OPAM: A Package Management System for OCaml
Developer Manual (version 1.2.1)

Thomas GAZAGNAIRE
thomas@gazagnaire.org
Louis GESBERT
louis.gesbert@ocamlpro.com

October 21, 2018

Contents

Overview

OPAM is a source-based package manager for OCaml. It supports multiple simultaneous compiler installations, flexible package constraints, and a Git-friendly development work-flow.

A package management system has typically two kinds of users: *end-users* who install and use packages for their own projects; and *packagers*, who create and upload packages. End-users want to install on their machine a consistent collection of *packages* – a package being a collection of OCaml libraries and/or programs. Packagers want to take a collection of their own libraries and programs and make them available to other developpers.

This document describes the design of OPAM to answer both of these needs.

Conventions

In this document, `$home`, `$opam`, and `$path` are assumed to be defined as follows:

- `$home` refers to the end-user home path, typically `/home/thomas/` on linux and `/Users/thomas/` on OSX.
- `$opam` refers to the filesystem subtree containing the client state. Default directory is `$home/.opam`.
- `$path` refers to a list of paths in the packager filesystem, where lives the collection of programs (`ocamlc`, `ocamldep`, `ocamlopt`, `ocamlbuild`, ...).

User variables are written in capital letters, prefixed by `$`. For instance package names will be written `$NAME`, package versions `$VERSION`, and the version of the ocaml compiler currently installed `$SWITCH`.

This document is organized as follows: Section 1 describes the core of OPAM, e.g. the management of packages. Section 3 describes how repositories are handled, Section 2 focus on compiler switches and finally Section 4 explain how packages can define configuration variables (which can be later used by the build system).

1 Managing Packages

1.1 State

The client state is stored on the filesystem, under `$opam`. All the configurations files, libraries and binaries related to a specific instance of the OCaml compiler in `$opam/$SWITCH`, where `$SWITCH` is the name of that specific compiler instance. See Section 2 for more details about compiler switches.

- `$opam/config` is the main configuration file. It defines the version of OPAM, the repository addresses and the current compiler version. The file format is described in §1.2.2.
- `$opam/packages/$NAME/$NAME.$VERSION/opam` is the specification for the package `$NAME` with version `$VERSION` (which might not be installed). The format of `opam` files is described in §1.2.3.
- `$opam/packages/$NAME/$NAME.$VERSION/descr` contains the description for the version `$VERSION` of package `$NAME` (which might not be installed). The first line of this file is the package synopsis.
- `$opam/packages/$NAME/$NAME.$VERSION/url` contains the upstream location for version `$VERSION` of package `$NAME` (which might not be installed). The format of `url` files is described in §1.2.4.
- `$opam/packages/$NAME/$NAME.$VERSION/files/` contains the optional overlay files on top of the upstream sources, for version `$VERSION` of package `$NAME` (which might not be installed). This files are copied in the build directory before building and installing a package.
- `$opam/archives/$NAME.$VERSION`opam.tar.gz+ contains the source archives for the version `$VERSION` of package `$NAME`. This archive might be a bit different from the upstream library as it might have been repackaged by OPAM to include the evenutal overlay files.
- `$opam/packages.dev.` contains cached information for development packages. OPAM uses it on update to check which package needs to be upgraded.

1.2 Files

1.2.1 General Syntax of Structured Files

Most of the files in the client and server states share the same syntax defined in this section.

Comments Two kinds of comments are available: the usual `(* ... *)` OCaml comment blocks and also `#` which discard everything until the end of the current line.

Base types The base types for values are:

- `BOOL` is either `true` or `false`
- `STRING` is a doubly-quoted OCaml string, for instance: `"foo"`, `"foo-bar"`, ...
- `SYMBOL` contains only non-letter and non-digit characters, for instance: `=`, `<=`, ... Some symbols have a special meaning and thus are not valid `SYMBOL`s: "`( ) [ ] { } :`".
- `IDENT` starts with a letter and is followed by any number of letters, digit and symbols, for instance: `foo`, `foo-bar`, ...

