient in practice.

3.5.4 End of Line Strings

You can also write string using end of line strings. They are a convenient way to write blocks of text inside a Dune file.

The characters `"\|` or `"\>` introduce end of line strings and span to the end of the current line. If the next line also starts with `"\|` or `"\>`, Dune reads it as a continuation of the same string. For readability, either leave the text following the delimiter empty or start it with a space (that will be ignored).

For instance:

```
"\| this is a block
"\| of text
```

represents the same text as the string `"this is a block\nof text"`.

Escape sequences are interpreted in text that follows `"\|` but not in text that follows `"\>`. Both delimiters can be mixed inside the same block of text.

3.5.5 Lists

Lists are sequences of values enclosed by parentheses. For instance `(x y z)` is a list containing the three atoms `x`, `y` and `z`. Lists can be empty, for instance: `()`.

Lists can be nested, allowing arbitrary representation for complex descriptions. For instance:

```
(html
 (head (title "Hello world!"))
 (body
   This is a simple example of using S-expressions))
```

3.6 Actions

`(action ...)` fields describe user actions.

User actions are always run from the same subdirectory of the current build context as the `dune` file they are defined in, so for instance, an action defined in `src/foo/dune` will be run from `$build/<context>/src/foo`.

The argument of `(action ...)` fields is a small DSL that's interpreted by Dune directly and doesn't require an external shell. All atoms in the DSL support *Variables*. Moreover, you don't need to specify dependencies explicitly for the special `%{<kind>:...}` forms; these are recognized and automatically handled by Dune.

The DSL is currently quite limited, so if you want to do something complicated, it's recommended to write a small OCaml program and use the DSL to invoke it. You can use shexp to write portable scripts or *Configurator* for configuration related tasks. You can also use *(Experimental) Dune Action Plugin* to express program dependencies directly in the source code.

The following constructions are available:

3.6.1 run

`(run <prog> <args>)`

Execute a program. `<prog>` is resolved locally if it is available in the current workspace, otherwise it is resolved using the `PATH`.

Example:

```
(run capnp compile -o %{bin:capnpc-ocaml} schema.capnp)
```

3.6.2 system

`(system <cmd>)`

Execute a command using the system shell: `sh` on Unix and `cmd` on Windows.

Example:

```
(system "command arg1 arg2")
```

3.6.3 bash

(bash <cmd>)

Execute a command using `/bin/bash`. This is obviously not very portable.

Example:

```
(bash "echo $PATH")
```

3.6.4 dynamic-run

(dynamic-run <prog> <args>)

Execute a program that was linked against the `dune-action-plugin` library. `<prog>` is resolved in the same way as in *run*.

Example:

```
(dynamic-run ./plugin.exe)
```

3.6.5 chdir

(chdir <dir> <DSL>)

Run an action in a different directory.

Example:

```
(chdir src
 (run ./build.exe))
```

3.6.6 setenv

(setenv <var> <value> <DSL>)

Run an action with an environment variable set.

Example:

```
(setenv
  VAR value
  (bash "echo $VAR"))
```

3.6.7 with-accepted-exit-codes

(with-accepted-exit-codes <pred> <DSL>)

New in version 2.0.

Specifies the list of expected exit codes for the programs executed in `<DSL>`. `<pred>` is a predicate on integer values, and it's specified using the *Predicate Language*. `<DSL>` can only contain nested occurrences of `run`, `bash`, `system`, `chdir`, `setenv`, `ignore-<outputs>`, `with-stdin-from`, and `with-<outputs>-to`.

Example:

```
(with-accepted-exit-codes
 (or 1 2)
 (run false))
```

3.6.8 echo

(echo <string>)

Output a string on `stdout`.

Example:

```
(echo "Hello, world")
```

3.6.9 with-<outputs>-to

(with-<outputs>-to <file> <DSL>)

Redirect the output to a file, where <outputs> is one of: `stdout`, `stderr` or `outputs` (for both `stdout` and `stderr`).

Example:

```
(with-stdout-to conf.txt
 (run ./get-conf.exe))
```

3.6.10 with-stdin-from

(with-stdin-from <file> <DSL>)

Redirect the input from a file.

Example:

```
(with-stdin-from data.txt
 (run ./tests.exe))
```

3.6.11 ignore-<outputs>

(ignore-<outputs> <DSL>)

Ignore the output, where <outputs> is one of: `stdout`, `stderr`, or `outputs`.

Example:

```
(ignore-stderr
 (run ./get-conf.exe))
```

3.6.12 cat

(cat <file> ...)

Sequentially print the contents of files to stdout.

Example:

```
(cat data.txt)
```

3.6.13 copy

(copy <src> <dst>)

Copy a file. If these files are OCaml sources, you should follow the `module_name.xxx.ml` *naming convention* to preserve Merlin's functionality.

Example:

```
(copy data.txt.template data.txt)
```

3.6.14 copy#

(copy# <src> <dst>)

Copy a file and add a line directive at the beginning.

Example:

```
(copy# config.windows.ml config.ml)
```

More precisely, `copy#` inserts the following line:

```
# 1 "<source file name>"
```

Most languages recognize such lines and update their current location to report errors in the original file rather than the copy. This is important because the copy exists only under the `_build` directory, and in order for editors to jump to errors when parsing the build system's output, errors must point to files that exist in the source tree. In the beta versions of Dune, `copy#` was called `copy-and-add-line-directive`. However, most of time, one wants this behavior rather than a bare copy, so it was renamed to something shorter.

3.6.15 write-file

(write-file <file> <string>)

Writes `<string>` to `<file>`.

Example:

```
(write-file users.txt jane,joe)
```

3.6.16 pipe-<outputs>

(pipe-<outputs> <DSL> <DSL> <DSL>...)

New in version 2.7.

Execute several actions (at least two) in sequence, filtering the <outputs> of the first command through the other command, piping the standard output of each one into the input of the next.

Example:

```
(pipe-stdout
 (run ./list-tests.exe)
 (run ./exec-tests.exe))
```

3.6.17 diff

(diff <file1> <file2>)

(diff <file1> <file2>) is similar to (run diff <file1> <file2>) but is better and allows promotion. See *Diffing and Promotion* for more details.

Example:

```
(diff test.expected test.output)
```

3.6.18 diff?

(diff? <file1> <file2>)

(diff? <file1> <file2>) is similar to (diff <file1> <file2>) except that <file2> should be produced by a part of the same action rather than be a dependency, is optional and will be consumed by diff?.

Example:

```
(progn
 (with-stdout-to test.output (run ./test.exe))
 (diff? test.expected test.output))
```

3.6.19 cmp

(cmp <file1> <file2>)

(cmp <file1> <file2>) is similar to (run cmp <file1> <file2>) but allows promotion. See *Diffing and Promotion* for more details.

Example:

```
(cmp bin.expected bin.output)
```

3.6.20 progn

(progn <DSL> ...)

Execute several commands in sequence.

Example:

```
(progn
 (run ./proga.exe)
 (run ./progb.exe))
```

3.6.21 concurrent

(concurrent <DSL> ...)

Execute several commands concurrently and collect all resulting errors, if any.

> **Warning:** The concurrency is limited by the `-j` flag passed to Dune. In particular, if Dune is running with `-j 1`, these commands will actually run sequentially, which may cause a deadlock if they talk to each other.

Example:

```
(concurrent
 (run ./proga.exe)
 (run ./progb.exe))
```

3.6.22 no-infer

(no-infer <DSL>)

Perform an action without inference of dependencies and targets. This is useful if you are generating dependencies in a way that Dune doesn't know about, for instance by calling an external build system.

Example:

```
(no-infer
 (progn
  (run make)
  (copy mylib.a lib.a)))
```

Note: expansion of the special `%{<kind>:...}` is done relative to the current working directory of the DSL being executed. So for instance, if you have this action in a `src/foo/dune`:

```
(action (chdir ../../.. (echo %{dep:dune})))
```

Then `%{dep:dune}` will expand to `src/foo/dune`. When you run various tools, they often use the filename given on the command line in error messages. As a result, if you execute the command from the original directory, it will only see the basename.

To understand why this is important, let's consider this `dune` file living in `src/foo`:

```
(rule
 (target blah.ml)
 (deps blah.mll)
 (action
  (run ocamllex -o %{target} %{deps})))
```

Here the command that will be executed is:

```
$ ocamllex -o blah.ml blah.mll
```

And it will be executed in `_build/<context>/src/foo`. As a result, if there is an error in the generated `blah.ml` file, it will be reported as:

```
File "blah.ml", line 42, characters 5-10:
Error: ...
```

Which can be a problem, as your editor might think that `blah.ml` is at the root of your project. Instead, this is a better way to write it:

```
(rule
 (target blah.ml)
 (deps blah.mll)
 (action
  (chdir %{workspace_root}
   (run ocamllex -o %{target} %{deps}))))
```

3.7 Ordered Set Language

A few fields take an ordered set as argument and can be specified using a small DSL.

Dune interprets this DSL into an ordered set of strings using the following rules:

- `:standard` denotes the standard value of the field when it's absent
- an atom not starting with a `:` is a singleton containing only this atom
- a list of sets is the concatenation of its inner sets
- `(<sets1> \ <sets2>)` is the set composed of elements of `<sets1>` that do not appear in `<sets2>`

In addition, some fields support the inclusion of an external file using the syntax `(:include <filename>)`. For instance, this is useful when you need to run a script to figure out some compilation flags. `<filename>` is expected to contain a single S-expression and cannot contain `(:include ...)` forms.

Note that inside an ordered set, a list's first element cannot be an atom except if it starts with `-` or `:`. The reason for this is that we're planning to add simple programmatic features in the future so that one may write:

```
(flags (if (>= %{ocaml_version} 4.06) ...))
```

This restriction will allow you to add this feature without introducing breaking changes. If you want to write a list where the first element doesn't start with `-`, you can simply quote it: `("x" y z)`.

Most fields using the ordered set language also support *Variables*. Variables are expanded after the set language is interpreted.

3.8 Boolean Language

The Boolean language allows the user to define simple Boolean expressions that Dune can evaluate. Here's a semiformal specification of the language:

```
op    ::=   '=' | '<' | '>' | '<>' | '>=' | '<='
expr  ::=   (and <expr>+)
            (or <expr>+)
            (<op> <template> <template>)
            (not <expr>)
            <template>
```

After an expression is evaluated, it must be exactly the string `true` or `false` to be considered as a Boolean. Any other value will be treated as an error.

Below is a simple example of a condition expressing that the build has a Flambda compiler, with the help of variable expansion, and is targeting OSX:

```
(and %{ocaml-config:flambda} (= %{ocaml-config:system} macosx))
```

3.9 Predicate Language

The predicate language allows the user to define simple predicates (Boolean-valued functions) that Dune can evaluate. Here is a semiformal specification of the predicate language:

```
pred  ::=   (and pred pred)
            (or pred pred)
            (not pred)
            :standard
            element
```

The exact meaning of `:standard` and the nature of `element` depend on the context. For example, in the case of the *subdir*, an `element` corresponds to file glob patterns. Another example is the user action *with-accepted-exit-codes*, where an `element` corresponds to a literal integer.

3.10 Library Dependencies

Library dependencies are specified using `(libraries ...)` fields in `library` and `executables` stanzas.

For libraries defined in the current scope, you can either use the real name or the public name. For libraries that are part of the *installed world*, or for libraries that are part of the current workspace but in another scope, you need to use the public name. For instance: `(libraries base re)`.

When resolving libraries, ones that are part of the workspace are always preferred to ones that are part of the *installed world*.

3.10.1 Alternative Dependencies

Sometimes, one doesn't want to depend on a specific library but rather on whatever is already installed, e.g., to use a different backend, depending on the target.

Dune allows this by using a `(select ... from ...)` form inside the list of library dependencies.

Select forms are specified as follows:

```
(select <target-filename> from
 (<literals> -> <filename>)
 (<literals> -> <filename>)
 ...)
```

<literals> are lists of literals, where each literal is one of:

- <library-name>, which will evaluate to true if <library-name> is available, either in the workspace or in the *installed world*
- !<library-name>, which will evaluate to true if <library-name> is not available in the workspace or in the *installed world*

When evaluating a select form, Dune will create <target-filename> by copying the file given by the first (<literals> -> <filename>) case where all the literals evaluate to true. It is an error if none of the clauses are selectable. You can add a fallback by adding a clause of the form (-> <file>) at the end of the list.

3.10.2 Re-Exported Dependencies

A dependency `foo` may be marked as always *re-exported* using the following syntax:

```
(re_export foo)
```

For instance:

```
(library
 (name bar)
 (libraries (re_export foo)))
```

This states that this library explicitly re-exports the interface of `foo`. Concretely, when something depends on `bar`, it will also be able to see `foo` independently of whether *implicit transitive dependencies* are allowed or not. When they are allowed, which is the default, all transitive dependencies are visible, whether they are marked as re-exported or not.

3.11 Preprocessing Specification

Some stanzas including `(library)` accept a `(preprocessing)` field. The possible values for its argument are:

```
pp-spec     ::=   <pp-module>
                  (per-module <per-module>+)
per-module  ::=   ( <pp-module> <module>+ )
pp-module   ::=   no_preprocessing
                  (action <action>)
                  (pps <ppx-rewriters-and-flags>)
                  (staged_pps <ppx-rewriters-and-flags>)
```

```
future_syntax
```

3.11.1 no_preprocessing

When `no_preprocessing` is passed, files are given as-is to the compiler. This is the default behavior.

3.11.2 Preprocessing With Actions

In `(action <action>)`, `<action>` uses the same DSL as described in *Actions*, and for the same reason given in that section, it will be executed from the root of the current build context. It's expected to be an action that reads the file given as a dependency named `input-file` and outputs the preprocessed file on its standard output.

More precisely, `(preprocess (action <action>))` acts as if you had set up a rule for every file of the form:

```
(rule
 (target file.pp.ml)
 (deps file.ml)
 (action
  (with-stdout-to %{target}
   (chdir %{workspace_root} <action>))))
```

The equivalent of a `-pp <command>` option passed to the OCaml compiler is `(system "<command> %{input-file}")`.

3.11.3 Using PPX Rewriters

If `(pps <ppx-rewriters-and-flags>)` is used, the corresponding rewriters are set up using the "fast pipeline" (using a separate preprocessing step). If `(staged_pps <ppx-rewriters-and-flags>)` is used, they are set up using the "classic pipeline" (using the `-ppx` command-line argument).

The distinction between these pipelines is explained in *How Preprocessing Works*.

PPX rewriters need to be compiled as a driver to be used by Dune. To run PPXs that do not support this (usually old ones), it is possible to use the ppxfind tool.

3.11.4 Arguments to PPX Rewriters

In `(pps <ppx-rewriters-and-flags>)` and `(staged_pps <ppx-rewriters-and-flags>)`, `<ppx-rewriters-and-flags>` is a sequence where each element is either a command line flag if it starts with a `-` or the name of a library.

If you want to pass command line flags that don't start with a `-`, you can separate library names from flags using `--`. So for instance from the following `preprocess` field:

```
(preprocess (pps ppx1 -foo ppx2 -- -bar 42))
```

The list of libraries will be `ppx1` and `ppx2`, and the command line arguments will be: `-foo -bar 42`.

3.11.5 Future Syntax

The `future_syntax` specification is a special value that brings some of the newer OCaml syntaxes to older compilers.

It is equivalent to `no_preprocessing` when using one of the most recent versions of the compiler. When using an older one, it is a shim preprocessor that backports some of the newer syntax elements. This allows you to use some of the new OCaml features while keeping compatibility with older compilers.

One example of supported syntax is the custom `let-syntax` that was introduced in 4.08, allowing the user to define custom `let` operators.

Note that this feature is implemented by the third-party ocaml-syntax-shims project, so if you use this feature, you must also declare a dependency on this package.

3.11.6 Per-Module Preprocessing Specification

By default, a preprocessing specification applies to all modules in the library/set of executables. It's possible to select the preprocessing on a module-by-module basis by using the `(per-module ...)` syntax. For instance:

```
(preprocess
 (per_module
  ((action (run ./pp.sh X=1 %{input-file})) foo bar)
  ((action (run ./pp.sh X=2 %{input-file})) baz)))
```

The modules `Foo` and `Bar` will be preprocessed with `pp.sh X=1`, and `Baz` will be preprocessed with `pp.sh X=2`.

3.11.7 Preprocessor Dependencies

If your preprocessor needs extra dependencies, you should use the `preprocessor_deps` field available in the `library`, `executable`, and `executables` stanzas. It uses the *Dependency Specification* to declare what the preprocessor needs.

3.12 Cram Tests

3.12.1 Synopsis

Cram tests are integrations tests that describe a shell session. These tests contain commands and expected outputs. When executed, the commands are executed and the actual output is compared to the expected output.

Here is an example showing how `echo`, `cat`, and `rm` interact.

```
Create a file:

  $ echo contents > data.txt

Display it:

  $ cat data.txt
  contents

Remove it:

  $ rm data.txt
```

(continues on next page)

(continued from previous page)

```
Try to remove it again:

  $ rm data.txt
  rm: cannot remove 'data.txt': No such file or directory
  [1]
```

The syntax mimics a shell session: there are comments and shell commands with their output.

3.12.2 Examples

Simple Commands

This is the simplest test case: it executes the command `touch this-file.txt` and expects that the command has no output.

```
$ touch this-file.txt
```

Output

This executes `ls` and expects it to display `this-file.txt`:

```
$ ls
this-file.txt
```

There can be several output lines if the command is expected to print several lines. Also, note that if a command has no output, the next one can come in the next line.

```
$ touch other-file.txt
$ ls
other-file.txt
this-file.txt
```

Comments

Lines that are not indented are ignored. These act as comments.

```
"touch" will create an empty file:

  $ touch data.txt

Printing it will do nothing:

  $ cat data.txt
```

Continuation Lines

Continuation lines are used when a command fits on several lines. This can happen in all the cases where pressing Enter would not run the command. For example, when passing a backslash character to escape the line ending. In that case, all the continuation lines are grouped together as a single command.

This syntax mimics the PS2 prompt in shells - the ">" character is not passed to the command.

```
$ echo \
>   a \
>   b \
>   d \
>   c
a b c d
```

This is often used with shell "heredocs" to create files:

```
$ cat > file.txt << EOF
> Everything
> here will
> written to
> the file
> EOF

$ cat file.txt
Everything
here will
written to
the file
```

Exit Codes

When a command exits with a nonzero exit code, it is displayed between square brackets after its output:

```
$ false
[1]

$ echo hello; false
hello
[1]
```

3.12.3 Syntax Details

Cram tests are parsed line by line, depending on the first characters of each line:

- If a line starts with ␣␣$␣ (␣ denoting a space character), the rest is a command.
- If it starts with ␣␣>␣, the rest is the continuation of a command (continuation lines must immediately follow a command).
- If it start with ␣␣ and something else, the rest is the expected output or exit code of the previous command.
- Everything else is a comment.

3.12.4 File and Directory Tests

There are two types of Cram tests: file tests and directory tests. File tests are files with a `.t` extension. Directory tests are files named `run.t` within a directory with a name that ends with `.t`.

A Cram test begins its execution in a temporary directory where its dependencies (as listed in the corresponding *cram stanzas*, if any) are available. In the case of a directory test, the contents of the directory are also available.

File tests have the nice property that they are self-contained: everything happens in a single file. This is handy because it does not make a deep file hierarchy in a project. But if the test requires some files, these need to be created using `cat` and heredocs. Directory tests, on the other hand, allow creating these test fixtures as normal files. This can be more comfortable because it makes the usual tooling (syntax highlighting, completion, etc.) available.

3.12.5 Executing Cram Tests

Every Cram test has a name. For file tests, the name of `something.t` is `something`, and for directory tests, the name of `something.t/run.t` is `something`.

There are several ways to execute Cram tests:

- All Cram tests are attached to the *@runtest* alias. So `dune runtest` will run all Cram tests.
- Every Cram test creates an alias after its name. So, `dune build @something` will run tests named `something`.