Compound types Types can be composed together to build more complex values:

- `X Y` is a space-separated pair of value.
- `X | Y` is a value of type either `X` or `Y`.
- `?X` is zero or one occurrence of a value of type `X`.
- `X+` is a space-separated list of values of at least one value of type `X`.
- `X*` is a space-separated list of values of values of type `X` (it might contain no value).

All structured files share the same syntax:

```
<file>  := <item>*

<item>  := IDENT : <value>
         | ?IDENT: <value>
         | IDENT STRING { <item>+ }

<value> := BOOL
```

```
        | INT
        | STRING
        | SYMBOL
        | IDENT
        | [ <value>+ ]
        | value { <value>+ }
```

1.2.2 Global Configuration File: `config`

`$opam/config` follows the syntax defined in §1.2.1 with the following restrictions:

```
<file> :=
    opam-version: "1.2"
    repositories: [ STRING+ ]
    switch: STRING
    ?jobs: INT
    ?solver: <single-command>
    ?solver-criteria: STRING
    ?solver-upgrade-criteria: STRING
    ?solver-fixup-criteria: STRING
    ?download-command: <single-command>
    ?download-jobs: INT

<single-command> := [ (<argument> ?{ <filter> })+ ]
```

- `opam-version` indicates the current OPAM repository format – normally corresponding to the OPAM minor version (`MAJOR.MINOR`)
- `repositories` contains the names of the currently configured repositories.
- `switch` is the name of the currently active OPAM switch.
- `jobs` is the maximum number of build processes that can be run simultaneously.
- `solver` is the external solver to call. The value may be either the single identifiers `aspcud` or `packup`, which have built-in support, or a command. The string variables `input`, `output` and `criteria` (only) are defined when evaluating this field.
- `solver-criteria`, `solver-upgrade-criteria` and `solver-fixup-criteria` are the optimisation criteria provided to the solver resp. in the default case (install requested packages at their latest version, minimising the impact on other packages), for global upgrades (minimise outdated packages) and for fixup (resolve dependencies while minimising changes).
- `download-command` will be called to fetch remote files over http(s) or ftp. The value may be either the single identifiers `curl` or `wget`, which have built-in support, or a custom command. Only the special variables `url`, `out`, `retries` (strings) and `compress` (bool) can be used in this field.
- `download-jobs` is the maximum number of simultaneous downloads (all remote hosts included)

1.2.3 Package Specification files: `opam`

`$opam/packages/$NAME/$NAME.$VERSION/opam` follows the syntax defined in §1.2.1 with the following restrictions:

```
<file> :=
    opam-version: "1.2"
    ?name:          STRING
    ?version:       STRING
    maintainer:     STRING
    ?authors:       [ STRING+ ]
```

```
    ?license:       STRING
    ?homepage:      STRING
    ?doc:           STRING
    ?bug-reports:   STRING
    ?dev-repo:      STRING
    ?tags:          [ STRING+ ]
    ?patches:       [ (STRING ?{ <filter> } )+ ]
    ?substs:        [ STRING+ ]
    ?build:         commands
    ?install:       commands
    ?build-doc:     commands
    ?build-test:    commands
    ?remove:        commands
    ?depends:       [ <and-formula(package-with-flags)> ]
    ?depopts:       [ (STRING ?{ <flags> })+ ]
    ?conflicts:     [ <package>+ ]
    ?depexts:       [ [[STRING+] [STRING+]]+ ]
    ?messages:      [ (STRING ?{ <filter> } )+ ]
    ?post-messages: [ (STRING ?{ <filter> } )+ ]
    ?available:     [ <filter> ]
    ?os:            [ <formula(os)>+ ]
    ?ocaml-version: [ <and-formula(constraint)>+ ]
    ?libraries:     [ STRING+ ]
    ?syntax:        [ STRING+ ]
    ?flags:         [ IDENT+ ]
    ?features:      [ (IDENT STRING <filter>)+ ]

<argument>       := STRING
                  | IDENT

<command>        := [ (<argument> ?{ <filter> })+ ] ?{ <filter> }

<commands>       := <command>
                  | [ <command>+ ]

<filter>         := <argument>
                  | !<argument>
                  | <argument> <comp> <argument>
                  | formula(<filter>)

<formula(x)>     := <formula(x)> '&' <formula(x)>
                  | <formula(x)> '|' <formula(x)>
                  | ( <formula(x)> )
                  | !<formula(x)>
                  | <x>

<package>        := STRING
                  | STRING { <and-formula(constraint)> }

<flags>          := IDENT
                  | IDENT '&' <flags>

<package-with-flags>
                 := STRING
                  | STRING { <flags> }
                  | STRING { ?(<flags> '&') <and-formula(constraint)> }

<constraint>     := <comp> STRING
<comp>           := '=' | '<' | '>' | '>=' | '<=' | '!='

<and-formula(x)> := <x> <and-formula(x)>
```

```
                  | <formula(x)>

<or-formula(x)>  := <x> <or-formula(x)>
                  | <package(x)>

<os>             := STRING
                  | '!' STRING
```

- `opam-version` specifies the file format version, it should be the current OPAM version in the format `MAJOR.MINOR` (*i.e.* with patch version omitted)
- `name`, `version` contain resp. `$NAME` and `$VERSION`, specifying the package. Both fields are optional when they can be inferred from the directory name (*e.g.* when the file sits in the repository).
- `maintainer` is a mandatory contact address for the package maintainer (the format `"name <email>"` is allowed).
- `authors` is a list of strings listing the original authors of the software.
- `license` is the abbreviated name of the license under which the source software is available.
- `homepage`, `doc`, `bug-reports` are URLs pointing to the related pages for the package.
- `dev-repo` is the URL of the package's source repository, which may be useful for developers: not to be mistaken with the URL file, which points to the specific packaged version. It is typically a version-control address, and follows the format allowed for `TARGET` by `opam pin add`. You can use an address of the form `vcs+scheme://xxx` to specify the version control system to use, *e.g.* `git+ssh://address` or `hg+https://address`. (Note: this has been added in OPAM 1.2.1)
- `tags` contains an optional list of semantic tags used to classify the packages. The `"org:foo"` tag is reserved for packages officially distributed by organization "foo".
- `patches` is a list of files relative to the project source root (often added through the `files/` metadata subdirectory). The listed patch files will be applied sequentially to the source like with the `patch` command, after having gone through *variable interpolation expansion* like files listed in `substs`. Patches may be applied conditionally by adding *filters*.
- `substs` contains a list of files relative to the project source root. Variable interpolations will be expanded on these files before the build takes place (see §4.3.1 for the file format and §4 for the semantic of file substitution).
- `build` is the list of commands that will be run in order to compile the package. Any command is allowed, but these should write exclusively to the package's source directory (given as `CWD` to the command), be non-interactive and perform no network i/o. All libraries, syntax extensions, binaries, platform-specific configuration and install files (`$NAME.config` and `$NAME.install`, see §1.2.2 and §2.2.3) should be produced within the source directory subtree during this step. Each command is provided as a list of terms (a command and zero or more arguments) ; individual terms as well as full commands can be made conditional by adding *filters*: they will be ignored if the *filter* evaluates to `false` or is undefined.

  Additionally, strings in each term undergo variable expansion: `"%{foo}%"` is replaced by the contents of variable `foo`. Variable `foo`, if undefined globally, implicitly refers to the package being defined, in the version being defined (while `$NAME:foo` refers to the *installed* version of the package). See §4.2 and the output of the command `opam config list` for a list of predefined variables.

  Filters typically refer to the `os` variable, like in the following example:

```
build: [
  ["mv" "Makefile.unix"  "Makefile"] {os != "win32"}
  ["mv" "Makefile.win32" "Makefile"] {os  = "win32"}
  [make]
]
```

- `install` follows the exact same format as `build`, but should only be used to move products of `build` from the build directory to their final destination under the current `prefix`, and adjust some configuration files there when needed. Commands in `install` are executed sequentially after the build is finished. These commands should only write to subdirectories of `prefix`, without altering the source directory itself.