When a Cram test is executed, the commands it contains are executed, and a corrected file is created where the command outputs are inserted after each command. This corrected file is then offered for *promotion* by Dune.

Concretely, this means that Dune will display the difference between the Cram test's current contents and the latest run's output. This diff can be applied by running `dune promote`, as usual.

```
 $ touch changed-name.txt
 $ ls
-other-file.txt
+changed-name.txt
 this-file.txt
```

3.13 Scopes

Any directory containing at least one `<package>.opam` file defines a scope. This scope is the subtree starting from this directory, excluding any other scopes rooted in subdirectories.

Typically, any given project will define a single scope. Libraries and executables that aren't meant to be installed will be visible inside this scope only.

Because scopes are exclusive, if you wish to include your current project's dependencies in your workspace, you can copy them in a `vendor` directory, or any name of your choice. Dune will look for them there rather than in the *installed world*, and there will be no overlap between the various scopes.

3.14 Variables

Some fields can contains variables that are expanded by Dune. The syntax of variables is as follows:

```
%{var}
```

or, for more complex forms that take an argument:

```
%{fun:arg}
```

In order to write a plain `%{`, you need to write `\%{` in a string.

Dune supports the following variables:

- `project_root` is the root of the current project. It is typically the root of your project, and as long as you have a `dune-project` file there, `project_root` is independent of the workspace configuration.
- `workspace_root` is the root of the current workspace. Note that the value of `workspace_root` isn't constant and depends on whether your project is vendored or not.
- `cc` is the C compiler command line (list made of the compiler name followed by its flags) that will be used to compile foreign code. For more details about its content, please see *this section*.
- `cxx` is the C++ compiler command line being used in the current build context.
- `ocaml_bin` is the path where `ocamlc` lives.
- `ocaml` is the `ocaml` binary.
- `ocamlc` is the `ocamlc` binary.
- `ocamlopt` is the `ocamlopt` binary.
- `ocaml_version` is the version of the compiler used in the current build context.
- `ocaml_where` is the output of `ocamlc -where`.
- `arch_sixtyfour` is `true` if using a compiler that targets a 64-bit architecture and `false` otherwise.
- `null` is `/dev/null` on Unix or `nul` on Windows.
- `ext_obj`, `ext_asm`, `ext_lib`, `ext_dll`, and `ext_exe` are the file extensions used for various artifacts.
- `ext_plugin` is `.cmxs` if `natdynlink` is supported and `.cma` otherwise.
- `ocaml-config:v` is for every variable `v` in the output of `ocamlc -config`. Note that Dune processes the output of `ocamlc -config` in order to make it a bit more stable across versions, so the exact set of variables accessible this way might not be exactly the same as what you can see in the output of `ocamlc -config`. In particular, variables added in new OCaml versions need to be registered in Dune before they can be used.
- `profile` is the profile selected via `--profile`.
- `context_name` is the name of the context (`default`, or defined in the workspace file)
- `os_type` is the type of the OS the build is targeting. This is the same as `ocaml-config:os_type`.
- `architecture` is the type of the architecture the build is targeting. This is the same as `ocaml-config:architecture`.
- `model` is the type of the CPU the build is targeting. This is the same as `ocaml-config:model`.
- `system` is the name of the OS the build is targeting. This is the same as `ocaml-config:system`.
- `ignoring_promoted_rules` is `true` if `--ignore-promoted-rules` was passed on the command line and `false` otherwise.

- `<ext>:<path>` where `<ext>` is one of `cmo`, `cmi`, `cma`, `cmx`, or `cmxa`. See *Variables for Artifacts*.
- `env:<var>=<default` expands to the value of the environment variable `<var>`, or `<default>` if it does not exist. For example, `%{env:BIN=/usr/bin}`. Available since Dune 1.4.0.
- There are some Coq-specific variables detailed in *Coq-Specific Variables*.

In addition, `(action ...)` fields support the following special variables:

- `target` expands to the one target.
- `targets` expands to the list of target.
- `deps` expands to the list of dependencies.
- `^` expands to the list of dependencies, separated by spaces.
- `dep:<path>` expands to `<path>` (and adds `<path>` as a dependency of the action).
- `exe:<path>` is the same as `<path>`, except when cross-compiling, in which case it will expand to `<path>` from the host build context.
- `bin:<program>` expands `<path>` to `program`. If `program` is installed by a workspace package (see *install* stanzas), the locally built binary will be used, otherwise it will be searched in the `<path>` of the current build context. Note that `(run %{bin:program} ...)` and `(run program ...)` behave in the same way. `%{bin:..}` is only necessary when you are using `(bash ...)` or `(system ...)`.
- `bin-available:<program>` expands to `true` or `false`, depending on whether `<program>` is available or not.
- `lib:<public-library-name>:<file>` expands to the file's installation path `<file>` in the library `<public-library-name>`. If `<public-library-name>` is available in the current workspace, the local file will be used, otherwise the one from the *installed world* will be used.
- `lib-private:<library-name>:<file>` expands to the file's build path `<file>` in the library `<library-name>`. Both public and private library names are allowed as long as they refer to libraries within the same project.
- `libexec:<public-library-name>:<file>` is the same as `lib:...`, except when cross-compiling, in which case it will expand to the file from the host build context.
- `libexec-private:<library-name>:<file>` is the same as `lib-private:...` except when cross-compiling, in which case it will expand to the file from the host build context.
- `lib-available:<library-name>` expands to `true` or `false` depending on whether the library is available or not. A library is available if at least one of the following conditions holds:
  - It's part the *installed world*.
  - It's available locally and is not optional.
  - It's available locally, and all its library dependencies are available.
- `version:<package>` expands to the version of the given package. Packages defined in the current scope have priority over the public packages. Public packages that don't install any libraries will not be detected. How Dune determines the version of a package is described *here*.
- `read:<path>` expands to the contents of the given file.
- `read-lines:<path>` expands to the list of lines in the given file.
- `read-strings:<path>` expands to the list of lines in the given file, unescaped using OCaml lexical convention.

The `%{<kind>:...}` forms are what allows you to write custom rules that work transparently, whether things are installed or not.

Note that aliases are ignored by `%{deps}`

The intent of this last form is to reliably read a list of strings generated by an OCaml program via:

```
List.iter (fun s -> print_string (String.escaped s)) l
```

1. Dealing with circular dependencies introduced by variables

If you ever see Dune reporting a dependency cycle that involves a variable such as *%{read:<path>}*, try to move *<path>* to a different directory.

The reason you might see such dependency cycle is because Dune is trying to evaluate the *%{read:<path>}* too early. For instance, let's consider the following example:

```
(rule
 (targets x)
 (enabled_if %{read:y})
 (action ...))

(rule
 (with-stdout-to y (...)))
```

When Dune loads and interprets this file, it decides whether the first rule is enabled by evaluating `%{read:y}`. To evaluate `%{read:y}`, it must build `y`. To build `y`, it must figure out the build rule that produces `y`, and in order to do that, it must first load and evaluate the above `dune` file. You can see how this creates a cycle.

Some cycles might be more complex. In any case, when you see such an error, the easiest thing to do is move the file that's being read to a different directory, preferably a standalone one. You can use the *subdir* stanza to keep the logic self-contained in the same `dune` file:

```
(rule
 (targets x)
 (enabled_if %{read:dir-for-y/y})
 (action ...))

(subdir
 dir-for-y
 (rule
  (with-stdout-to y (...))))
```

3.14.1 Expansion of Lists

Forms that expand to a list of items, such as `%{cc}`, `%{deps}`, `%{targets}`, or `%{read-lines:...}`, are suitable to be used in `(run <prog> <arguments>)`. For instance in:

```
(run foo %{deps})
```

If there are two dependencies, `a` and `b`, the produced command will be equivalent to the shell command:

```
$ foo "a" "b"
```

If you want both dependencies to be passed as a single argument, you must quote the variable:

```
(run foo "%{deps}")
```

which is equivalent to the following shell command:

```
$ foo "a b"
```

(The items of the list are concatenated with space.) Please note: since `%{deps}` is a list of items, the first one may be used as a program name. For instance:

```
(rule
 (targets result.txt)
 (deps    foo.exe (glob_files *.txt))
 (action  (run %{deps})))
```

Here is another example:

```
(rule
 (target foo.exe)
 (deps   foo.c)
 (action (run %{cc} -o %{target} %{deps} -lfoolib)))
```

3.15 Dependency Specification

Dependencies in `dune` files can be specified using one of the following:

- `(:name <dependencies>)` will bind the list of dependencies to the `name` variable. This variable will be available as `%{name}` in actions.
- `(file <filename>)`, or simply `<filename>`, depend on this file.
- `(alias <alias-name>)` depends on the construction of this alias. For instance: `(alias src/runtest)`.
- `(alias_rec <alias-name>)` depends on the construction of this alias recursively in all children directories wherever it is defined. For instance: `(alias_rec src/runtest)` might depend on `(alias src/runtest)`, `(alias src/foo/bar/runtest)`, etc.
- `(glob_files <glob>)` depends on all files matched by `<glob>`. See the *glob* for details.
- `(glob_files_rec <glob>)` is the recursive version of `(glob_files <glob>)`. See the *glob* for details.
- `(source_tree <dir>)` depends on all source files in the subtree with root `<dir>`.
- `(universe)` depends on everything in the universe. This is for cases where dependencies are too hard to specify. Note that Dune will not be able to cache the result of actions that depend on the universe. In any case, this is only for dependencies in the *installed world*. You must still specify all dependencies that come from the workspace.
- `(package <pkg>)` depends on all files installed by `<package>`, as well as on the transitive package dependencies of `<package>`. This can be used to test a command against the files that will be installed.
- `(env_var <var>)` depends on the value of the environment variable `<var>`. If this variable becomes set, becomes unset, or changes value, the target will be rebuilt.
- `(sandbox <config>)` requires a particular sandboxing configuration. `<config>` can be one (or many) of:
  - `always`: the action requires a clean environment
  - `none`: the action must run in the build directory
  - `preserve_file_kind`: the action needs the files it reads to look like normal files (so Dune won't use symlinks for sandboxing)
- `(include <file>)` read the s-expression in `<file>` and interpret it as additional dependencies. The s-expression is expected to be a list of the same constructs enumerated here.

In all these cases, the argument supports *Variables*.

3.15.1 Named Dependencies

Dune allows a user to organize dependency lists by naming them. The user is allowed to assign a group of dependencies a name that can later be referred to in actions (like the `%{deps}`, `%{target}`, and `%{targets}` built in variables).

One instance where this is useful is for naming globs. Here's an example of an imaginary bundle command:

```
(rule
 (target archive.tar)
 (deps
  index.html
  (:css (glob_files *.css))
  (:js foo.js bar.js)
  (:img (glob_files *.png) (glob_files *.jpg)))
 (action
  (run %{bin:bundle} index.html -css %{css} -js %{js} -img %{img} -o %{target})))
```

Note that a named dependency list can also include unnamed dependencies (like `index.html` in the example above). Also, such user defined names will shadow build in variables, so `(:workspace_root x)` will shadow the built-in `%{workspace_root}` variable.

3.15.2 Glob

You can use globs to declare dependencies on a set of files. Note that globs will match files that exist in the source tree as well as buildable targets, so for instance you can depend on `*.cmi`.

Dune supports globbing files in a single directory via `(glob_files ...)` and, starting with Dune 3.0, in all subdirectories recursively via `(glob_files_rec ...)`. The glob is interpreted as follows:

- anything before the last `/` is taken as a literal path
- anything after the last `/`, or everything if the glob contains no `/`, is interpreted using the glob syntax

Absolute paths are permitted in the `(glob_files ...)` term only. It's an error to pass an absolute path (i.e., a path beginning with a `/`) to `(glob_files_rec ...)`.

The glob syntax is interpreted as follows:

- `\<char>` matches exactly `<char>`, even if it's a special character (`*`, `?`, ...).
- `*` matches any sequence of characters, except if it comes first, in which case it matches any character that is not `.` followed by anything.
- `**` matches any character that is not `.` followed by anything, except if it comes first, in which case it matches anything.
- `?` matches any single character.
- `[<set>]` matches any character that is part of `<set>`.
- `[!<set>]` matches any character that is not part of `<set>`.
- `{<glob1>,<glob2>,...,<globn>}` matches any string that is matched by one of `<glob1>`, `<glob2>`, etc.

Table 5: Glob syntax examples

Syntax	Files matched	Files not matched
`x`	`x`	`y`
`\*`	`*`	`x`
`file*.txt`	`file1.txt`, `file2.txt`	`f.txt`
`*.txt`	`f.txt`	`.hidden.txt`
`a**`	`aml`	`a.ml`
`**`	`a/b`, `a.b`	(none)
`a?.txt`	`a1.txt`, `a2.txt`	`b1.txt`, `a10.txt`
`f[xyz].txt`	`fx.txt`, `fy.txt`, `fz.txt`	`f2.txt`, `f.txt`
`f[!xyz].txt`	`f2.txt`, `fa.txt`	`fx.txt`, `f.txt`
`a.{ml,mli}`	`a.ml`, `a.mli`	`a.txt`, `b.ml`
`../a.{ml,mli}`	`../a.ml`, `../a.mli`	`a.ml`

3.16 OCaml Flags

In `library`, `executable`, `executables`, and `env` stanzas, you can specify OCaml compilation flags using the following fields:

- `(flags <flags>)` to specify flags passed to both `ocamlc` and `ocamlopt`
- `(ocamlc_flags <flags>)` to specify flags passed to `ocamlc` only
- `(ocamlopt_flags <flags>)` to specify flags passed to `ocamlopt` only

For all these fields, `<flags>` is specified in the *Ordered Set Language*. These fields all support `(:include ...)` forms.

The default value for `(flags ...)` is taken from the environment, as a result it's recommended to write `(flags ...)` fields as follows:

```
(flags (:standard <my options>))
```

3.17 Sandboxing

The user actions that run external commands (`run`, `bash`, `system`) are opaque to Dune, so Dune has to rely on manual specification of dependencies and targets. One problem with manual specification is that it's error-prone. It's often hard to know in advance what files the command will read, and knowing a correct set of dependencies is very important for build reproducibility and incremental build correctness.

To help with this problem Dune supports sandboxing. An idealized view of sandboxing is that it runs the action in an environment where it can't access anything except for its declared dependencies.

In practice, we have to make compromises and have some trade-offs between simplicity, information leakage, performance, and portability.

The way sandboxing is currently implemented is that for each sandboxed action we build a separate directory tree (sandbox directory) that mirrors the build directory, filtering it to only contain the files that were declared as dependencies. We run the action in that directory, and then we copy the targets back to the build directory.

You can configure Dune to use sandboxing modes `symlink`, `hardlink`, or `copy`, which determine how the individual files are populated (they will be symlinked, hardlinked, or copied into the sandbox directory).

This approach is very simple and portable, but that comes with certain limitations:

- The actions in the sandbox can use absolute paths to refer to anywhere outside the sandbox. This means that only dependencies on relative paths in the build tree can be enforced/detected by sandboxing.
- The sandboxed actions still run with full permissions of Dune itself, so sandboxing is not a security feature. It won't prevent network access either.
- We don't erase the environment variables of the sandboxed commands. This is something we want to change.
- Performance impact is usually small, but it can get noticeable for fast actions with very large sets of dependencies.

3.17.1 Per-Action Sandboxing Configuration

Some actions may rely on sandboxing to work correctly. For example, an action may need the input directory to contain nothing except the input files, or the action might create temporary files that break other build actions.

Some other actions may refuse to work with Sandboxing. For example, if they rely on absolute path to the build directory staying fixed, or if they deliberately use some files without declaring dependencies (this is usually a very bad idea, by the way).

Generally it's better to improve the action so it works with or without sandboxing (especially with), but sometimes you just can't do that.

Things like this can be described using the "sandbox" field in the dependency specification language (see *Dependency Specification*).

3.17.2 Global Sandboxing Configuration

Dune always respects per-action sandboxing specification. You can configure it globally to prefer a certain sandboxing mode if the action allows it.

This is controlled by:

- `dune --sandbox <...>` CLI flag (see `man dune-build`)
- `DUNE_SANDBOX` environment (see `man dune-build`)
- `(sandboxing_preference ..)` field in the configuration file (see `man dune-config`)

3.18 Locks

Given two rules that are independent, Dune will assume that their associated actions can be run concurrently. Two rules are considered independent if neither of them depend on the other, either directly or through a chain of dependencies. This basic assumption allows Dune to parallelize the build.

However, it is sometimes the case that two independent rules cannot be executed concurrently. For instance, this can happen for more complicated tests. In order to prevent Dune from running the actions at the same time, you can specify that both actions take the same lock:

```
(rule
 (alias  runtest)
 (deps   foo)
 (locks  m)
 (action (run test.exe %{deps})))

(alias
```

(continues on next page)

(continued from previous page)

```
(rule   runtest)
(deps   bar)
(locks  m)
(action (run test.exe %{deps})))
```

Dune will make sure that the executions of `test.exe foo` and `test.exe bar` are serialized.

Although they don't live in the filesystem, lock names are interpreted as file names. So for instance, `(with-lock m ...)` in `src/dune` and `(with-lock ../src/m)` in `test/dune` refer to the same lock.

Note also that locks are per build context. So if your workspace has two build contexts setup, the same rule might still be executed concurrently between the two build contexts. If you want a lock that is global to all build contexts, simply use an absolute filename:

```
(rule
 (alias   runtest)
 (deps   foo)
 (locks  /tcp-port/1042)
 (action (run test.exe %{deps})))
```

3.19 Diffing and Promotion

`(diff <file1> <file2>)` is very similar to `(run diff <file1> <file2>)`. In particular it behaves in the same way:

- When `<file1>` and `<file2>` are equal, it does nothing.
- When they are not, the differences are shown and the action fails.

However, it is different for the following reason:

- The exact command used for diff files can be configured via the `--diff-command` command line argument. Note that it's only called when the files are not byte equals
- By default, it will use `patdiff` if it is installed. `patdiff` is a better diffing program. You can install it via opam with:

  ```
  $ opam install patdiff
  ```

- On Windows, both `(diff a b)` and `(diff? a b)` normalize end-of-line characters before comparing the files.
- Since `(diff a b)` is a built-in action, Dune knows that `a` and `b` are needed, so you don't need to specify them explicitly as dependencies.
- You can use `(diff? a b)` after a command that might or might not produce `b`, for cases where commands optionally produce a *corrected* file
- If `<file1>` doesn't exist, it will compare with the empty file.
- It allows promotion. See below.

Note that `(cmp a b)` does no end-of-line normalization and doesn't print a diff when the files differ. `cmp` is meant to be used with binary files.

3.19.1 Promotion

Whenever an action `(diff <file1> <file2>)` or `(diff? <file1> <file2>)` fails because the two files are different, Dune allows you to promote `<file2>` as `<file1>` if `<file1>` is a source file and `<file2>` is a generated file.

More precisely, let's consider the following Dune file:

```
(rule
 (with-stdout-to data.out (run ./test.exe)))

(rule
 (alias   runtest)
 (action (diff data.expected data.out)))
```

Where `data.expected` is a file committed in the source repository. You can use the following workflow to update your test:

- Update the code of your test.
- Run `dune runtest`. The diff action will fail and a diff will be printed.
- Check the diff to make sure it's what you expect. This diff can be displayed again by running `dune promotion diff`.
- Run `dune promote`. This will copy the generated `data.out` file to `data.expected` directly in the source tree.

You can also use `dune runtest --auto-promote`, which will automatically do the promotion.

3.20 Package Specification

Installation is the process of copying freshly built libraries, binaries, and other files from the build directory to the system. Dune offers two ways of doing this: via opam or directly via the `install` command. In particular, the installation model implemented by Dune was copied from opam. Opam is the standard OCaml package manager.

In both cases, Dune only know how to install whole packages. A package being a collection of executables, libraries, and other files. In this section, we'll describe how to define a package, how to "attach" various elements to it, and how to proceed with installing it on the system.