  This field contains typically just `[make "install"]`. It is recommended to prefer the usage of a `$NAME.install` file and omit the `install` field.

- `build-doc` and `build-test` follow the same specification as the `build` field. They are processed after the build phase when documentation or tests have been requested.

- `remove` follows the same format as `build`, and is used to uninstall the package. It should be the reverse operation of `install`, and absent when `install` is.

- `depends` describes the requirements on other packages for this package to be built and installed. It contains a list of formulas over package names, optionally parametrized by version constraints, e.g.:

  - A simple package name: `"foo"`;
  - A package name with version constraints: `"foo" {>= "1.2" & <= "3.4"}`

  Elements of the list are implicitely ANDed: `["foo" "bar"]` is equivalent to `["foo" & "bar"]`, and `"foo" {<= "1.2"} ("bar" | "gna" {= "3.14"})` is to be understood as "`"foo"` *is required, at a version lesser or equal to* 1.2*, as well as one of* "`bar`" *or* "`gna`" *version* 3.14"

  Additionally, the version constraints may be prefixed by *dependency flags*. These are one of `build`, `test` and `doc` and limit the meaning of the dependency:

  - `build` dependencies are no longer needed at run-time: they won't trigger recompilations of your package.
  - `test` dependencies are only needed when building tests (by instructions in the `build-test` field)
  - likewise, `doc` dependecies are only required when building the package documentation

  For example:

  ```
  depends: [
    "foo" {build}
    "bar" {build & doc}
    "baz" {build & >= "3.14"}
  ]
  ```

- `depopts`, for "optional dependencies", is similar to `depends` in format, with some restrictions. It contains packages that are *used*, if present, by the package being defined, either during build or runtime, but that are not *required* for its installation. The implementation uses this information to define build order and trigger recompilations, but won't automatically install *depopts* when installing your package.

  The optional dependencies may have *dependency flags*, but they may not specify version constraints nor formulas. `depopts` can be combined with `conflicts` to add version constraints on the optional dependencies. Note that this changed in OPAM 1.2: previously these constraints could be put in `depopts`. You should now write `depopts: "foo"`, `conflicts: "foo" {< "2"}` rather than `depopts: "foo" {>= "2"}` which had a non-obvious meaning.

- `conflicts` is a list of package names with optional version constraints indicating that the current package can't coexist with some packages or some specific versions.

- `depexts`, for "external dependencies", is a list that can be used for describing the dependencies of the package towards software or packages external to the OPAM ecosystem, for various system. It contains pairs of lists of the form `[ predicates ext-packages ]`. `predicates` is used to select the element of the list based on the current system: it is a list of tags (strings) that can correspond to the OS, architecture or distribution. The `predicates` is used as a conjunction: the pair will only be selected when *all* tags are active. The resulting `ext-packages` should be identifiers of packages recognised by the system's package manager.

  There is currently no definite specification for the precise tags you should use, but the closest thing is the *opam-depext* project (https://github.com/OCamlPro/opam-depext). The `depexts` information can be retrieved through the `opam list -external` command.

- `messages` (since version 1.0.1) is used to display an additional (one-line) message when prompting a solution implying the given package. The typical use-case is to tell the user that some functionality will not be available as some optional dependencies are not installed.

- `post-messages` (since version 1.1.0) allows to print specific messages to the user after the end of installation. The special boolean variable `failure` is defined in the scope of the filter, and can be used to print messages in case there was an error (typically, a hint on how it can be resolved, or a link to an open issue). `success` is also defined as syntactic sugar for `!failure`.

- `available` field (since version 1.1.0) can be used to add constraints on the OS and OCaml versions currently in use, using the built-in `os` and `ocaml-version` variables. In case the filter is not valid, the package is disabled. The `os` and `ocaml-version` fields are deprecated, please use `available` instead in newly created packages.