3.20.1 Declaring a Package

To declare a package, simply add a `package` stanza to your `dune-project` file:

```
(package
 (name mypackage)
 (synopsis "My first Dune package!")
 (description "\| This is my first attempt at creating
              "\| a project with Dune.
))
```

Once you have done this, Dune will know about the package named `mypackage` and you will be able to attach various elements to it. The `package` stanza accepts more fields, such as dependencies.

Note that package names are in a global namespace, so the name you choose must be universally unique. In particular, package managers never allow users to release two packages with the same name.

In older projects using Dune, packages were defined by manually writing a file called <package-name>.opam at the root of the project. However, it's not recommended to use this method in new projects, as we expect to deprecate it in the future. The right way to define a package is with a package stanza in the dune-project file.

See *How to Generate Opam Files from dune-project* for instructions on configuring Dune to automatically generate .opam files based on the package stanzas.

3.20.2 Attaching Elements to a Package

Attaching an element to a package means declaring to Dune that this element is part of the said package. The method to attach an element to a package depends on the kind of the element. In this subsection, we will go through the various kinds of elements and describe how to attach each of them to a package.

In the rest of this section, <prefix> refers to the directory in which the user chooses to install packages. When installing via opam, it's opam that sets this directory. When calling dune install, the installation directory is either guessed or can be manually specified by the user. Defaults directories which replace guessing can be set during the compilation of dune.

3.20.3 Sites of a Package

When packages need additional resources outside their binary, their location could be hard to find. Moreover, some packages could add resources to another package, e.g., in the case of plugins. These locations are called sites in Dune. One package can define them. During execution, one site corresponds to a list of directories. They are like layers, and the first directories have a higher priority. Examples and precisions are available at *How to Load Additional Files at Runtime*.

Libraries

In order to attach a library to a package, merely add a public_name field to your library. This is the name that external users of your libraries must use in order to refer to it. Dune requires that a library's public name is either the name of the package it is part of or start with the package name followed by a dot character.

For instance:

```
(library
 (name mylib)
 (public_name mypackage.mylib))
```

After you have added a public name to a library, Dune will know to install it as part of the package it is attached to. Dune installs the library files in a directory <prefix>/lib/<package-name>.

If the library name contains dots, the full directory in which the library files are installed is lib/<comp1>/<comp2>/.../<compn>, where <comp1>, <comp2>, ... <compn> are the dot-separated component of the public library name. By definition, <comp1> is always the package name.

Executables

Similar to libraries, to attach an executable to a package simply add a `public_name` field to your `executable` stanza or a `public_names` field for `executables` stanzas. Designate this name to match the available executables through the installed `PATH` (i.e., the name users must type in their shell to execute the program), because Dune cannot guess an executable's relevant package from its public name. It's also necessary to add a `package` field unless the project contains a single package, in which case the executable will be attached to this package.

For instance:

```
(executable
 (name main)
 (public_name myprog)
 (package mypackage))
```

Once `mypackage` is installed on the system, the user will be able to type the following in their shell:

```
$ myprog
```

to execute the program.

Other Files

For all other kinds of elements, you must attach them manually via an *install* stanza.

3.21 Aliases

Aliases are build targets that do not correspond to specific files. For example, the `runtest` alias corresponds to running tests.

3.21.1 Model and Syntax

Dependencies and actions can be attached to an alias. When this alias is requested to be built, these dependencies are built and these actions are executed. Aliases are attached to specific directories.

In commands such as `dune build`, the syntax to refer to the `x` alias is `@x`, for example `dune build @x`. This is why it is common to refer to it as "the `@x` alias", or "attaching a rule to `@x`".

Building `@x` will build `x` in all subdirectories of the current directory. This is the most common case, but it is possible to restrict this using different syntaxes:

- `@sub/dir/x` will build `x` in `sub/dir` and its subdirectories.
- `@@x` will build `x` in the current directory only.
- `@@sub/dir/x` will build `x` in `sub/dir` only.

If `dir` is the directory of a *build context*, it restricts the alias to this context.

To summarize, the syntax is:

- `@` (recursive) or `@@` (non-recursive): determine if subdirectories are included
- optional *build context root*: restrict to a particular *build context*
- optional directory: only consider this subdirectory

- alias name

Examples:

- `dune build @_build/foo/runtest` only runs the tests for the `foo` build context
- `dune build @runtest` will run the tests for all build contexts

3.21.2 User-Defined Aliases

It is possible to use any name for alias names; it will then be available on the command line. For example, if a Dune file contains the following, then `dune build @deploy` will execute that command.

```
(rule
 (alias deploy)
 (action ./run-deployer.exe))
```

3.21.3 Built-In Aliases

Some aliases are defined and managed by Dune itself:

@all

This alias corresponds to every known file target in a directory.

Since version 2.0 of the dune language, JS targets of executables are no longer included in the *all* alias by default. To get back the old behavior of including the JS targets in *all*, one can add the `js` target to the executable's `modes` field.

@default

This alias corresponds to the default argument for `dune build`: `dune build` is equivalent to `dune build @@default` (`@@` indicates a *non-recursive alias*). Similarly, `dune build dir` is equivalent to `dune build @@dir/default`.

When a directory doesn't explicitly define what the `default` alias means via an *alias* stanza, the following implicit definition is assumed:

```
(alias
 (name default)
 (deps (alias_rec all)))
```

But if such a stanza is present in the `dune` file in a directory, it will be used instead. For example, if the following is present in `tests/dune`, `dune build tests` will run tests there:

```
(alias
 (name default)
 (deps (alias_rec runtest)))
```

@install

Building this alias will create the `*.install` files used by the *opam integration*. In turn, these depend on installable files.

@check

This alias corresponds to the set of targets necessary for development tools to work correctly. For example, it will build `*.cmi`, `*.cmt`, and `*.cmti` files so that Merlin and `ocaml-lsp-server` can be used in the project. It is also useful in the development loop because it will catch compilation errors without executing expensive operations such as linking executables.

See also:

@ocaml-index for a fast feedback loop that also indexes the project.

@ocaml-index

This alias corresponds to the set of targets necessary for development tools to provide project-wide queries such as "get all references of this value". These targets are indexes built using the required *ocaml-index* binary. Since this alias also incudes the `*.cmi`, `*.cmt`, and `*.cmti` files usually built by `check`, it can be used in most projects as a replacement to get a fast feedback loop while maintaining the indexes up-to-date.

@runtest

Actions that run tests are attached to this alias. For example this convention is used by the `(test)` stanza.

`dune runtest` is a shortcut for `dune build @runtest`.

See also:

Writing and Running Tests

@fmt

This alias is used by formatting rules: when it is built, code formatters will be executed (using *promotion*).

`dune fmt` is a shortcut for `dune build @fmt --auto-promote`.

It is possible to build on top of this convention. If some actions are manually attached to the `fmt` alias, they will be executed by `dune fmt`.

Example:

```
(rule
 (with-stdout-to
  data.json.formatted
  (run jq . %{dep:data.json})))

(rule
 (alias fmt)
 (action
  (diff data.json data.json.formatted)))
```

@lint

This alias runs linting tools.

@doc

This alias builds documentation for public libraries as HTML pages.

See also:

Generating Documentation

@doc-private

This alias builds documentation for all libraries, both public & private.

@doc-json

This alias builds documentation for public libraries as JSON files. These are produced by `odoc`'s option `--as-json` and can be consumed by external tools.

3.22 Foreign Sources, Archives, and Objects

Dune provides basic support for including foreign source files as well as archives of foreign object files into OCaml projects via the `foreign_stubs` and `foreign_archives` fields. Individual object files can also be included via the `extra_objects` field.

3.22.1 Foreign Stubs

You can specify foreign sources using the `foreign_stubs` field of the `library` and `executable` stanzas. For example:

```
(library
 (name lib)
 (foreign_stubs (language c) (names src1 src2))
 (foreign_stubs (language cxx) (names src3) (flags -O2)))
```

Here we declare an OCaml library `lib`, which contains two C sources `src1` and `src2`, and one C++ source, `src3`, which need to be compiled with `-O2`. These source files will be compiled and packaged with the library, along with the link-time flags to be used when linking the final executables. When matching `names` to source files, Dune treats `*.c` files as C sources, and `*.cpp`, `*.cc`, and `*.cxx` files as C++ sources.

Here is a complete list of supported subfields:

- `language` specifies the source language, where `c` means C and `cxx` means C++. In the future, more languages may be supported.
- `names` specifies the *names* of source files. When specifying a source file, omit the extension and any relative parts of the path; Dune will scan all library directories to find all matching files and raise an error if multiple source files map to the same object name. If you need to have multiple object files with the same name, you can package them into different *Foreign Archives* via the `foreign_archives` field. This field uses the *Ordered*

Set Language where the `:standard` value corresponds to the set of names of all source files whose extensions match the specified `language`.

- `flags` are passed when compiling source files. This field is specified using the *Ordered Set Language*, where the `:standard` value comes from the environment settings `c_flags` and `cxx_flags`, respectively. Note that, for C stubs, Dune unconditionally adds the flags present in the OCaml config fields `ocamlc_cflags` and `ocamlc_cppflags` to the compiler command line. This behavior can be disabled since Dune 2.8 via the `dune-project` option *use_standard_c_and_cxx_flags*.
- `include_dirs` are tracked as dependencies and passed to the compiler via the `-I` flag. You can use *Variables* in this field and refer to a library source directory using the `(lib library-name)` syntax. Additionally, the syntax `(include filename)` can be used to specify a file containing additional arguments to `(include_dirs ...)`. The named file can either contain a single path to be added to this list of include directories, or an S-expression listing additional `(include_dirs ...)` arguments (the `(lib ...)` and `(include ...)` syntax is also supported in files included in this way). For example, `(include_dirs dir1 (lib lib1) (lib lib2) (include inc1) dir2)` specifies the directory `dir1`, the source directories of `lib1`, and `lib2`, the list of directories contained in the file `inc1`, and the directory `dir2`, in this order. Some examples of possible contents of the file `inc1` are:
  - `dir3` which would add `dir3` to the list of include directories
  - `((lib lib3) dir4 (include inc2))` which would add the source directory of the library `lib3`, the directory `dir4`, and the result of recursively including the contents of the file `inc2`. The contents of included directories are tracked recursively, e.g., if you use `(include_dir dir)` and have headers `dir/base.h` and `dir/lib/lib.h`, they both will be tracked as dependencies.
  - `extra_deps` specifies any other dependencies that should be tracked. This is useful when dealing with `#include` statements that escape into a parent directory like `#include "../a.h"`.

Mode-Dependent Stubs

Since Dune 3.5, it is possible to use different foreign stubs when building in *native* or *byte* mode. This feature needs to be activated by adding `(using mode_specific_stubs 0.1)` in the `dune-project` file.

Then it is allowed to use the `mode` field when describing `foreign_stubs`. If the same stub is defined twice, Dune will automatically chose the correct one. This allows the use of different sets of flags or even different source files from which the stubs are built.

```
(executable
 (name main)
 (modes native byte_complete)
 (foreign_stubs
   (language c)
   (mode byte)
   (names c_stubs))
 (foreign_stubs
   (language c)
   (mode native)
   (flags :standard -DNATIVE_CODE) ; A flag specific to native builds
   (names c_stubs)))  ; This could be the name of an implementation
                      ; specific to native builds
```

Note that, as of version `0.1` of this extension, this mechanism does not work for `foreign_archives`.

3.22.2 Foreign Archives

You can also specify archives of separately compiled foreign object files that need to be packaged with an OCaml library or linked into an OCaml executable. To do that, use the `foreign_archives` field of the corresponding `library` or `executable` stanza. For example:

```
(library
 (name lib)
 (foreign_stubs (language c) (names src1 src2))
 (foreign_stubs (language cxx) (names src3) (flags -O2))
 (foreign_archives arch1 some/dir/arch2))
```

Here, in addition to *Foreign Stubs*, we also specify foreign archives `arch1` and `arch2`, where the latter is stored in a subdirectory `some/dir`.

You can build a foreign archive manually, e.g., using a custom `rule` as described in *Foreign Build Sandboxing*, or ask Dune to build it via the `foreign_library` stanza:

```
(foreign_library
 (archive_name arch1)
 (language c)
 (enabled_if true)
 (names src4 src5)
 (include_dir headers))
```

This asks Dune to compile C source files `src4` and `src5` with headers tracked in the `headers` directory and put the resulting object files into an archive `arch1`, whose full name is typically `libarch1.a` for static linking and `dllarch1.so` for dynamic linking.

The `foreign_library` stanza supports all *Foreign Stubs* fields. The `archive_name` field specifies the archive's name. You can refer to the same archive name from multiple OCaml libraries and executables, so a foreign archive is a bit like a foreign library, hence the name of the stanza. The `enabled_if` field has the same meaning as in the *library* stanza.

Foreign archives are particularly useful when embedding a library written in a foreign language and/or built with another build system. See *Foreign Build Sandboxing* for more details.

3.22.3 Extra Objects

It's possible to specify native object files to be packaged with OCaml libraries or linked into OCaml executables. Do this by using the `extra_objects` field of the `library` or `executable` stanzas. For example:

```
(executable
 (public_name main)
 (extra_objects foo bar))

(rule
 (targets foo.o bar.o)
 (deps foo.c bar.c)
 (action (run ocamlopt %{deps})))
```

This example builds an executable which is linked against a pair of native object files, `foo.o` and `bar.o`. The `extra_objects` field takes a list of object names, which correspond to the object file names with their path and extension omitted.

In this example, the sources corresponding to the objects (`foo.c` and `bar.c`) are assumed to be present in the same directory as the OCaml source code, and a custom `rule` is used to compile the C source code into object files using

`ocamlopt`. This is not necessary; one can instead compile foreign object files manually and place them next to the OCaml source code.

3.22.4 Flags

Depending on the *use_standard_c_and_cxx_flags* option, the base *:standard* set of flags for C will contain only `ocamlc_cflags` or both `ocamlc_cflags` and `ocamlc_cppflags`.

There are multiple levels where one can declare custom flags (using the *Ordered Set Language*), and each level inherits the flags of the previous one in its *:standard* set:

- In the global `env` definition of a `dune-workspace` file
- In the per-context *env* definitions in a *dune-workspace* file
- In the env definition of a *dune* file
- In a *foreign_* field of an executable or a library

The `%{cc}` *variable* will contain the flags from the first three levels only.

3.23 Command Line Interface

This is a short overview of the commands available in Dune. Reference documentation for each command is available through `dune COMMAND --help`.

`dune build`
: Build the given targets, or the default ones.

`dune cache`
: Manage the shared cache of build artifacts.

  `dune cache size`
  : Query the size of the Dune cache.

  `dune cache trim`
  : Trim the Dune cache.

`dune clean`
: Clean the project.

`dune coq`
: Command group related to Coq.

  `dune coq top`
  : Execute a Coq toplevel with the local configuration.

`dune describe`
: Describe the workspace.

  `dune describe aliases`
  : Print aliases in a given directory. Works similarly to ls.

  `dune describe env`
  : Print the environment of a directory.

dune describe external-lib-deps

Print out the external libraries needed to build the project. It's an approximated set of libraries.

dune describe installed-libraries

Print out the libraries installed on the system.

dune describe opam-files

Print information about the opam files that have been discovered.

dune describe package-entries

prints information about the entries per package.

dune describe pp

Build a given file and print the preprocessed output.

dune describe rules

Dump rules.

dune describe targets

Print targets in a given directory. Works similarly to ls.

dune describe workspace

Print a description of the workspace's structure. If some directories are provided, then only those directories of the workspace are considered.

dune diagnostics

Fetch and return errors from the current build.

dune exec

Execute a command in a similar environment as if installation was performed.

dune fmt

Format source code.

dune format-dune-file

Format dune files.

dune help

Additional Dune help.

dune init

Command group for initializing Dune components.

dune init executable

Initialize a binary executable.

dune init library

Initialize an OCaml library.

dune init project

Initialize a whole OCaml project.

dune init test

Initialize a test harness.

dune install

Install packages defined in workspace.

dune installed-libraries

Print out libraries installed on the system.

dune ocaml

Command group related to OCaml.

dune ocaml dump-dot-merlin

Print Merlin configuration.

dune ocaml merlin

Command group related to Merlin.

dune ocaml merlin dump-config

Prints the entire content of the Merlin configuration for the given folder in a user friendly form.

dune ocaml merlin start-session

Start a Merlin configuration server.

dune ocaml ocaml-merlin

Start a Merlin configuration server.

dune ocaml top

Print a list of toplevel directives for including directories and loading `.cma` files.

dune ocaml top-module

Print a list of toplevel directives for loading a module into the toplevel.

dune ocaml utop

Load library in UTop.

dune ocaml-merlin

Start a Merlin configuration server.

dune printenv

Print the environment of a directory.

dune promotion

Control how changes are propagated back to source code.

dune promotion apply

Promote files from the last run.

dune promotion diff

List promotions to be applied.

dune promote

A command alias for `dune promotion apply`.

dune rpc

Dune's RPC mechanism. Experimental.

dune rules

Dump rules.

dune runtest

Run tests.

dune test

A command alias for `dune runtest`.

dune shutdown
: Cancel and shutdown any builds in the current workspace.

dune subst
: Substitute watermarks in source files.

dune top
: Print a list of toplevel directives for including directories and loading `.cma` files.

dune uninstall
: Uninstall packages defined in the workspace.

dune upgrade
: Upgrade projects across major Dune versions.

dune utop
: Load library in UTop.

3.24 Dune Libraries

3.24.1 Configurator

Configurator is a small library designed to query features available on the system in order to generate configuration for Dune builds. Such generated configuration is usually in the form of command line flags, generated headers, and stubs, but there are no limitations on this.

Configurator allows you to query for the following features:

- Variables defined in `ocamlc -config`,
- pkg-config flags for packages,
- Test features by compiling C code,
- Extract compile time information such as `#define` variables.

Configurator is designed to be cross-compilation friendly and avoids _running_ any compiled code to extract any of the information above.

Configurator started as an independent library, but now lives in dune. It is released as the package `dune-configurator`.

Usage

We'll describe configurator with a simple example. Everything else can be easily learned by studying configurator's API.

To use Configurator, write an executable that will query the system using Configurator's API and output a set of targets reflecting the results. For example:

```
module C = Configurator.V1

let clock_gettime_code = {|
#include <time.h>

int main()
```

(continues on next page)

(continued from previous page)

```
{
  struct timespec ts;
  clock_gettime(CLOCK_REALTIME, &ts);
  return 0;
}
|}

let () =
  C.main ~name:"foo" (fun c ->
    let has_clock_gettime =
      C.c_test c clock_gettime_code ~link_flags:["-lrt"] in

    C.C_define.gen_header_file c ~fname:"config.h"
      [ "HAS_CLOCK_GETTIME", Switch has_clock_gettime ]);
```

Usually, the module above would be named `discover.ml`. The next step is to invoke it as an executable and tell Dune about the targets that it produces:

```
(executable
 (name discover)
 (libraries dune-configurator))

(rule
 (targets config.h)
 (action (run ./discover.exe)))
```

Another common pattern is to produce a flags file with Configurator and then use this flag file using `:include`:

```
(library
 (name mylib)
 (foreign_stubs (language c) (names foo))
 (c_library_flags (:include (flags.sexp))))
```

For this, generate the list of flags for your library (for example, using `Configurator.V1.Pkg_config`), and then write them to a file: in the above example, `flags.sexp` with `Configurator.V1.write_flags "flags.sexp" flags`.

Upgrading From the Old Configurator

The old Configurator is the independent Configurator opam package. It's now deprecated, and users are encouraged to migrate to Dune's own Configurator. The advantage of the transition include:

- No extra dependencies,
- No need to manually pass `-ocamlc` flag,
- New Configurator is cross-compilation compatible.

The following steps must be taken to transition from the old Configurator:

- Mentions of the `configurator` opam package should be replaced with `dune-configurator`.
- The library name `configurator` should be changed `dune-configurator`.
- The `-ocamlc` flag in rules that runs Configurator scripts should be removed. This information is now passed automatically by Dune.

- The new Configurator API is versioned explicitly. The version that's compatible with old Configurator is under the `V1` module. Hence, to transition one's code, it's enough to add this module alias:

```
module Configurator = Configurator.V1
```

3.24.2 *dune-build-info* Library

Dune can embed build information such as versions in executables via the special `dune-build-info` library. This library exposes some information about how the executable was built, such as the version of the project containing the executable or the list of statically linked libraries with their versions. Printing the version at which the current executable was built is as simple as:

```
Printf.printf "version: %s\n"
  (match Build_info.V1.version () with
   | None -> "n/a"
   | Some v -> Build_info.V1.Version.to_string v)
```

For libraries and executables from development repositories that don't have version information written directly in the `dune-project` file, the version is obtained by querying the version control system. For instance, the following Git command is used in Git repositories:

```
$ git describe --always --dirty --abbrev=7
```

which produces a human readable version string of the form `<version>-<commits-since-version>-<hash>[-dirty]`.

Note that in the case where the version string is obtained from the version control system, the version string will only be written in the binary once it's installed or promoted to the source tree. In particular, if you evaluate this expression as part of your package build, it will return `None`. This ensures that committing doesn't hurt your development experience. Indeed, if Dune stored the version directly inside the freshly built binaries, then every time you commit your code, the version would change and Dune would need to rebuild all the binaries and everything that depends on them, such as tests. Instead, Dune leaves a placeholder inside the binary and fills it during installation or promotion.

3.24.3 (Experimental) Dune Action Plugin

This library is experimental and no backwards compatibility is implied. Use at your own risk.

`Dune-action-plugin` provides a monadic interface to express program dependencies directly inside the source code. Programs using this feature should be declared using *dynamic-run* instead of usual *run*.

3.25 Dune Cache

Dune implements a cache of build results that is shared across different workspaces. Before executing a build rule, Dune looks it up in the shared cache, and if it finds a matching entry, Dune skips the rule's execution and restores the results in the current build directory. This can greatly speed up builds when different workspaces share code, as well as when switching branches or simply undoing some changes within the same workspace.

3.25.1 Configuration

For now, Dune cache is an opt-in feature. There are three ways to enable it. Choose the one that is more convenient for you:

- Add `(cache enabled)` to your Dune configuration file (`~/.config/dune/config` by default).
- Set the environment variable `DUNE_CACHE` to `enabled`
- Run Dune with the `--cache=enabled` flag.

By default, Dune stores the cache in your `XDG_CACHE_HOME` directory on *nix systems and `%LOCALAPPDATA%\Microsoft\Windows\Temporary Internet Files\dune` on Windows. You can change the default location by setting the environment variable `DUNE_CACHE_ROOT`.

3.25.2 Cache Storage Mode

Dune supports two modes of storing and restoring cache entries: *hardlink* and *copy*. If your file system supports hard links, we recommend that you use the *hardlink* mode, which is generally more efficient and reliable.

The *hardlink* Mode

By default, Dune uses hard links when storing and restoring cache entries. This is fast and has zero disk space overhead for files that still live in a build directory. There are two disadvantages of this mode:

- The cache storage must be on the same partition as the build tree.
- A cache entry can be corrupted by modifying the hard link that points to it from the build directory. To reduce the risk of cache corruption, Dune systematically removes write permissions from all build results. It is worth noting that modifying files in the build directory is a bad practice anyway.

The *copy* Mode

If you specify `(cache-storage-mode copy)` in the configuration file, Dune will copy files to and from the cache instead of using hard links. This mode is slower and has higher disk space usage. On the positive side, it is more portable and doesn't have the disadvantages of the *hardlink* mode (see above).

You can also set or override the storage mode via the environment variable `DUNE_CACHE_STORAGE_MODE` and the command line flag `--cache-storage-mode`.

3.25.3 Trimming the Cache

Storing all historically produced build results in the cache is infeasible, so you'll need to occasionally trim the cache. To do that, run the `dune cache trim --size=BYTES` command. This will remove the oldest used cache entries to keep the cache overhead below the specified size. By "overhead" we mean the cache entries whose hard link count is equal to 1, i.e., which aren't used in any build directory. Trimming cache entries whose hard link count is greater than 1 would not free any disk space.

Note that previous versions of Dune, cache provided a "cache daemon" that could periodically trim the cache. The current version doesn't require an additional daemon process, so this automated trimming functionality is no longer provided.

3.25.4 Reproducibility

Reproducibility Check

While the main purpose of Dune cache is to speed up build times, it can also be used to check build reproducibility. By specifying `(cache-check-probability FLOAT)` in the configuration file, or running Dune with the `--cache-check-probability=FLOAT` flag, you instruct Dune to re-execute randomly chosen build rules and compare their results with those stored in the cache. If the results differ, the rule is not reproducible, and Dune will print out a corresponding warning.

Non-Reproducible Rules

Some build rules are inherently not reproducible because they involve running non-deterministic commands that, for example, depend on the current time or download files from the Internet. To prevent Dune from caching such rules, mark them as non-reproducible by using `(deps (universe))`. Please see *Dependency Specification*.

3.26 Coq

Table of Contents

- *Coq*
  - *Introduction*
  - *coq.theory*
    - *Coq Dependencies*
    - *Coq Documentation*
    - *Recursive Qualification of Modules*
    - *How Dune Locates and Builds theories*
    - *Public and Private Theories*
    - *Limitations*
    - *Coq Language Version*
    - *Coq Language Version 1.0*
  - *coq.extraction*
  - *coq.pp*
  - *Examples of Coq Projects*
    - *Simple Project*
    - *Multi-Theory Project*
    - *Composing Projects*
    - *Composing With Installed Theories*
    - *Building Documentation*
    - *Coq Plugin Project*

- *Running a Coq Toplevel*
  - *Limitations*
- *Coq-Specific Variables*
- *Coq Environment Fields*

3.26.1 Introduction

Dune can build Coq theories and plugins with additional support for extraction and `.mlg` file preprocessing.

A *Coq theory* is a collection of `.v` files that define Coq modules whose names share a common prefix. The module names reflect the directory hierarchy.

Coq theories may be defined using *coq.theory* stanzas, or be auto-detected by Dune by inspecting Coq's install directories.

A *Coq plugin* is an OCaml *library* that Coq can load dynamically at runtime. Plugins are typically linked with the Coq OCaml API.

Since Coq 8.16, plugins need to be "public" libraries in Dune's terminology, that is to say, they must declare a `public_name` field.

A *Coq project* is an informal term for a *dune-project* containing a collection of Coq theories and plugins.

The `.v` files of a theory need not be present as source files. They may also be Dune targets of other rules.

To enable Coq support in a Dune project, specify the *Coq language version* in the *dune-project* file. For example, adding

```
(using coq 0.8)
```

to a *dune-project* file enables using the `coq.theory` stanza and other `coq.*` stanzas. See the *Dune Coq language* section for more details.

3.26.2 coq.theory

The Coq theory stanza is very similar in form to the OCaml *library* stanza:

```
(coq.theory
 (name <module_prefix>)
 (package <package>)
 (synopsis <text>)
 (modules <ordered_set_lang>)
 (plugins <ocaml_plugins>)
 (flags <coq_flags>)
 (modules_flags <flags_map>)
 (coqdoc_flags <coqdoc_flags>)
 (stdlib <stdlib_included>)
 (mode <coq_native_mode>)
 (theories <coq_theories>))
```

The stanza builds all the `.v` files in the given directory and its subdirectories if the *include-subdirs* stanza is present.

For usage of this stanza, see the *Examples of Coq Projects*.

The semantics of the fields are:

- <module_prefix> is a dot-separated list of valid Coq module names and determines the module scope under which the theory is compiled (this corresponds to Coq's -R option).

  For example, if <module_prefix> is foo.Bar, the theory modules are named foo.Bar.module1, foo.Bar.module2, etc. Note that modules in the same theory don't see the foo.Bar prefix in the same way that OCaml wrapped libraries do.

  For compatibility, *Coq lang 1.0* installs a theory named foo.Bar under foo/Bar. Also note that Coq supports composing a module path from different theories, thus you can name a theory foo.Bar and a second one foo.Baz, and Dune composes these properly. See an example of *a multi-theory* Coq project for this.

- The modules field allows one to constrain the set of modules included in the theory, similar to its OCaml counterpart. Modules are specified in Coq notation. That is to say, A/b.v is written A.b in this field.

- If the package field is present, Dune generates install rules for the .vo files of the theory. pkg_name must be a valid package name.

  Note that *Coq lang 1.0* will use the Coq legacy install setup, where all packages share a common root namespace and install directory, lib/coq/user-contrib/<module_prefix>, as is customary in the Make-based Coq package ecosystem.

  For compatibility, Dune also installs, under the user-contrib prefix, the .cmxs files that appear in <ocaml_plugins>. This will be dropped in future versions.

- <coq_flags> are passed to coqc as command-line options. :standard is taken from the value set in the (coq (flags <flags>)) field in env profile. See *env* for more information.

- <flags_map> is a list of pairs of valid Coq module names and a list of <coq_flags>. Note that if a module is present here, the :standard variable will be bound to the value of <coq_flags> effective for the theory. This way it is possible to override the default flags for particular files of the theory, for example:

```
(coq.theory
  (name Foo)
  (modules_flags
    (bar (:standard \ -quiet))))
```

  It is more common to just use this field to *add* some particular flags, but that should be done using (:standard <flag1> <flag2> ...) as to propagate the default flags. (Appeared in *Coq lang 0.9*)

- <coqdoc_flags> are extra user-configurable flags passed to coqdoc. The default value for :standard is --toc. The --html or --latex flags are passed separately depending on which mode is target. See the section on *documentation using coqdoc* for more information.

- <stdlib_included> can either be yes or no, currently defaulting to yes. When set to no, Coq's standard library won't be visible from this theory, which means the Coq prefix won't be bound, and Coq.Init.Prelude won't be imported by default.

- If the plugins field is present, Dune will pass the corresponding flags to Coq so that coqdep and coqc can find the corresponding OCaml libraries declared in <ocaml_plugins>. This allows a Coq theory to depend on OCaml plugins. Starting with (lang coq 0.6), <ocaml_plugins> must contain public library names.

- Your Coq theory can depend on other theories — globally installed or defined in the current workspace — by adding the theories names to the <coq_theories> field. Then, Dune will ensure that the depended theories are present and correctly registered with Coq.

  See *Locating Theories* for more information on how Coq theories are located by Dune.

- If Coq has been configured with -native-compiler yes or ondemand, Dune will always build the cmxs files together with the vo files. This only works on Coq versions after 8.13 in which the option was introduced.

  You may override this by specifying (mode native) or (mode vo).

Before *Coq lang 0.7*, the native mode had to be manually specified, and Coq did not use Coq's configuration

Versions of Dune < 3.7.0 would disable native compilation if the `dev` profile was selected.

- If the `(mode vos)` field is present, only Coq compiled interface files `.vos` will be produced for the theory. This is mainly useful in conjunction with `dune coq top`, since this makes the compilation of dependencies much faster, at the cost of skipping proof checking. (Appeared in *Coq lang 0.8*).

Coq Dependencies

When a Coq file `a.v` depends on another file `b.v`, Dune is able to build them in the correct order, even if they are in separate theories. Under the hood, Dune asks coqdep how to resolve these dependencies, which is why it is called once per theory.

Coq Documentation

Given a *coq.theory* stanza with `name A`, Dune will produce two *directory targets*, `A.html/` and `A.tex/`. HTML or LaTeX documentation for a Coq theory may then be built by running `dune build A.html` or `dune build A.tex`, respectively (if the *dune file* for the theory is the current directory).

There are also two aliases *@doc* and `@doc-latex` that will respectively build the HTML or LaTeX documentation when called. These will determine whether or not Dune passes a `--html` or `--latex` flag to `coqdoc`.

Further flags can also be configured using the `(coqdoc_flags)` field in the `coq.theory` stanza. These will be passed to `coqdoc` and the default value is `:standard` which is `--toc`. Extra flags can therefore be passed by writing `(coqdoc_flags :standard --body-only)` for example.

Recursive Qualification of Modules

If you add:

```
(include_subdirs qualified)
```

to a *dune* file, Dune considers all the modules in the directory and its subdirectories, adding a prefix to the module name in the usual Coq style for subdirectories. For example, file `A/b/C.v` becomes the module `A.b.C`.

How Dune Locates and Builds theories

Dune organises it's knowledge about Coq theories in 3 databases:

- Scope database: A Dune *scope* is a part of the project sharing a single common `dune-project` file. In a single scope, any theory in the database can depend on any other theory in that database as long as their visibilities are compatible. A public theory for example cannot depend on a private theory.
- Public theory database: The set of all scopes that Dune knows about is termed a *workspace*. Only public theories coming from scopes are added to the database of all public theories in the current workspace.

  The public theory database allows theories to depend on theories that are in a different scope. Thus, you can depend on theories belonging to another *dune-project* as long as they share a common scope under another *dune-project* file or a *dune-workspace* file.

  Doing so is usually as simple as placing a Coq project within the scope of another. This process is termed *composition*. See the *interproject composition* example.

  Inter-project composition allows Dune to compute module dependencies using a fine granularity. In practice, this means that Dune will only build the parts of a depended theory that are needed by your project.

Inter-project composition has been available since *Coq lang 0.4*.

- Installed theory database: If a theory cannot be found in the list of workspace-public theories, Dune will try to locate the theory in the list of installed locations Coq knows about.

  This list is built using the output of `coqc --config` in order to infer the `COQLIB` and `COQPATH` environment variables. Each path in `COQPATH` and `COQLIB/user-contrib` is used to build the database of installed theories.

  Note that, for backwards compatibility purposes, installed theories do not have to be installed or built using Dune. Dune tries to infer the name of the theory from the installed layout. This is ambiguous in the sense that a file-system layout of *a/b* will provide theory names `a` and `a.b`.

  Resolving this ambiguity in a backwards-compatible way is not possible, but future versions of Dune Coq support will provide a way to improve this.

  Coq's standard library gets a special status in Dune. The location at `COQLIB/theories` will be assigned a entry with the theory name `Coq`, and added to the dependency list implicitly. This can be disabled with the `(stdlib no)` field in the `coq.theory` stanza.

  The `Coq` prefix can then be used to depend on Coq's stdlib in a regular, qualified way. We recommend setting `(stdlib no)` and adding `(theories Coq)` explicitly.

  Composition with installed theories has been available since *Coq lang 0.8*.

The databases above are used to locate a theory dependencies. Note that Dune has a complete global view of every file involved in the compilation of your theory and will therefore rebuild if any changes are detected.

Public and Private Theories

A *public theory* is a *coq.theory* stanza that is visible outside the scope of a *dune-project* file.

A *private theory* is a *coq.theory* stanza that is limited to the scope of the *dune-project* file it is in.

A private theory may depend on both private and public theories; however, a public theory may only depend on other public theories.

By default, all *coq.theory* stanzas are considered private by Dune. In order to make a private theory into a public theory, the `(package )` field must be specified.

```
(coq.theory
 (name private_theory))

(coq.theory
 (name private_theory)
 (package coq-public-theory))
```

Limitations

- `.v` files always depend on the native OCaml version of the Coq binary and its plugins, unless the natively compiled versions are missing.
- A `foo.mlpack` file must the present in directories of locally defined plugins for things to work. `coqdep`, which is used internally by Dune, will recognize a plugin by looking at the existence of an `.mlpack` file, as it cannot access (for now) Dune's library database. This is a limitation of `coqdep`. See the *example plugin* or the this template.

  This limitation will be lifted soon, as newer versions of `coqdep` can use findlib's database to check the existence of OCaml libraries.

Coq Language Version

The Coq lang can be modified by adding the following to a *dune-project* file:

```
(using coq 0.8)
```

The supported Coq language versions (not the version of Coq) are:

- `0.9`: Support for per-module flags with the `(module_flags ...)` ` field.
- `0.8`: Support for composition with installed Coq theories; support for `vos` builds.

Deprecated experimental Coq language versions are:

- `0.1`: Basic Coq theory support.
- `0.2`: Support for the `theories` field and composition of theories in the same scope.
- `0.3`: Support for `(mode native)` requires Coq >= 8.10 (and Dune >= 2.9 for Coq >= 8.14).
- `0.4`: Support for interproject composition of theories.
- `0.5`: `(libraries ...)` field deprecated in favor of `(plugins ...)` field.
- `0.6`: Support for `(stdlib no)`.
- `0.7`: `(mode )` is automatically detected from the configuration of Coq and `(mode native)` is deprecated. The `dev` profile also no longer disables native compilation.

Coq Language Version 1.0

Guarantees with respect to stability are not yet provided, but we intend that the `(0.8)` version of the language becomes `1.0`. The `1.0` version of Coq lang will commit to a stable set of functionality. All the features below are expected to reach `1.0` unchanged or minimally modified.

3.26.3 coq.extraction

Coq may be instructed to *extract* OCaml sources as part of the compilation process by using the `coq.extraction` stanza:

```
(coq.extraction
 (prelude <name>)
 (extracted_modules <names>)
 <optional-fields>)
```

- `(prelude <name>)` refers to the Coq source that contains the extraction commands.
- `(extracted_modules <names>)` is an exhaustive list of OCaml modules extracted.
- `<optional-fields>` are `flags`, `stdlib`, `theories`, and `plugins`. All of these fields have the same meaning as in the `coq.theory` stanza.

The extracted sources can then be used in `executable` or `library` stanzas as any other sources.

Note that the sources are extracted to the directory where the `prelude` file lives. Thus the common placement for the `OCaml` stanzas is in the same *dune* file.

Warning: using Coq's `Cd` command to work around problems with the output directory is not allowed when using extraction from Dune. Moreover the `Cd` command has been deprecated in Coq 8.12.

3.26.4 coq.pp

Authors of Coq plugins often need to write `.mlg` files to extend the Coq grammar. Such files are preprocessed with the `coqpp` binary. To help plugin authors avoid writing boilerplate, we provide a `(coq.pp ...)` stanza:

```
(coq.pp
 (modules <ordered_set_lang>))
```

This will run the `coqpp` binary on all the `.mlg` files in `<ordered_set_lang>`.

3.26.5 Examples of Coq Projects

Here we list some examples of some basic Coq project setups in order.

Simple Project

Let us start with a simple project. First, make sure we have a *dune-project* file with a *Coq lang* stanza present:

```
(lang dune 3.17)
(using coq 0.8)
```

Next we need a *dune* file with a *coq.theory* stanza:

```
(coq.theory
 (name myTheory))
```

Finally, we need a Coq `.v` file which we name `A.v`:

```
(** This is my def *)
Definition mydef := nat.
```

Now we run `dune build`. After this is complete, we get the following files:

```
.
├── A.v
├── _build
│   ├── default
│   │   ├── A.glob
│   │   ├── A.v
│   │   └── A.vo
│   └── log
├── dune
└── dune-project
```

Multi-Theory Project

Here is an example of a more complicated setup:

```
.
├── A
│   ├── AA
│   │   └── aa.v
│   ├── AB
│   │   └── ab.v
│   └── dune
├── B
│   ├── b.v
│   └── dune
└── dune-project
```

Here are the *dune* files:

```
; A/dune
(include_subdirs qualified)
(coq.theory
 (name A))

; B/dune
(coq.theory
 (name B)
 (theories A))
```

Notice the `theories` field in `B` allows one *coq.theory* to depend on another. Another thing to note is the inclusion of the *include_subdirs* stanza. This allows our theory to have *multiple subdirectories*.

Here are the contents of the `.v` files:

```
(* A/AA/aa.v is empty *)

(* A/AB/ab.v *)
Require Import AA.aa.

(* B/b.v *)
From A Require Import AB.ab.
```

This causes a dependency chain `b.v -> ab.v -> aa.v`. Now we might be interested in building theory `B`, so all we have to do is run `dune build B`. Dune will automatically build the theory `A` since it is a dependency.

Composing Projects

To demonstrate the composition of Coq projects, we can take our previous two examples and put them in project which has a theory that depends on theories in both projects.

```
.
├── CombinedWork
│   ├── comb.v
│   └── dune
├── DeeperTheory
```

(continues on next page)

(continued from previous page)

```
    ├── A
    │   ├── AA
    │   │   └── aa.v
    │   ├── AB
    │   │   └── ab.v
    │   └── dune
    ├── B
    │   ├── b.v
    │   └── dune
    ├── Deep.opam
    └── dune-project
├── dune-project
└── SimpleTheory
    ├── A.v
    ├── dune
    ├── dune-project
    └── Simple.opam
```

The file `comb.v` looks like:

```
(* Files from DeeperTheory *)
From A.AA Require Import aa.
(* In Coq, partial prefixes for theory names are enough *)
From A Require Import ab.
From B Require Import b.

(* Files from SimpleTheory *)
From myTheory Require Import A.
```

We are referencing Coq modules from all three of our previously defined theories.

Our *dune* file in `CombinedWork` looks like:

```
(coq.theory
 (name Combined)
 (theories myTheory A B))
```

As you can see, there are dependencies on all the theories we mentioned.

All three of the theories we defined before were *private theories*. In order to depend on them, we needed to make them *public theories*. See the section on *Public and Private Theories*.

Composing With Installed Theories

We can also compose with theories that are installed. If we wanted to have a theory that depends on the Coq theory `mathcomp.ssreflect` we can add the following to our stanza:

```
(coq.theory
 (name my_mathcomp_theory)
 (theories mathcomp.ssreflect))
```

Note that `mathcomp` on its own would also work, since there would be a `matchcomp` directory in `user-contrib`, however it would not compose locally with a `coq.theory` stanza with the `mathcomp.ssreflect` name (in case one

exists). So it is advisable to use the actual theory name. Dune is not able to validate theory names that have been installed since they do not include their Dune metadata.

Building Documentation

Following from our last example, we might wish to build the HTML documentation for `A`. We simply do `dune build A/A.html/`. This will produce the following files:

```
A
├── AA
│   ├── aa.glob
│   ├── aa.v
│   └── aa.vo
├── AB
│   ├── ab.glob
│   ├── ab.v
│   └── ab.vo
└── A.html
    ├── A.AA.aa.html
    ├── A.AB.ab.html
    ├── coqdoc.css
    ├── index.html
    └── toc.html
```

We may also want to build the LaTeX documentation of the theory `B`. For this we can call `dune build B/B.tex/`. If we want to build all the HTML documentation targets, we can use the *@doc* alias as in `dune build @doc`. If we want to build all the LaTeX documentation then we use the `@doc-latex` alias instead.

Coq Plugin Project

Let us build a simple Coq plugin to demonstrate how Dune can handle this setup.

```
.
├── dune-project
├── src
│   ├── dune
│   ├── hello_world.ml
│   ├── my_plugin.mlpack
│   └── syntax.mlg
└── theories
    ├── dune
    └── UsingMyPlugin.v
```

Our *dune-project* will need to have a package for the plugin to sit in, otherwise Coq will not be able to find it.

```
(lang dune 3.17)
(using coq 0.8)

(package
 (name my-coq-plugin)
 (synopsis "My Coq Plugin")
 (depends coq-core))
```

Now we have two directories, `src/` and `theories/` each with their own *dune* file. Let us begin with the plugin *dune* file:

```
(library
 (name my_plugin)
 (public_name my-coq-plugin.plugin)
 (synopsis "My Coq Plugin")
 (flags :standard -rectypes -w -27)
 (libraries coq-core.vernac))

(coq.pp
 (modules syntax))
```

Here we define a library using the *library* stanza. Importantly, we declared which external libraries we rely on and gave the library a `public_name`, as starting with Coq 8.16, Coq will identify plugins using their corresponding findlib public name.

The *coq.pp* stanza allows `src/syntax.mlg` to be preprocessed, which for reference looks like:

```
DECLARE PLUGIN "my-coq-plugin.plugin"

VERNAC COMMAND EXTEND CallToC CLASSIFIED AS QUERY
| [ "Hello" ] -> { Feedback.msg_notice Pp.(str Hello_world.hello_world) }
END
```

Together with `hello_world.ml`:

```
let hello_world = "hello world!"
```

They make up the plugin. There is one more important ingredient here and that is the `my_plugin.mlpack` file, needed to signal `coqdep` the existence of `my_plugin` in this directory. An empty file suffices. See *this note on .mlpack files*.

The file for `theories/` is a standard *coq.theory* stanza with an included `libraries` field allowing Dune to see `my-coq-plugin.plugin` as a dependency.

```
(coq.theory
 (name MyPlugin)
 (package my-coq-plugin)
 (plugins my-coq-plugin.plugin))
```

Finally, our .v file will look something like this:

```
(* For Coq < 8.16 *)
Declare ML Module "my_plugin".

(* For Coq = 8.16 *)
Declare ML Module "my_plugin:my-coq-plugin.plugin".

(* At some point Coq 8.17 or 8.18 will transition to the syntax below, check Coq's manual␣
↪*)
Declare ML Module "my-coq-plugin.plugin".

Hello.
```

Running `dune build` will build everything correctly.

3.26.6 Running a Coq Toplevel

Dune supports running a Coq toplevel binary such as `coqtop`, which is typically used by editors such as CoqIDE or Proof General to interact with Coq.

The following command:

```
$ dune coq top <file> -- <args>
```

runs a Coq toplevel (`coqtop` by default) on the given Coq file `<file>`, after having recompiled its dependencies as necessary. The given arguments `<args>` are forwarded to the invoked command. For example, this can be used to pass a `-emacs` flag to `coqtop`.

A different toplevel can be chosen with `dune coq top --toplevel CMD <file>`. Note that using `--toplevel echo` is one way to observe what options are actually passed to the toplevel. These options are computed based on the options that would be passed to the Coq compiler if it was invoked on the Coq file `<file>`.

In certain situations, it is desirable to not rebuild dependencies for a `.v` files but still pass the correct flags to the toplevel. For this reason, a `--no-build` flag can be passed to `dune coq top` which will skip any building of dependencies.

Limitations

- Only files that are part of a stanza can be loaded in a Coq toplevel.
- When a file is created, it must be written to the file system before the Coq toplevel is started.
- When new dependencies are added to a file (via a Coq `Require` vernacular command), it is in principle required to save the file and restart to Coq toplevel process.

3.26.7 Coq-Specific Variables

There are some special variables that can be used to access data about the Coq configuration. These are:

- `%{coq:version}` the version of Coq.
- `%{coq:version.major}` the major version of Coq (e.g., `8.15.2` gives `8`).
- `%{coq:version.minor}` the minor version of Coq (e.g., `8.15.2` gives `15`).
- `%{coq:version.suffix}` the suffix version of Coq (e.g., `8.15.2` gives `.2` and `8.15+rc1` gives `+rc1`).
- `%{coq:ocaml-version}` the version of OCaml used to compile Coq.
- `%{coq:coqlib}` the output of `COQLIB` from `coqc -config`.
- `%{coq:coq_native_compiler_default}` the output of `COQ_NATIVE_COMPILER_DEFAULT` from `coqc -config`.

See *Variables* for more information on variables supported by Dune.

3.26.8 Coq Environment Fields

The *env* stanza has a `(coq <coq_fields>)` field with the following values for `<coq_fields>`:

- `(flags <flags>)`: The default flags passed to `coqc`. The default value is `-q`. Values set here become the `:standard` value in the `(coq.theory (flags <flags>))` field.
- `(coqdoc_flags <flags>)`: The default flags passed to `coqdoc`. The default value is `--toc`. Values set here become the `:standard` value in the `(coq.theory (coqdoc_flags <flags>))` field.

3.27 Dune RPC

Starting from dune 3.0, dune's watch mode also runs an RPC server. This is a general mechanism introduced to integrate with various dune functionality. Some use cases we have in mind are:

- Text editors that would like to receive dune's error reporting
- In the future, custom actions that need to communicate with dune

There's no fixed scope for RPC, and we encourage users to submit requests to cover more functionality.

The purpose of this documentation is to explain how RPC works, and how to connect to it as a client. More concrete information such as what requests are available is in Dune_rpc

3.27.1 *dune-rpc* library

We provide a client library `dune-rpc` to make it easy to write clients. The library has a versioned interface and we guarantee to maintain its stability. The library documentation describes the guarantees in more detail.

RPC clients written with this library are guaranteed to work for all versions of the RPC server greater than it. In other words, a client using version `X` will work with all servers `>= X`. However, it will not work with any servers `< X`.

The library contains a client implementation parameterised over a concurrency monad. The `dune-rpc-lwt` package provides an instantiation of the RPC client defined in `dune-rpc` using lwt to implement the concurrency monad. It is of course possible to instantiate the dune RPC client using alternative concurrency implementations.

The `dune-rpc` library provides an API to do the following:

- Initialise an RPC session over a channel
- Send RPC requests and notifications
- Handle notifications received from the server
- Definitions of available requests, notifications, and their associated types.
- Subscribe to streams of build progress events

Here is an example of a simple Dune RPC client that uses the `dune-rpc` and `dune-rpc-lwt` packages.

3.27.2 Connecting

To connect to Dune's RPC server, it needs to be started in watch mode. It is possible to use `dune build --passive-watch-mode` to start an RPC server which will listen for requests without starting a build by itself. Then `dune rpc build .` will connect to it, trigger a build, and report status.

3.28 Packages

Packages are the units that are made installable. Executables and libraries that have a `(public_name)` are installable, and need to be attached to a package.

3.28.1 How Dune Knows About Packages

Packages can be defined in two ways:

- If a `pkg.opam` file is found, it signals the existence of the `pkg` package to Dune. The contents of the opam file are not interpreted by Dune.
- In `dune-project`, a `(package)` stanza with `(name pkg)` defines the `pkg` package.

Note: If you want to define public elements but are not interested in publishing to opam, it is not necessary to write empty `*.opam` files. Instead, prefer declaring it in `dune-project` as a `(package)` with just a `(name)`.

3.28.2 Generating Opam Files

In the latter case, if `(generate_opam_files)` is present in `(dune-project)`, Dune will generate `pkg.opam` using the metadata in found in `(package)` stanzas and global fields such as `(source)`.

See also:

How to Generate Opam Files from dune-project

If a `pkg.opam.template` file is found, the fields found in this opam file are added to the generated file, overriding the generated ones in case of collision. This can be used as an escape hatch to add arbitrary opam fields.

3.28.3 What Dune Installs

Dune will install the following files:

- `(library)` stanzas which have a `(public_name)`. The package name is the first component of the dot-separated public name: `(public_name pkg)` and `(public_name pkg.sub)` will look for a package named `pkg` to install their contents.
- `(executable)` and `(test)` stanzas with a `(public_name)`. The package to install the files in is determined by the `(package)` field, which can be omitted if there is only one package.
- `(install)` stanzas provide a low-level mechanism to install arbitrary files.
- Following the conventions of odig, files named `README*`, `CHANGE*`, `HISTORY*` and `LICENSE*` are installed in the packages defined by opam files in the same directory. These files can be generated by user rules, So for instance, a changelog generated by a user rule will be automatically installed as well.

3.29 Findlib Integration

Dune integrates with findlib so that it is possible to use a dependency built with Dune in a project that does not use Dune, or vice versa.

See also:

The OCaml Ecosystem explains the role of findlib and its relation with Dune.

To do so, Dune both interprets and generates `META` files.

3.29.1 How Dune Interprets `META` files

`META` files use the concept of *predicates*, which can be used to change the interpretation of the directives the files contain. However, Dune does not expose this to the user.

Instead, Dune interprets `META` files assuming the following set of predicates:

- `mt`: refers to a library that can be used with or without threads. Dune will force the threaded version.
- `mt_posix`: forces the use of POSIX threads rather than VM threads. VM threads are deprecated and will soon be obsolete.
- `ppx_driver`: when a library acts differently depending on whether it's linked as part of a driver or meant to add a `-ppx` argument to the compiler, choose the former behavior.

3.29.2 The Special Case of the OCaml Compiler

Libraries installed by the compiler are a special case: when the OCaml compiler is older than version 5.0, it does not include `META` files. In that situation, Dune uses its own internal database.

3.29.3 How Dune Generates `META` Files

When Dune builds a library, it generates a corresponding `META` file automatically. Usually, there is nothing to do. However, for the rare cases where a specific `META` file is needed, or to ease the transition of a project to Dune, it is possible to write/generate a specific one.

If a `META.<package>.template` is found in the same directory as `<package>.OPAM`, the generated `META.<package>` file will be produced by starting from the contents of `META.<package>.template` and replacing lines of the form `# DUNE_GEN` with the contents of the `META` it would normally generate.

For instance, to add `field = "..."` to the generated `META` file of package `pkg`, you can create a file named `META.pkg.template` with the following contents:

```
# DUNE_GEN
blah = "..."
```

CHAPTER

FOUR

EXPLANATION

These documents explain how certain feature works, or how Dune integrates with the rest of the OCaml ecosystem.

4.1 How Preprocessing Works

Preprocessing consists in transforming source code before it is compiled. The goal of this document is to explain how this works in Dune.

Dune supports two separate ways of applying preprocessors, the "classic pipeline" (used with `(staged_pps)`), and the "fast pipeline" (used for all other *preprocessing specifications* including `(pps)`).

The OCaml compilers provide options for specifying a preprocessing step. The `-pp` option is used to invoke a textual preprocessor (something that reads text and returns text). The `-ppx` option is used to invoke a *ppx rewriter* (a function that takes an AST and outputs an AST).

This is the "classic pipeline": preprocessing is part of the compilation itself. This is simple, but has a problem: in order to compute the dependencies of a module, it is necessary to pass the same `-pp` or `-ppx` option to `ocamldep`.

The classic pipeline has the following steps:

- preprocessing (as part of `ocamldep`)
- dependency analysis
- preprocessing (as part of compilation)
- compilation

Dune supports a "fast pipeline" where the preprocessor is invoked separately from the compiler and its output is saved. Afterwards the preprocessed code is compiled directly.

The fast pipeline has the following steps:

- preprocessing
- dependency analysis
- compilation

It has several advantages: it only invokes the preprocessor once per file, and the preprocessed code is reused between dependency analysis and different kinds of compilation. Also, when several preprocessors use `ppxlib`, they can be combined in a preprocessing program that traverses the AST only once.

However, some specific code generators or preprocessors require direct access to the compilation artefacts of their dependencies. Therefore they need to be used with the classic pipeline, even if it is slower. Note that a PPX is able to know if it was called as part of `ocamldep -ppx` or `ocamlopt -ppx`, so it can act differently in each phase.

Dune chooses which pipeline to use depending on the provided *Preprocessing Specification*. It will select the fast pipeline, unless `(staged_pps)` is used. In that case, the classic pipeline is used.

In the case of the fast pipeline, a single executable is built and accepts arguments for all preprocessors.

4.2 The OCaml Ecosystem

The OCaml ecosystem is not monolithic: the compiler and tools are not maintained by the same entities. As such, it can be difficult to understand the history and roles of the various pieces of this ecosystem. The goal of this page is to give a quick overview of the situation and the role that Dune plays in it.

4.2.1 The OCaml Compiler Distribution: Compiling and Linking

The OCaml compiler distribution contains "core" tools including the compilers (`ocamlc` and `ocamlopt`). They turn source files (with extensions `.ml` and `.mli`) into executables and libraries. Dependencies between compiled objects only exist at the module level, so this is a low-level tool.

4.2.2 Findlib: Metadata for Libraries

Findlib is a tool that defines the concept of library, so that libraries can depend on other libraries on top of the notion of module. Definitions of libraries, and other pieces of metadata, are stored in `META` files.

Findlib ships an executable named `ocamlfind` that can be used as a wrapper on top of the compilers to perform tasks such as producing an executable from compiled object files and external libraries.

4.2.3 Opam: a Collection of Software Projects

Opam is a package manager. It is used to determine which packages are necessary, and how to fetch and build them. Packages can contain libraries, executables, and other kinds of files.

The notion of version is specific to opam. If your project uses a function named `Png.read_file` but this function has been added only in version `1.2.0` of that package, opam needs to know about it.

Opam manages collections of installed packages, called switches. Using your project's dependencies (names and version constraints), it is able to create a switch that you'll be using to develop your project.

Public definitions of packages are available in a database called `opam-repository` which is maintained as a public Git repository. Publishing a package on opam (to make sure that external users can use your project) consists in adding its definition to `opam-repository`.

4.2.4 Dune: Giving Structure to Your Source Tree

Dune is a build system. It is used to orchestrate the compilation of source files into executables and libraries.

Assuming you have a development switch set up, you communicate to Dune about how your project is organized in terms of executables, libraries, and tests. It is then able to assemble the source files of your projects, with the dependencies installed in an opam switch, to create compiled assets for your project.

4.2.5 How Dune Integrates With the Ecosystem

Dune is designed to integrate with the tools mentioned above:

- By knowing how the OCaml compilers operate, it knows which build commands should be re-executed if some source files change.
- It outputs metadata like dependency information into `META` files that Findlib is able to make use of. This ensures that even if a project does not use Dune, it can use a library that has been produced by Dune. Conversely, it can read these files to determine dependency information for dependencies that have not been produced by Dune.
- It is able to generate opam files with filenames consistent with how opam looks for them. The generated files use build commands that make use of the *@install* and `@runtest` *aliases* so that the Dune abstractions map to the opam ones.

4.2.6 Dune is Opinionated

As described above, the OCaml ecosystem does not have a centralized toolchain. Units such as modules, libraries, and packages operate at different levels, and the relation between these can be confusing to users.

Dune tries to simplify the picture by reducing the difference between these objects:

- By default, a library will only expose a single top-level module named after the library (this is called a wrapped library).
- A library can only be installed in the package of the same name. This means that the names found in `dune-project` and `opam` files (package names) are consistent with the names found in `dune` files (library names). More precisely, libraries `foo`, `foo.bar` and `foo.baz` are part of the `foo` package.

4.3 How Dune integrates with opam

When instructed to do so (see *How to Generate Opam Files from dune-project*), Dune generates opam files with the following instructions:

```
build: [
  ["dune" "subst"] {dev}
  [
    "dune"
    "build"
    "-p"
    name
    "-j"
    jobs
    "@install"
    "@runtest" {with-test}
    "@doc" {with-doc}
  ]
]
```

Let's see what this means in detail.

4.3.1 Substitution

The first step is to call `dune subst`, but only if the `{dev}` opam variable is set. This variable is only set when the package is pinned. This means that *dune subst* does not run for released versions, but it does for development versions.

This is not a problem since released versions should have a `(version)` field set in `dune-project`, and *placeholder substitution* should have been performed. dune-release takes care of these steps.

4.3.2 Opam Variables

In the second command line, `name` is a variable that evaluates to the name of the package being built, and `jobs` is a variable that corresponds to the number of commands to run in parallel.

4.3.3 What `-p` Means

The `-p` flag, shorthand for `--release-of-packages`, is Dune's public interface to set up the options for an opam build. The exact semantics may change, but as of Dune 3.8 it is equivalent to the combination of:

- `--root .`: set the *root* to prevent Dune from *looking it up*.
- `--only-packages name`: ignore packages other than `name` defined in the project.
- `--profile release`: set the *build profile* to `release`. In particular, this ensures that warnings are not fatal.
- `--ignore-promoted-rules`: silently ignores all rules with `(mode promote)`.
- `--default-target @install`: make sure that `dune build` with no target argument builds `@install`, not `@@default` (this is not used in the opam integration since an explicit target is passed)
- `--no-config`: do not load the configuration file in the user's home directory.
- `--always-show-command-line`: ensures that the programs executed by Dune end up in the opam logs.
- `--promote-install-files`: ensures that `*.install` files are present in the source tree after the build.
- `--require-dune-project-file`: fail if `dune-project` is not present. In some previous Dune versions, `dune-project` could be generated when it is not present. This is not desirable with opam since the version the package has been prepared with is not known.

4.3.4 The Targets We're Building

The targets are specified as:

```
"@install"
"@runtest" {with-test}
"@doc" {with-doc}
```

The `{with- }` syntax is an opam filter. It means that the string before is present or not depending on the opam variable. These variables, in turn, are set depending on the opam configuration.

Concretely, in the next table, if the opam command on the left is executed, the Dune target on the right will be built:

opam command	Dune target
`opam install pkg`	`@install`
`opam install pkg --with-test`	`@install @runtest`
`opam install pkg --with-test --with-doc`	`@install @runtest @doc`

This filtering mechanism is also used to declare dependencies. If a package is using `lwt` and `alcotest`, but the latter only in its test suite, its `depends:` field is:

```
"lwt"
"alcotest" {with-test}
```

This is expanded to just `"lwt"` in `opam install pkg`, but to `"lwt" "alcotest"` in `opam install pkg --with-test`.

The meaning of these *aliases* is the following:

- *@install* depends on all the `*.install` files in the project. In turn, these depend on all the installable files (libraries and executables with a public name and files that are manually installed through `(install)` stanzas).
- *@runtest* is the alias to which all tests are attached, including `(test)` stanzas. `dune build @runtest` is equivalent to `dune runtest`.
- *@doc* executes `odoc` to create HTML docs under `_build`.

4.3.5 What Opam Expects From Dune

Given this `build:` lines and the fact that there is no `install:` line, what happens is the following:

- Opam executes `dune subst`, if the package is being pinned.
- Opam executes the build instruction, usually just `dune build -p pkg @install`
- This Dune command builds all the installable files and creates a `pkg.install` file.
- This file contains the paths to built files (somewhere in the `_build` directory) and the opam sections they should be installed in.
- Opam interprets this file and copies the built files to their destination. The install file is also used as a manifest of which files belong to which package, which is used when uninstalling the package.

4.4 How Dune Uses Dune to Build Dune

Dune's build system is itself Dune. This works thanks to a bootstrap process. This document explains how this works.

4.4.1 boot/bootstrap.ml

`boot/bootstrap.ml` is an OCaml script (it is interpreted, not compiled) that is a mini-build system tailored to Dune itself. It computes dependencies between the various modules by calling `ocamldep`, and it will generate build and link commands. It knows how to execute these commands in parallel. It does not read any `dune` file. However, the project structure and its system dependencies are encoded in `boot/libs.ml`.

This step produces `_boot/dune.exe`.

4.4.2 Completing the Opam Installation

`_boot/dune.exe` is the bootstrap Dune. Since it has been built from the Dune sources, it will act like Dune: it can read `dune` files, etc.

This is actually the `dune` executable that will get installed. This is the same executable as the one obtained by running `opam install dune`. At this stage of the process, Opam does not know about this: it expects a `dune.install` file that explains what files to install.

The next command run by the Opam instruction is the following:

```
$ ./_boot/dune.exe build dune.install --release --profile dune-bootstrap
```

By using the `dune-bootstrap` *build profile*, it does not run a full build, but only copy `_boot/dune.exe` to its install location (as the *dune* binary), and generate `dune.install`.

4.4.3 `make dev`: Everything Else for Local Development

The above describes how Dune itself is built through Opam, but that's not all there is it to it: the Dune repository contains other libraries that need to be built. In fact, executing the `boot/bootstrap.ml` script did not generate files useful for editor integration.

So the main `Makefile` has a `make dev` target that will run `_boot/dune.exe build @install`: this will rebuild the project using Dune itself.

As a special rule, this build will regenerate `boot/libs.ml` using the locations of the internal libraries used to build Dune.

4.5 The Dune Mental Model

It is not strictly necessary to understand Dune's underlying model to use it; but knowing how it works under the hood will help writing build rules, and also help understand some errors and what's possible with Dune.

Note: This document is a simplification of the reality: the actual rules might be different, it does not touch rule loading and glosses over how caching works, but should be a useful tool to build an understanding of Dune.

4.5.1 How Dune Works

The building block of Dune is the *rule*:

> A *rule* reads *dependencies* and writes *targets* using an *action* (and it can be attached to *aliases*).

When `dune build` is executed, it will first read the project's `dune` files to determine the rules that apply to the project. Once it has done this, it will determine what actions it needs to execute to build the required targets.

4.5.2 An Example

Let's take the following example.

- there's a CLI tool written in OCaml.
- it has some build-time configuration stored in `config.json`.
- it has an integration test, in which the tool is executed with `testdata.txt` as input.

Configuration Generation

To express the generation of the configuration module we could write:

```
(rule
 (deps convert/json2ml.exe config.json)
 (target config.ml)
 (action
  (run convert/json2ml.exe config.json -o config.ml)))
```

This rule will:

- read its dependencies: `convert/json2ml.exe` and `config.json`
- and write its target: `config.ml`
- using an action: `(run convert/json2ml.exe config.json -o config.ml)`

This rule is very explicit: we write a stanza for a single Dune rule.

Building the Executable

In contrast, to describe the compilation of the executable, we would write:

```
(executable
 (name tool)
 (modules main config))
```

Here, we use Dune's abstractions. Dune knows about the OCaml compilation model: the modules need to be compiled and linked together. So it will generate the following rules under the hood:

- one rule to compile the `Main` module:
  - it will read its dependency: `main.ml`
  - and write its output: `main.cmx`
  - using an action: `(run ocamlopt -c main.ml)`
- one rule to compile the `Config` module:
  - it will read its dependency: `config.ml`
  - and write its output: `config.cmx`
  - using an action: `(run ocamlopt -c config.ml)`
- one rule to link the `tool.exe` executable:
  - it will read its dependencies: `main.cmx` and `config.cmx`
  - and write its output: `tool.exe`

  - using an action: `(run ocamlopt -o tool.exe main.cmx config.cmx)`

Note that in this example, some files are targets of a rule and dependencies of another (`.cmx` files). We are unlikely to ever interact with them directly, so it can also be useful to think of the `(executable)` stanza as a group of rules with `main.ml` and `config.ml` as inputs and `tool.exe` as output.

Running the Tests

Some rules do not produce any output file, but we're still interested in running their actions. A test is a good example: we want the build process to exit with an error code if the action fails. In that case, the rule does not have targets, but we "attach" it to an *alias*, `runtest` in this case. This gives us a way of requesting this rule to be executed. As we are about to see, rules are executed lazily by asking for their targets to be built, so we would not be able to execute such rules.

```
(rule
 (deps tool.exe testdata.txt)
 (alias runtest)
 (action
  (run tool.exe testdata.txt)))
```

This rule:

- reads its dependencies: `tool.exe` and `testdata.txt`
- writes no targets
- using an action: `(run tool.exe testdata.txt)`
- (and it is attached to `runtest`)

4.5.3 What to Build

Dune can build *files* and *aliases*. These can be found on the command line:

- `dune build tool.exe` will build the `tool.exe` file.
- `dune build @example` will build the `example` alias.
- `dune build tool.exe @example` will build both the file `tool.exe` and the `example` alias.
- `dune runtest` is a shortcut for `dune build @runtest`: it will build the `runtest` alias.
- `dune build` is a shortcut for `dune build @@default`: it will build the default alias in the current directory (by default the `all` alias).

In other words, each `dune build` or `dune runtest` command always corresponds to a list of files and aliases to build.

See also:

Reference information on aliases

4.5.4 How Dune Interprets Rules

We have now seen that Dune sets up rules for a project, and that every build command has a list of files and aliases that we are asking to build.

Now let's see how this request is processed:

- to build a file, Dune will first check if it is in the source tree. In that case, there is nothing to do. Otherwise, it will check if it is the target of a rule. In that case, it will execute this rule. (Dune will raise an error in other cases: if the file is both in the source tree and the target of a rule, or if it is neither)
- to build an alias, Dune will execute all the rules that are attached to this alias.
- to execute a rule, Dune will first build all the dependencies (files or aliases) of this rule. Then it will execute the action attached to the rule. When Dune is about to execute an action, it checks (in various caches) if it executed it before on the same set of dependencies, and, if yes, it can skip executing it and reuse the previous result.

In the case of our example, if we call `dune runtest`, Dune will consider all rules attached to the `runtest` alias. In this case it is just the integration test rule. It needs to build its dependencies, `tool.exe` and `testdata.txt`. The latter is present in the source tree. However, `tool.exe` is the target of the linking rule defined by the `(executable)` stanza. This rule requires `main.cmx` and `config.cmx`. `main.cmx` is the target of the compilation rule for the `Main` module, which depends on `main.ml`. This file is in the source tree, so let's copy it under `_build`. This rule has all its dependencies available, so we can run its action, which writes `main.cmx`. Getting back to the dependencies of `tool.exe`, `config.cmx` is the target of the linking rule of the `Config` module. This rule has `config.ml` has a dependency. This file is itself the target of the configuration module rule, which lists `config.json` and `convert/json2ml.exe`. The first is available in the source tree and to simplify, let's assume that the second one has been built. This action has all its dependencies available, so we can execute its action to produce its target, `config.ml`. Now the module compilation rule for `Config` can be executed, producing `config.cmx`; and in turn the linking rule can be executed, producing `tool.exe`. Finally, `tool.exe` can be executed with `testdata.txt` as its argument.

In a nutshell: we recursively copied all the dependencies of the test rule, and executed the rules in the correct order.

This is a "cold build", where there were no previous build artifacts. Note that if we change only part of the project (say the `main.ml` file), only a small number of rules will be evaluated, the ones that depend on `main.ml`.

4.5.5 Conclusion

Dune's underlying model is based on rules. Stanzas are high-level constructs that can generate multiple rules, that are not always visible.

To build a target, Dune looks for the rule that produces that target and makes its way back to source files.

Rules define a directed acyclic graph which models dependency relations between files. Most of the rules in that graph may be executed for a cold build, but just the minimum will be executed for an incremental build.

4.6 A Tour of the Dune Codebase

Note: This document is based on Dune 3.15.0, whose source can be browsed here. The links in this tour point to this version, but will not reflect how this works in other versions of Dune.

Let's start with a very high level tour of how `dune build` operates.

As explained in *The Dune Mental Model*, `dune build` will interpret the targets listed on the command line, interpret the `dune` files in the workspace as rules, and execute the rules relevant to the requested targets.

These steps correspond to areas of the Dune codebase:

- the command-line interface is defined in `bin/`;
- the `dune` files are interpreted using a library defined in `src/dune_rules/`;
- they are registered into an engine in `src/dune_engine/`.

Next, we will go deeper into these areas.

4.6.1 Command-Line Interface

The command-line interface is defined using cmdliner. One thing to note is that we use binding operators to compose terms:

bin/print_rules.ml

```
174 let+ builder = Common.Builder.term
175 and+ out =
176   Arg.(
177     value
178     & opt (some string) None
179     & info [ "o" ] ~docv:"FILE" ~doc:"Output to a file instead of stdout.")
180 and+ recursive =
181   Arg.(
182     value
183     & flag
184     & info
185         [ "r"; "recursive" ]
186         ~doc:
187           "Print all rules needed to build the transitive dependencies of the given␣
    ↪\
188            targets.")
189 and+ syntax = Syntax.term
190 and+ targets = Arg.(value & pos_all dep [] & Arg.info [] ~docv:"TARGET") in
```

4.6.2 Parsing of Dune Files

Parsing `dune` files is done in two steps:

- They are parsed as S-expressions using `src/dune_sexp/parser.mli`;
- Then they are decoded using `src/dune_sexp/decoder.mli`. The result of this decoding step is added to an extensible variant using a mechanism in `src/dune_lang/stanza.mli`.

Instead of writing a parser or using pattern matching, we define decoders, which are abstract values of type `'a Decoder.t` (returning a value of type `'a`). These decoders are assembled using combinators. For example, we can use simple decoders to write a decoder for a record type. This decoder abstraction is monadic, but the applicative subset is sufficient for most decoders.

As an example, here is how `(copy_files)` is parsed:

src/dune_rules/stanzas/copy_files.ml

```
31 let long_form =
32   let check = Dune_lang.Syntax.since Stanza.syntax (2, 7) in
```

(continues on next page)

(continued from previous page)

```
  let+ alias = field_o "alias" (check >>> Dune_lang.Alias.decode)
  and+ mode = field "mode" ~default:Rule.Mode.Standard (check >>> Rule_mode_decoder.
↪decode)
  and+ enabled_if = Enabled_if.decode ~allowed_vars:Any ~since:(Some (2, 8)) ()
  and+ files = field "files" (check >>> String_with_vars.decode)
  and+ only_sources =
    field_o
      "only_sources"
      (Dune_lang.Syntax.since Stanza.syntax (3, 14) >>> decode_only_sources)
  and+ syntax_version = Dune_lang.Syntax.get_exn Stanza.syntax in
  let only_sources = Option.value only_sources ~default:Blang.false_ in
  { add_line_directive = false
  ; alias
  ; mode
  ; enabled_if
  ; files
  ; only_sources
  ; syntax_version
  }
```

The fields are queried individually, and a record is built using all the intermediate results. This will automatically take care of generating "unknown field X," "duplicate field X," and similar error messages.

Another interesting thing to note is that the fields are not decoded directly, but use the following pattern:

```
Syntax.since Stanza.syntax (x, y) >>> decoder
```

Let's unpack this: `(>>>)` will run a `unit Decoder.t` on the input before passing the input to an actual decoder. The first decoder can be used to implement a check and trigger an error in some cases.

Here, it is used for versioning. For example the `(copy_files)` stanza started supporting `(enabled_if)` in version 2.8. Decoding this field is protected by this `since` call: it means that if the language version in *dune-project* file is greater than 2.8. In particular, this ensures that the project can not be built with Dune versions older than `2.8.0`.

Once decoding succeeds, various stanzas are turned into various types defined in `src/dune_rules/stanzas/`.

4.6.3 Rule Generation

Using these parsed stanzas, the next step is to generate rules. This work starts in `src/dune_rules/gen_rules.ml`, which dispatches to various modules in `src/dune_rules/`.

Rules are registered on the build engine using the following function from the `Super_context` module:

```
val add_rule
  :  t
  -> ?mode:Rule.Mode.t
  -> ?loc:Loc.t
  -> dir:Path.Build.t
  -> Action.Full.t Action_builder.With_targets.t
  -> unit Memo.t
```

A value of `Super_context.t` represents an OCaml toolchain (`Context.t`) as well as various capabilities to expand variables and refer to *(env) stanzas*. The last, unlabelled argument corresponds to the fully annotated action. We'll go through its type below.

The modules in `src/dune_rules` often expose a function `gen_rules` taking a parsed stanza, a `Super_context.t` value, a directory name (and other arguments), and returning `unit Memo.t`.

Note: The `Memo` module is central to how Dune operates. It is a monadic memoization framework that allows two things:

- Sharing and caching expensive internal computations, such as computing the list of libraries Dune knows about, or computing the list of flags that should be used to compile a given module.
- Incremental recomputation of this cached data. `Memo` tracks dependencies between memoized values and will only recompute the necessary ones when an input changes. This is a mini in-memory build system that works like a spreadsheet. It is essential to the watch mode.

An example of rule is the *mdx* stanza, implemented in `src/dune_rules/mdx.ml`. There are several steps in setting up rules for a `(mdx)` stanza:

- How to run `ocaml-mdx deps` on the input file to produce a `.mdx.deps`
- Run `ocaml-mdx dune-gen` to produce a `mdx_gen.ml-gen` OCaml source file
- Compile this executable
- Run this executable to produce a `.corrected` file
- Register a *diff* action between the `.corrected` file and the original file

Let's walk through these rules.

The first one is about producing a `.mdx.deps` file. It is a simple call to `Super_context.add_rule`.

```
312 let* () = Super_context.add_rule sctx ~loc ~dir (Deps.rule ~dir ~mdx_prog files)
```

`Deps.rule` is defined in a helper function:

```
77 let rule ~dir ~mdx_prog (files : Files.t) =
78   Command.run_dyn_prog
79     ~dir:(Path.build dir)
80     mdx_prog
81     ~stdout_to:files.deps
82     [ Command.Args.A "deps"; Lazy.force color_always; Dep (Path.build files.Files.src) ]
```

This is a rule made by just running a command, here `mdx_prog` (a resolved path to `ocaml-mdx`, meaning it can point to a binary in `PATH` or a built version in the current workspace). Its arguments are a domain-specific language defined in `src/dune_rules/command.mli` where `A` refers to a plain string, and `Dep` refers to a string that should be interpreted as a dependency. Between that, and the `~stdout_to` parameter, it is enough for Dune to know about the rule's dependencies (what it will read) and its target (what it will produce).

The second rule, which generates `mdx_gen.ml-gen`, is similar. It is also done by calling `Command.run_dyn_prog`.

The third rule, to build the executable, calls `Exe.build_and_link` that is a helper function.

Let's observe how the fourth rule (that calls the generated executable) is set up.

```
let mdx_action ~loc:_ =
  let open Action_builder.With_targets.O in
  let mdx_input_dependencies = (* ... *) in
  let executable, command_line = (* ... *) in
  let deps, sandbox = (* ... *) in
  let+ action =
```

(continues on next page)

(continued from previous page)

```
    Action_builder.with_no_targets deps
    >>> Action_builder.with_no_targets
          (Action_builder.env_var "MDX_RUN_NON_DETERMINISTIC")
    >>> Action_builder.with_no_targets
          (Action_builder.map mdx_input_dependencies ~f:(fun d -> (), d)
           |> Action_builder.dyn_deps)
    >>> Command.run_dyn_prog
          ~dir:(Path.build dir)
          ~stdout_to:files.corrected
          executable
          command_line
  and+ locks =
    Expander.expand_locks expander stanza.locks |> Action_builder.with_no_targets
  in
  Action.Full.add_locks locks action |> Action.Full.add_sandbox sandbox
in
Super_context.add_rule sctx ~loc ~dir (mdx_action ~loc)
```

Here, the `mdx_action` that is set up is not just a single `Command.run_dyn_prog` call. It is assembled using combinators from `Action_builder.With_targets`. This is another monad used in Dune. It corresponds to what can happen at build time, like running commands or creating files, or more complex actions such as reading a file that needs to be built by another rule. It is also used to track dependencies and targets. The "thing" that we register to the Dune engine using `Super_context.add_rule` has type `Action.Full.t Action_builder.With_targets.t`.

Note: This is different from `Memo`, which corresponds to what happens within Dune itself. But it is also possible to use `Memo` from an `Action_builder` context. In that sense, `Action_builder` is more powerful: at execution time, `Action_builder` will manage what happens in the `_build` directory, while `Memo` is only concerned with what happens in memory.

Finally, to register the correction, the technique is to attach the *diff* action to the *@runtest* alias (a collection of rules) using this call:

```
(* Attach the diff action to the @runtest for the src and corrected files *)
Files.diff_action files
|> Super_context.add_alias_action sctx (Alias.make Alias0.runtest ~dir) ~loc ~dir
```

Where `Files.diff_action` is defined as:

```
let diff_action { src; corrected; deps = _ } =
  let src = Path.build src in
  let open Action_builder.O in
  let+ () = Action_builder.path src
  and+ () = Action_builder.path (Path.build corrected) in
  Action.Full.make (Action.diff ~optional:false src corrected)
;;
```

As explained above, `Action_builder` keeps tracks of dependencies, so using `let+ () = Action_builder.path src` is a way to declare `src` as a dependency of the current action.

4.6.4 The Engine

The engine is the core, reusable part of Dune. It contains all the composable primitives that make it a build system.

The fact that it is split from the *rules part* makes it possible to create a different build system using this library. For example, Jane Street internally uses a build system with this engine as a backend, but a different frontend and CLI.

In the context of Dune, the engine keeps track of the various directories and the rules in them and is able to build files using them. In addition, it takes care of the various caches that Dune uses, such as the one present in the `_build` directory, the *shared cache*, etc.

4.6.5 Libraries

Dune, as a package, is primarily an executable, but its source tree embeds a few public libraries. These are developed in `otherlibs/`.

Some of these have a special link to Dune, such as `dune-build-info` or `dune-site`. Others are just helper libraries that we develop as part of Dune, but they have a strong relation to the Dune internals, like `dyn` or `xdg`.

See also:

Dune Libraries

4.6.6 Vendored Libraries

As an opam package, Dune has no dependencies. But it uses some existing libraries by copying, or "vendoring", their source code into the `vendor/` directory.

In some cases, the external dependency is extracted from the upstream repository. In other cases, we carry patches and refer to a fork in the ocaml-dune GitHub organization.

The source code in the `vendor/` directory is not meant to be edited directly. Instead, it is edited in the external repository, and the copy in the Dune source tree is updated by running an update script, such as update-spawn.sh.

4.6.7 Tests

The `test/` directory contains all the tests for Dune itself. Additionally, the tests for our *Libraries* are stored in `otherlibs/` next to the library itself.

We have 3 kind of tests:

- Unit tests, in `test/unit-tests` (we have very few of these, usually prefering other kinds)
- Expect tests, in `test/expect-tests` (using `ppx_expect`)
- *Cram tests*, in `test/blackbox-tests/`. This is our preferred way of testing.

The actual Cram tests are in `test/expect-tests/test-cases`. There is a mix of file tests and directory tests. For regression tests, the pattern `githubNUMBER.t` is used.

The `dune` file at `test/expect-tests/test-cases/dune` sets up some metadata for the tests. For example, if a test has an external dependency like `strace`, a dependency on `%{bin:strace}` will prevent the test from even trying to start. Some tests are also disabled on some configurations using `(enabled_if)`.

Finally, some programs available in the Cram tests are defined in `test/expect-tests/blackbox-tests/utils`. For example, we have a dune_cmd program that contains reimplementations of common utilities like `stat`, which do not have the same output on the different systems we use to test Dune.

See also:

Working on the Dune Codebase

CHAPTER

FIVE

ADVANCED TOPICS

These documents describe some advanced or very specific features of Dune.

5.1 Dynamic Loading of Packages with Findlib

The preferred way for new development is to use *Plugins and Dynamic Loading of Packages*.

Dune supports the `findlib.dynload` package from Findlib that enables dynamically-loading packages and their dependencies (using the OCaml Dynlink module). Adding the ability for an application to have plugins just requires adding `findlib.dynload` to the set of library dependencies:

```
(library
  (name mytool)
  (public_name mytool)
  (modules ...)
)

(executable
  (name main)
  (public_name mytool)
  (libraries mytool findlib.dynload)
  (modules ...)
)
```

Use `Fl_dynload.load_packages l` in your application to load the list `l` of packages. The packages are loaded only once, so trying to load a package statically linked does nothing.

A plugin creator just needs to link to your library:

```
(library
  (name mytool_plugin_a)
  (public_name mytool-plugin-a)
  (libraries mytool)
)
```

For clarity, choose a naming convention. For example, all the plugins of `mytool` should start with `mytool-plugin-`. You can automatically load all the plugins installed for your tool by listing the existing packages:

```
let () = Findlib.init ()
let () =
  let pkgs = Fl_package_base.list_packages () in
```

(continues on next page)

(continued from previous page)

```
let pkgs =
  List.filter
    (fun pkg -> 14 <= String.length pkg && String.sub pkg 0 14 = "mytool-plugin-")
    pkgs
in
Fl_dynload.load_packages pkgs
```

5.2 Profiling Dune

If `--trace-file FILE` is passed, Dune will write detailed data about internal operations, such as the timing of commands that Dune runs.

The format is compatible with Catapult trace-viewer. In particular, these files can be loaded into Chromium's `chrome://tracing`. Note that the exact format is subject to change between versions.

5.3 Package Version

Dune determines a package's version by looking at the `version` field in the *package*. If the version field isn't set, it looks at the toplevel `version` field in the `dune-project` field. If neither are set, Dune assumes that we are in development mode and reads the version from the VCS, if any. The way it obtains the version from the VCS is described in *the build-info section*.

When installing the files of a package on the system, Dune automatically inserts the package version into various metadata files such as `META` and `dune-package` files.

5.4 OCaml Syntax

If a `dune` file starts with `(* -*- tuareg -*- *)`, then it is interpreted as an OCaml script that generates the `dune` file as described in the rest of this section. The code in the script will have access to a Jbuild_plugin module containing details about the build context it's executed in.

The OCaml syntax gives you an escape hatch for when the S-expression syntax is not enough. It isn't clear whether the OCaml syntax will be supported in the long term, as it doesn't work well with incremental builds. It is possible that it will be replaced by just an `include` stanza where one can include a generated file.

Consequently **you must not** build complex systems based on it.

5.5 Variables for Artifacts

For specific situations where one needs to refer to individual compilation artifacts, special variables (see *Variables*) are provided, so the user doesn't need to be aware of the particular naming conventions or directory layout implemented by Dune.

These variables can appear wherever a *Dependency Specification* is expected and also inside *Actions*. When used inside *Actions*, they implicitly declare a dependency on the corresponding artifact.

The variables have the form `%{<ext>:<path>}`, where `<path>` is interpreted relative to the current directory:

- `cmo:<path>`, `cmx:<path>`, and `cmi:<path>` expand to the corresponding artifact's path for the module specified by `<path>`. The basename of `<path>` should be the name of a module as specified in a `(modules)` field.
- `cma:<path>` and `cmxa:<path>` expands to the corresponding artifact's path for the library specified by `<path>`. The basename of `<path>` should be the name of the library as specified in the `(name)` field of a `library` stanza (*not* its public name).

In each case, the expansion of the variable is a path pointing inside the build context (i.e., `_build/<context>`).

5.6 Building an Ad Hoc `.cmxs`

In the model exposed by Dune, a `.cmxs` target is created for each library. However, the `.cmxs` format itself is more flexible and is capable to containing arbitrary `.cmxa` and `.cmx` files.

For the specific cases where this extra flexibility is needed, one can use *Variables for Artifacts* to write explicit rules to build `.cmxs` files not associated to any library.

Below is an example where we build `my.cmxs` containing `foo.cmxa` and `d.cmx`. Note how we use a *library* stanza to set up the compilation of `d.cmx`.

```
(library
 (name foo)
 (modules a b c))

(library
 (name dummy)
 (modules d))

(rule
 (targets my.cmxs)
 (action (run %{ocamlopt} -shared -o %{targets} %{cmxa:foo} %{cmx:d})))
```

CHAPTER

SIX

MISCELLANEOUS

These documents contain tidbits of info that do not fit anywhere else, and information about the project itself.

6.1 FAQ

6.1.1 Why Do Many Dune Projects Contain a `Makefile`?

Many Dune projects contain a root `Makefile`. It's often only there for convenience for the following reasons:

1. There are many different build systems out there, all with a different CLI. If you have been hacking for a long time, the one true invocation you know is `make && make install`, possibly preceded by `./configure`.
2. You often have a few common operations that aren't part of the build, so `make <blah>` is a good way to provide them.
3. `make` is shorter to type than `dune build @install`

6.1.2 How to Add a Configure Step to a Dune Project

The with-configure-step example shows one way to add a configure step that preserves composability; i.e., it doesn't require manually running the `./configure` script when working on multiple projects simultaneously.

6.1.3 Can I Use `topkg` with Dune?

While it's possible to use the topkg-jbuilder, it's not recommended. dune-release subsumes `topkg-jbuilder` and is specifically tailored to Dune projects.

6.1.4 How Do I Publish My Packages with Dune?

Dune is just a build system and considers publishing outside of its scope. However, the dune-release project is specifically designed for releasing Dune projects to opam. We recommend using this tool for publishing Dune packages.

6.1.5 Where Can I Find Some Examples of Projects Using Dune?

The dune-universe repository contains a snapshot of the latest versions of all opam packages that depend on Dune. Therefore, it's a useful reference to find different approaches for constructing build rules.

6.1.6 What is Jenga?

jenga is a build system developed by Jane Street, mainly for internal use. It was never usable outside of Jane Street, so it's not recommended for general use. It has no relationship to Dune apart from Dune being the successor to Jenga externally. Eventually, Dune is expected to replace Jenga internally at Jane Street as well.

6.1.7 How to Make Warnings Non-Fatal

jbuilder formerly displayed warnings, but most of them wouldn't stop the build. However, Dune makes all warnings fatal by default. This can be a challenge when porting a codebase to Dune. There are two ways to make warnings non-fatal:

- The `jbuilder` compatibility executable works even with `dune` files. You can use it while some warnings remain and then switch over to the `dune` executable. This is the recommended way to handle the situation.
- You can pass `--profile release` to `dune`. It will set up different compilation options that usually make sense for release builds, including making warnings non-fatal. This is done by default when installing packages from opam.
- You can change the flags used by the `dev` profile by adding the following stanza to a `dune` file:

```
(env
  (dev
    (flags (:standard -warn-error -A))))
```

6.1.8 How to Turn Specific Errors into Warnings

Dune is strict about warnings by default in that all warnings are treated as fatal errors. To change certain errors into warnings for a project, you can add the following to `dune-workspace`:

```
(env (dev (flags :standard -warn-error -27-32)))
```

In this example, the warnings 27 (unused-var-strict) and 32 (unused-value-declaration) are treated as warnings rather than errors.

6.1.9 How to Display the Output of Commands as They Run

When Dune runs external commands, it redirects and saves their output, then displays it when complete. This ensures that there's no interleaving when writing to the console.

But this might not be what the you want. For example, when you debug a hanging build.

In that case, one can pass `-j1 --no-buffer` so the commands are directly printed on the console (and the parallelism is disabled so the output stays readable).

6.1.10 How Can I Generate an `mli` File From an `ml` File

When a module starts as just an implementation (`.ml` file), it can be tedious to define the corresponding interface (`.mli` file).

It is possible to use the `ocaml-print-intf` program (available on opam through `$ opam install ocaml-print-intf`) to generate the right `mli` file:

```
$ dune exec -- ocaml-print-intf ocaml_print_intf.ml
val root_from_verbose_output : string list -> string
val target_from_verbose_output : string list -> string
val build_cmi : string -> string
val print_intf : string -> unit
val version : unit -> string
val usage : unit -> unit
```

The `ocaml-print-intf` program has special support for Dune, so it will automatically understand external dependencies.

6.1.11 How Can I Build a Single Library?

You might want to do this when you don't have all the dependencies installed to compile an entire project, or parts of the project don't build for whatever reason. Maybe you want to check if your changes compile or produce build artifacts needed by `ocaml-lsp-server`.

Suppose you have a library defined in `src/foo/dune`:

```
(library
 (public_name my_library)
 ...)
```

You can build this library on its own by running the following from the project root directory:

```
$ dune build %{cmxa:src/foo/my_library}
```

Note that the path (`src/foo` in the example above) is relative to the current directory - not the project root. If the library defines a `name` distinct from its `public_name` then that can be used interchangeably with the `public_name` in this command.

6.1.12 Files and Directories Whose Names Begin with "." (Period) are Ignored by `source_tree`

Dune's default behaviour is to ignore files and directories starting with "." when copying directories with `source_tree`. This is to avoid accidentally copying the `.git` directory into the `_build` directory during a build.

This is a common source of confusion when interoperating with other libraries that use hidden directories for configuration, such as Rust. For example consider this rule which builds a Rust library contained in a subdirectory foo-rs:

```
(rule
 (target foo.a)
 (deps
  (source_tree foo-rs))
 (action
```

(continues on next page)

(continued from previous page)

```
 (progn
  (chdir
   foo-rs
   (run cargo build --release))
  (run mv foo-rs/target/release/%{target} %{target})))
```

The build config for the Rust project will be in a directory `foo-rs/.cargo/config.toml`, and by default the `.cargo` directory won't get copied into the `_build` directory and so the Rust project will build with an incorrect configuration.

To fix this, create a `dune` file at the top level of the Rust project (i.e., `foo-rs/dune`):

```
(dirs :standard .cargo)
```

If you're following the standard advice for embedding Rust projects into Dune projects then you likely already have a `dune` project inside your Rust project that looks like:

```
(dirs :standard \ target)
(data_only_dirs vendor)
```

In this case you can update it to look like this:

```
(dirs :standard .cargo \ target)
(data_only_dirs vendor)
```

6.1.13 How Can I Write Inline Tests in a Package Without my Users Needing to Install `ppx_inline_test`?

If you came to OCaml from Rust and noticed that Dune has a feature for running inline tests you might be wondering how to do the OCaml equivalent of:

```
// define a private function
fn foo() { ... }

// test the function right next to its definition
#[test]
fn test_of_foo() { ... }
```

That is, writing tests for private functions right next to the definition of those functions. The *Inline Tests* documentation describes how to do this using the `ppx_inline_test` package; however, if you do this in your package, then your package must *unconditionally* depend on the `ppx_inline_test` package. Opam has a notion of test-only dependencies (its `with-test` flag), but you cannot use this with `ppx_inline_test`. The consequence of this is that anyone depending on your package is also transitively depending on `ppx_inline_test` as well as all of its dependencies.

The reason for this is OCaml code with preprocessor directives (such as those used for inline tests with `ppx_inline_test`) is technically not valid OCaml code until it has been preprocessed. Unlike the cargo build system used for Rust, Dune does not have a preprocessor built into it. Instead, it relies on external tools (such as `ppx_inline_test`) to parse the code and replace any preprocessor directives with valid OCaml. Dune doesn't know how to parse OCaml code at all so it can't even remove inline tests from the code in cases where `ppx_inline_test` is unavailable.

The blessed workaround for folks who want to use `ppx_inline_test` in their packages but don't want to add it as a dependency is to create a new (unreleased) package which contains all the tests. In the original package, expose all the private APIs you intend to test via public modules named something foreboding such as `For_test` so your users

know not to rely on their contents and then have the test package define tests that call your "private" APIs through the `For_test` modules.

6.2 Goal of Dune

The Dune project strives to provide the best possible build tool for the entire OCaml community, including individual developers contributing to open source projects in their free time, larger companies (such as Jane Street), and communities, like MirageOS and Irmin. Additionally, we aim to provide the same features for other neighbouring communities, such as Coq and possibly Reason/Bucklescript, in the future.

We haven't reached this goal yet, as Dune still requires development in some areas to be such a tool, but we're steadily working towards that goal. On a practical level, a few boxes must be checked, and a considerable number of details needs to be sorted out. At a high-level, we think a tool that works for everyone in the OCaml community should at least:

1. have excellent backward compatibility properties
2. have a robust and scalable core
3. remain a no-brainer dependency
4. remain accessible
5. have very good support for the OCaml language
6. be extensible

At this point, we've done a good job at 1, 3, 4, and 5. We're currently working towards 2 and are doing the preparatory work for 6. Once all these boxes have been checked, we'll consider the Dune project complete.

Below, we develop each point and give some insights into our current and future focuses.

6.2.1 Have Excellent Backward-Compatibility Properties

In an open source community, two types of groups exist: those with enough resources to continuously bring their projects up-to-date and those who work on them in their free time. The latter obviously can't provide the same level of continuous support and updates as the former.

From the Dune point of view, we consider every released project with `dune` files a precious piece that will potentially never change, so we discourage changing Dune in a way where it could no longer understand a released project.

Of course, we can't give a 100% guarantee that Dune will always behave exactly the same. That would be unrealistic and would prevent the project from moving forward. In order to provide good backward-compatibility properties while still keeping the project fresh and dynamic, we need to properly delimit, document, and version the set of behaviours on which users rely. For this to be manageable, the surface Dune API must remain small.

A distinguishing feature of Dune allows the user to declare which version of the `dune` tool they wrote the project against, and `dune` will morph itself to behave the same as this version of the `dune` binary, even if it's a newer version. As a result, a recent `dune` binary version can understand a wide range of Dune projects written against many different versions of Dune, and while we strictly follow semantic versioning, new major versions of Dune effectively introduce very few breaking changes. Most projects don't need upper bounds on Dune.

This guarantee is of course limited to documented behaviours.

6.2.2 Have a Robust and Scalable Core

Tech companies tend be fond of big mono repositories, so for compatibility, Dune must consume large repositories without blinking. It not only needs to build fast, but more importantly, it must not impede fast feedback during development, no matter the size of the repository.

Note that we'll only test Dune on repositories as large as people participating in Dune's development require. Currently, the largest user is Jane Street. If someone wanted to use Dune on much larger repositories than the ones used at Jane Street, and this required a significant amount of effort on Dune, this wouldn't be considered unless we get some help to do so and we can keep the other promises.

In particular, while making Dune scalable, we must also ensure Dune doesn't turn into a monster, because no one wants to force their users to install a monster to build their project. This brings us to the next point of Dune being a no-brainer dependency:

6.2.3 Remain a No-Brainer Dependency

Dune is a hard dependency of any Dune project. Anyone using Dune to develop their project will have to ask their user to install Dune. For this reason, it is very important to keep Dune as lean as possible.

We need to be careful when we start relying on an external piece of software or when we introduce new concepts. We must not introduce duplication or useless stuff. The overall projects has to remain lean.

It's also important to keep Dune as easy to install as possible. Currently, the only requirement to build Dune is a working OCaml compiler. Nothing else is required, not even a shell, and we should keep it this way.

6.2.4 Remain Accessible

Since Dune aims to be the best possible tool for the whole OCaml community, it's important to keep Dune accessible. Getting started and learning Dune should be straightforward.

For that purpose, when designing the language (the command line interface or the documentation), we must take on the new-user perspective, one who just discovered Dune and its features, because Dune should be suitable for everyone! It also needs to provide advanced and more complex features for expert users. However, the documentation should always flow from the simpler concepts and common tasks to the more complex ones, even if the simpler features can be explained as instances of the more general ones.

6.2.5 Have Excellent Support for the OCaml Language

There are many, many build systems out there. Dune stands out because it primarily targets the OCaml community, so Dune must come with excellent support for the OCaml language and OCaml projects in general.

If it didn't, Dune would just be yet another generic build system.

Perhaps in the future some of the general build system will take over, and Dune might just become a plugin in this system. It could even disappear into the language, if the compiler gains significant high-level features. But for now, Dune is a standalone build system that primarily serves the OCaml community's needs, and to the extent that is reasonably possible, the needs of other functional language communities.

6.2.6 Be Extensible

No matter the quality of the OCaml language's support, it will never be enough to cover every single project need. For this reason, Dune must provide some form of openness for projects with need that don't completely fit in the Dune model.

In the long run, extensibility tends to obstruct innovation, and we should always strive to ensure that we cover all the general needs of the main Dune language; however, we'll always need an escape hatch for Dune to remain a practical choice.

It's pretty clear that extensibility must be done via OCaml code, and currently it's a bit difficult to use OCaml as a proper extension language, though some work is being done to help on that front.

6.3 Working on the Dune Codebase

This section gives guidelines for working on Dune itself. Many of these are general guidelines specific to Dune. However, given that Dune is a large project developed by many different people, it's important to follow these guidelines in order to keep the project in a good state and pleasant to work on for everybody.

Table of Contents

- *Dependencies*
- *Bootstrapping*
- *Writing Tests*
- *Setting Up Your Development Environment Using Nix*
- *Releasing Dune*
- *Adding Stanzas*
- *Dune Rules*
- *Documentation*
- *Vendoring*
- *General Guidelines*
- *Benchmarking*
- *Formatting*

6.3.1 Dependencies

To create a directory-local opam switch with the dependencies necessary to build the tests, run:

```
$ make dev-switch
```

6.3.2 Bootstrapping

Dune uses Dune as its build system, which requires some specific commands to work. Running `make dev` bootstraps (if necessary) and runs `./dune.exe build @install`.

If you want to just run the bootstrapping step itself, build the `bootstrap` phony target with

```
$ make bootstrap
```

You can always rerun this to bootstrap again.

Once you've bootstrapped Dune, you should be using it to develop Dune itself. Here are the most common commands you'll be running:

```
# to make sure everything compiles:
$ ./dune.exe build @check
# run all the tests
$ ./dune.exe runtest
# run a particular cram foo.t:
$ ./dune.exe build @foo
```

Note that tests are currently written for version 4.14.1 of the OCaml compiler. Some tests depend on the specific wording of compilation errors which can change between compiler versions, so to reliably run the tests make sure that `ocaml.4.14.1` is installed. The `TEST_OCAMLVERSION` in the `Makefile` at the root of the Dune repo contains the current compiler version for which tests are written.

See also:

How Dune Uses Dune to Build Dune

6.3.3 Writing Tests

Most of our tests are written as expectation-style tests. While creating such tests, the developer writes some code and then lets the system insert the output produced during the code execution. The system puts it right next to the code in the source file.

Once you write and commit a test, the system checks that the captured output matches the one produced by a fresh code execution. When the two don't match, the test fails. The system then displays a diff between what was expected and what the code produced.

We write both our unit tests and integration tests in this way. For unit tests, we use the ppx_expect framework, where we introduce tests via `let%expect_test`, and `[%expect ...]` nodes capture expectations:

```
let%expect_test "<test name>" =
  print_string "Hello, world!";
  [%expect {|
    Hello, world!
  |}]
```

For integration tests, we use a system similar to Cram tests for testing shell commands and their behavior:

```
$ echo 'Hello, world!'
Hello, world!

$ false
[1]
```

(continues on next page)

(continued from previous page)

```
$ cat <<EOF
> multi
> line
> EOF
multi
line
```

See also:

actions_to_sh tests
An example of expect-tests.

mdx-stanza/locks.t
An example of Cram test.

When running Dune inside tests, the `INSIDE_DUNE` environment variable is set. This has the following effects:

- Change the default root detection behaviour to use the current directory rather than the top most `dune-project` / `dune-workspace` file.
- Be less verbose when Dune outputs a user message.
- Error reporting is deterministic by default.
- Prefer not to use a diff program for displaying diffs.

This list is not exhaustive and may change in the future. In order to find the exact behaviour, it is recommended to search for `INSIDE_DUNE` in the codebase.

Guidelines

As with any long running software project, code written by one person will eventually be maintained by another. Just like normal code, it's important to document tests, especially since test suites are most often composed of many individual tests that must be understood on their own.

A well-written test case should be easily understood. A reader should be able to quickly understand what property the test is checking, how it's doing it, and how to convince oneself that the test outcome is the right one. A well-written test makes it easier for future maintainers to understand the test and react when the test breaks. Most often, the code will need to be adapted to preserve the existing behavior; however, in some rare cases, the test expectation will need to be updated.

It's crucial that each test case makes its purpose and logic crystal clear, so future maintainers know how to deal with it.

When writing a test, we generally have a good idea of what we want to test. Sometimes, we want to ensure a newly developed feature behaves as expected. Other times, we want to add a reproduction case for a bug reported by a user to ensure future changes won't reintroduce the faulty behaviour. Just like when programming, we turn such an idea into code, which is a formal language that a computer can understand. While another person reading this code might be able to follow and understand what the code does step by step, it isn't clear that they'll be able to reconstruct the original developer's idea. Even worse, they might understand the code in a completely different way, which would lead them to update it incorrectly.

6.3.4 Setting Up Your Development Environment Using Nix

You can use Nix to setup the development environment. This can be done by running `nix develop` in the root of the Dune repository.

Note that Dune only takes OCaml as a dependency and the rest of the dependencies are used when running the test suite.

Running `nix develop` can take a while the first time, therefore it is advisable to save the state in a profile.

```sh
nix develop --profile nix/profiles/dune
```

And to load the profile:

```sh
nix develop nix/profiles/dune
```

This profile might need to be updated from time to time, since the bootstrapped version of Dune may become stale. This can be done by running the first command.

We have the following shells for specific tasks:

- `nix develop .#slim` for a dev environment with fewer dependencies that is faster to build.
- `nix develop .#slim-melange`: same as above, but additionally includes the `melange` and `mel` packages
- Building documentation requires `nix develop .#doc`.
- For running the Coq tests, you can use `nix develop .#coq`. NB: Coq native is not currently installed; this will cause some of the tests to fail. It's currently better to fallback to opam in this case.

6.3.5 Releasing Dune

Dune's release process relies on dune-release. Make sure you install and understand how this software works before proceeding. Publishing a release consists of two steps:

- Updating `CHANGES.md` to reflect the version being published.
- Running `$ make opam-release` to create the release tarball. Then publish it to GitHub and submit it to opam.

Major & Feature Releases

Given a new version `x.y.z`, a major release increments `x`, and a feature release increments `y`. Such a release must be done from the `main` branch. Once you publish the release, be sure to publish a release branch named `x.y`.

Point Releases

Point releases increment the `z` in `x.y.z`. Such releases are done from the respective `x.y` branch of the respective feature release. Once released, be sure to update `CHANGES.md` in the `main` branch.

6.3.6 Adding Stanzas

Adding new stanzas is the most natural way to extend Dune with new features. Therefore, we try to make this as easy as possible. The minimal amount of steps to add a new stanza is:

- Extend `Stanza.t` with a new constructor to represent the new stanza
- Modify `Dune_file` to parse the Dune language into this constructor
- Modify the rules to interpret this stanza into rules, usually done in `Gen_rules`

Versioning

Dune is incredibly strict with versioning of new features, modifications visible to the user, and changes to existing rules. This means that any added stanza must be guarded behind the version of the Dune language in which it was introduced. For example:

```
; ( "cram"
  , let+ () = Dune_lang.Syntax.since Stanza.syntax (2, 7)
    and+ t = Cram_stanza.decode in
    [ Cram t ] )
```

Here, Dune 2.7 introduced the Cram stanza, so the user must enable `(lang dune 2.7)` in their `dune` project file to use it.

`since` isn't the only primitive for making sure that versions are respected. See `Dune_lang.Syntax` for other commonly used functions.

Experimental & Independent Extensions

Sometimes, Dune's versioning policy is too strict. For example, it doesn't work in the following situations:

- When most Dune independent extensions only exist inside Dune for development convenience, e.g., build rules for Coq. Such extensions would like to impose their own versioning policy.
- When experimental features cannot guarantee Dune's strict backwards compatibility. Such features may dropped or modified at any time.

To handle both of these use cases, Dune allows the definition of new languages (with the same syntax). These languages have their own versioning scheme and their own stanzas (or fields). In Dune itself, `Syntax.t` represents such languages. Here's an example of how the Coq syntax is defined:

```
let coq_syntax =
  Dune_lang.Syntax.create ~name:"coq" ~desc:"the coq extension (experimental)"
    [ ((0, 1), `Since (1, 9)); ((0, 2), `Since (2, 5)) ]
```

The list provides which versions of the syntax are provided and which version of Dune introduced them.

Such languages must be enabled in the `dune` project file separately:

```
(lang dune 3.17)
(using coq 0.8)
```

If such extensions are experimental, it's recommended that they pass `~experimental:true`, and that their versions are below 1.0.

We also recommend that such extensions introduce stanzas or fields of the form `ext_name.stanza_name` or `ext_name.field_name` to clarify which extensions provide a certain feature.

6.3.7 Dune Rules

Creating Rules

A Dune rule consists of 3 components:

- *Dependencies* that the rule may read when executed (files, aliases, etc.), described by `'a Action_builder.t` values.
- *Targets* that the rule produces (files and/or directories), described by `'a Action_builder.With_targets.t'` values.
- *Action* that Dune must execute (external programs, redirects, etc.). Actions are represented by `Action.t` values.

Combined, one needs to produce an `Action.t Action_builder.With_targets.t` value to create a rule. The rule may then be added by `Super_context.add_rule` or a related function.

To make this maximally convenient, there's a `Command` module to make it easier to create actions that run external commands and describe their targets and dependencies simultaneously.

Loading Rules

Dune rules are loaded lazily to improve performance. Here's a sketch of the algorithm that tries to load the rule that generates some target file `t`.

- Get the directory that contains `t`. Call it `d`.
- Load all rules in `d` into a map from targets in that directory to rules that produce it.
- Look up the rule for `t` in this map.

To adhere to this loading scheme, we must generate our rules as part of the callback that creates targets in that directory. See the `Gen_rules` module for how this callback is constructed.

6.3.8 Documentation

User documentation lives in the `./doc` directory.

In order to build the user documentation, you must install python-sphinx, sphinx_rtd_theme and sphinx-copybutton.

Build the documentation with

```
$ make doc
```

For automatically updated builds, you can install sphinx-autobuild, and run

```
$ make livedoc
```

Nix users may drop into a development shell with the necessary dependencies for building docs `nix develop .#doc`.

Structure

For structure, we use the Diátaxis framework. The core idea is that documents should fit in one of the following categories:

- Tutorials, focused on learning
- How-to guides, focused on task solving
- Reference, focused on information
- Explanations, focused on understanding

Most features do not need a document in each category, but the important part is that a single document should not try to be in several categories at once.

ReStructured Text

For code blocks containing Dune files, use `.. code:: dune` and indent with 3 spaces. Use formatting consistent with how Dune formats Dune files (most importantly, do not leave orphan closing parentheses).

In a document that only contains Dune code blocks, it is possible to use the `.. highlight:: dune` directive to have `dune` be the default lexer, and then it is possible to use the `::` shortcut to end a line with a single `:` and start a code block. See the source of *Lexical Conventions* for an example.

For links, prefer references that use `:doc:` (link to a whole document) or `:term:` (link to a definition in the glossary) to `:ref:`.

Use the right lexers: - `dune` for `dune` and related files - `opam` for opam files - `console` for shell sessions and commands (start with `$`) - `cram` for cram tests

Style

Use American spelling.

Use Title Case for titles and headings (every word except "little words" like of, and, or, etc.).

For project names, use the following capitalization:

- **Dune** is the project, `dune` is the command. Files are called `dune` files.
- `dune-project` should always be written in monospace.
- **OCaml**
- **OCamlFormat**, and `ocamlformat` is the command.
- `odoc`, always in monospace.
- **opam**. Can be capitalised as Opam at the beginning of sentences only, as the official name is formatted opam. Even in titles, headers, and subheaders, it should be all lowercase: opam. The command is `opam`.
- **esy**. Can be capitalised as Esy.
- **Nix**. The command is `nix`.
- **Js_of_ocaml** can be abbreviated **JSOO**.
- **MDX**, rather than mdx or Mdx
- **PPX,** rather than ppx or Ppx; `ppxlib`
- **UTop,** rather than utop or Utop.

6.3.9 Vendoring

Dune vendors some code that it uses internally. This is done to make installing Dune easy as it requires nothing but an OCaml compiler as well as to prevent circular dependencies. Before vendoring, make sure that the license of the code allows it to be included in Dune.

The vendored code lives in the `vendor/` subdirectory. To vendor new code, create a shell script `update-<library>.sh`, that will be launched from the `vendor/` folder to download and unpack the source and copy the necessary source files into the `vendor/<library>` folder. Try to keep the amount of source code imported minimal, e.g., leave out `dune-project` files. For the most part, it should be enough to copy `.ml` and `.mli` files. Make sure to also include the license if there is such a file in the code to be vendored to stay compliant.

As these sources get vendored not as subprojects but parts of Dune, you need to deal with `public_name`. The preferred way is to remove the `public_name` and only use the private name. If that is not possible, the library can be renamed into `dune-private-libs.<library>`.

To deal with the modified `dune` files in `update-<library>.sh` scripts, you can commit the modified files to `dune` and make the `update-<library>.sh` script to use `git checkout` to restore the `dune` file.

For larger modifications, it is better to fork the upstream project in the ocaml-dune organisation and then vendor the forked copy in Dune. This makes the changes better visible and easier to update from upstream in the long run while keeping our custom patches in sync. The changes to the `dune` files are to be kept in the Dune repository.

It is preferable to cut out as many dependencies as possible, e.g., ones that are only necessary on older OCaml versions or build-time dependencies.

6.3.10 General Guidelines

Dune has grown to be a fairly large project that over time has acquired its own style. Below is an attempt to enumerate some important points of this style. These rules aren't axioms and we may break them when justified. However, we should have a good reason in mind when breaking them. Finally, the list isn't exhaustive by any means and is subject to change. Feel free to discuss anything in particular with the team.

- Parameter signatures should be self descriptive. Use labels when the types alone aren't sufficient to make the signature readable.

Bad:

```
val display_name : string -> string -> _ Pp.t
```

Good:

```
val display_name : first_name:string -> last_name:string -> _ Pp.t
```

- Avoid type aliases when possible. Yes, they might make some type signatures more readable, but they make the code harder to grep and make Merlin's inferred types more confusing.
- Every `.ml` file must have a corresponding `.mli`. The only exception to this rule is `.ml` files with only type definitions.
- Do not write `.mli` only modules. They offer no advantages to `.ml` modules with type definitions and one cannot define exceptions in `.mli` only modules
- Every module should have toplevel documentation that describes the module briefly. This is a good place to discuss its purpose, invariants, etc.
- Keep interfaces short & sweet. The less functions, types, etc., there are, the easier it is for users to understand, use, and ultimately modify the interface correctly. Instead of creating elaborate interfaces with the hope of future-proofing every use case, embrace change and make it easier to throw out or replace the interface.

Ideally the interface should have one obvious way to use it. A particularly annoying violator of this principle is the "logic-less chain of functions" helper. For example:

```
let foo t = bar t |> baz
```

If `bar` and `baz` are already public, then there's no need to add yet another helper to save the caller a line of code.

- Define bindings as close to their use site as possible. When they're far apart, reading code requires scrolling and IDE tools to understand the code.

Bad:

```
let dir = .. in
(* 50 odd lines or so that don't use [dir] *)
f dir
```

Good:

```
let dir = .. in
f dir
```

- A corollary to the previous guideline: keep the scope of bindings as small as possible.

Bad:

```
let x1 = f foo in let x2 = f bar in
let y1 = g foo in let y2 = g bar in
let dx = x2 -. x1 in
let dy = y2 -. y1 in
dx^2 +. dy^2
```

Good:

```
let dx =
  let x1 = f foo in let x2 = f bar in
  x2 -. x1
in
let dy =
  let y1 = g foo in let y2 = g bar in
  y2 -. y1
in
dx^2 +. dy^2
```

- Prefer `Code_error.raise` instead of `assert false`. The reader often has no idea what invariant is broken by the `assert false`. Kindly describe it to the reader in the error message.
- Avoid meaningless names like `x`, `a`, `b`, `f`. Try to find a more descriptive name or just inline it altogether.
- If a module `Foo` has a module type `Foo.S` and you'd like to avoid repeating its definition in the implementation and the signature, introduce an `.ml`-only module `Foo_intf` and write the `S` only once in there.
- Instead of introducing a type `foo`, consider introducing a module `Foo` with a type `t`. This is often the place to put functions related to `foo`.
- Avoid optional arguments. They increase brevity at the expense of readability and are annoying to grep. Furthermore, they encourage callers not to think at all about these optional arguments even if they often should.
- Avoid qualifying modules when accessing fields of records or constructors. Avoid it altogether if possible, or add a type annotation if necessary.

Bad:

```
let result = A.b () in
match result.A.field with
| B.Constructor -> ...
```

Good:

```
let result : A.t = A.b () in
match (result.field : B.t) with
| Constructor -> ...
```

- When constructing records, use the qualified names in in the record. Do not open the record. The local open syntax pulls in all kinds of names from the opened module and might shadow the values that you're trying to put into the record, leading to difficult debugging.

Bad; if `A.value` exists, it will pick that over `value`:

```
let value = 42 in
let record = A.{ field = value; other } in
...
```

Good:

```
let value = 42 in
let record = { A.field = value; other } in
...
```

- Stage functions explicitly with the `Staged` module.
- Do not raise `Invalid_argument`. Instead, raise with `Code_error.raise` which allows to attach more informative payloads than just strings.
- When ignoring the value of a let binding `let _ = ...`, we add type annotations to the ignored value `let (_ : t) = ...`. We do this convention because:
- We need to make sure we never ignore `Fiber.t` accidentally. Functions that return `Fiber.t` are always free of side effects so we need to bind on the result to force the side effect.
- Whenever a function is changed to return an error via its return value, we want the compiler to notify all the callers that need to be updated.
- To write a `to_dyn` function on a record type, use the following pattern. It ensures that the pattern matching will break when a field is added. To ignore a field, add `; d = _`, not `; _`.

```
let to_dyn {a; b; c} =
  Dyn.record
    [ ("a", A.to_dyn a)
    ; ("b", B.to_dyn b)
    ; ("c", C.to_dyn c)
    ]
```

- To write an equality function, use the following pattern (this applies to other kinds of binary functions). The same remarks about about pattern matching and ignoring fields apply.

```
let equal {a; b; c} t =
  A.equal a t.a &&
```

(continues on next page)

(continued from previous page)

```
  B.equal b t.b &&
  C.equal c t.c
```

Subjective Style Points

There's some stylistic decisions we made that don't have logical justification and are basically a matter of taste. Nevertheless, it's useful to follow them to keep the code consistent.

- Match patterns should be sorted by the length of their RHS when possible. Keep the shorter clauses near the top.
- If a module `Foo` defines a type `t`, all functions that take `t` in this module should have `t` as their first argument. This is the "t comes first" rule.
- Do not mix `|>` and `@@` in the same expression.
- Introduce bindings that will allow opportunities for record or label punning.
- Do not write inverted if-else expressions.

Bad:

```
(* try reading this out loud without short circuiting your brain *)
if not x then foo else bar
```

Good:

```
if x then bar else foo
```

- We prefer snake_casing identifiers. This includes the names of modules and module types.
- Avoid qualifying constructors and record fields. Instead, add type annotations to the type being matched on or being constructed, e.g.,

Bad:

```
let foo = Command.Args.S []
```

Good:

```
let (foo : _ Command.Args.t) = S []
```

6.3.11 Benchmarking

Dune Bench

You can benchmark Dune's performance by running `make bench`. This will run a subset of the Duniverse. If you are running the bench locally, make sure that you bootstrap since that is the executable that the bench will run.

The bench will build a specially selected portion of the Duniverse once, called a "clean build". Afterwards, the build will be run 5 more times and are termed the "Null builds".

In each run of the CI, there will be an `ocaml-benchmarks` status in the summary. Clicking `Details` will show a bench report.

The report contains the following information:

- The build times for Clean and Null builds

- The size of the `dune.exe` binary
- User CPU times for the Clean and Null builds
- System CPU times for the Clean and Null builds
- All the garbage collection stats apart from "forced collections" for Clean and Null builds

Pull requests that add new libraries are likely to increase the size of the dune binary.

Performance gains in Dune can be observed in the Clean and Null build times.

Memory usage can be observed in the garbage collection stats.

Inline Benchmarks

Certain performance-critical parts of Dune are benchmarked using the `inline_benchmarks` library. These benchmarks are run when running the tests. Their outputs are currently not recorded and are only used to detect performance regressions.

Build-Time Benchmarks

We benchmark the build time of Dune in every PR. The times can be found here:

https://bench.ci.dev/ocaml/dune?worker=autumn&image=bench.Dockerfile

Melange Bench

We also benchmark a demo Melange project's build time:

https://ocaml.github.io/dune/dev/bench/

Monorepo Benchmark

We benchmark the performance of Dune in building a large monorepo in every PR. The benchmark results can be found here:

https://bench.ci.dev/ocaml/dune/branch/main?worker=fermat&image=bench%2Fmonorepo%2Fbench.Dockerfile

You can find more information about these benchmarks here.

6.3.12 Formatting

When changing the formatting configuration, it is possible to add the reformatting commit to the `.git-blame-ignore-revs` file. The commit will disappear from blame views. It is also possible to configure `git` to have the same behavior locally.

It is recommended to edit that file in a second PR, to make sure that the referenced commit has not changed.

See also:

GitHub - Ignore commits in the blame view

INDEX

A

alias, 4

B

build context, 4
build context root, 4
build profile, 4
build target, 4

D

dialect, 5

E

environment, 4

I

installation, 4
installed world, 4

P

package, 4
placeholder substitution, 5
project, 4

R

root, 4

S

scope, 4

W

workspace, 4