- `libraries` and `syntax` contain the libraries and syntax extensions defined by the package. See Section 4 for more details.

- `flags` allow a limited set of idents that give special handling instructions for the package:

  `light-uninstall` the package's uninstall instructions don't require the package source. This is currently inferred when the only uninstall instructions have the form 'ocamlfind remove...', but making it explicit is preferred (since OPAM 1.2.0).

  `verbose` when this is present, the stdout of the package's build and install instructions will be printed to the user (since OPAM 1.2.1).

  `plugin` the package installs a program named `opam-$NAME`. OPAM will hand over to it when run with subcommand `$NAME`, automatically installing it if necessary (since OPAM 1.2.2).

  Since some of those wheren't known in OPAM 1.2.0, which will complain about them, they shouldn't be used in the 1.2 branch. Using a `flags:foo` *tag* instead is thus allowed as a compatible way to activate flag `foo` on 1.2.x versions.

- `features` is currently experimental and shouldn't be used on the main package repository. It allows to define custom variables that better document what *features* are available in a given package build. Each feature is defined as an identifier, a documentation string, and a filter expression. The filter expression can evaluate to either a boolean or a string, and the defined identifier can be used as a variable in any filter (but recursive features are not allowed and will return `undefined`).

  This is typically useful to pass appropriate flags to `./configure` scripts, depending on what is installed.

1.2.4 URL files: `url`

The syntax of `url` files follows the one described in §1.2.1 with the following restrictions:

```
<file> :=
    ?src:       STRING
    ?archive:   STRING
    ?http:      STRING
```

```
?local:     STRING
?git:       STRING
?darcs:     STRING
?hg:        STRING
?mirrors:   [ STRING+ ]
?checksum:  STRING
```

`src`, `archive`, `http`, `local`, `git`, `hg`, `darcs` are the location where the package upstream sources can be downloaded. It can be one of:

- A directory on the local file system (which will be copied to the build directory). Use `local`.
- An archive file on the local file system (which will be unpacked into the build directory). Use `local`.
- An archive file at a URL that is understood by either curl or wget (which will be fetched using either curl (if that available) or wget (if curl is not available) and unpacked into the build directory). Use `http`.
- a version-controlled repository under git, darcs or hg, or a specific commit, tag or branch in that repository if the string ends by `#<SHA1>` or `#<tag-name>` or `#<branch-name>`. Use `git`, `hg` or `darcs`.
- OPAM will try to guess the source kind if you use `src` or `archive`. It shouldn't be used for version control systems, for which the above are preferred.

`mirrors`, if specified, is assumed to be a list of addresses of the same kind as the primary address. Mirrors will be tried in order in case the download from the primary upstream fails.

1.3 Commands

1.3.1 Creating a Fresh Client State

When an end-user starts OPAM for the first time, he needs to initialize `$opam/` in a consistent state. In order to do so, he should run:

```
$ opam init [--kind $KIND] $REPO $ADDRESS [--comp $VERSION]
```

Where:

- `$KIND` is the kind of OPAM repository (default is `http`);
- `$REPO` is the name of the repository (default is `default`); and
- `$ADDRESS` is the repository address (default is `http://opam.ocamlpro.com/pub`).
- `$COMP` is the compiler version to use (default is the version of the compiler installed on the system).

This command will:

1. Create the file `$opam/config` (as specified in §1.2.2)
2. Create an empty `$opam/$SWITCH/installed` file, `$SWITCH` is the version from the OCaml used to compile `$opam`. In particular, we will not fail now if there is no `ocamlc` in `$path`.
3. Initialize `$opam/repo/$REPO` by running the appropriate operations (depending on the repository kind).
4. Copy all OPAM and description files (ie. copy every file in `$opam/repo/$REPO/packages/` to `$opam/packages/`).
5. Create `$opam/repo/package-index` and for each version `$VERSION` of package `$NAME` appearing in the repository, append the line `'$REPO $NAME $VERSION'` to the file.
6. Create the empty directories `$opam/archives`, `$opam/$SWITCH/lib/`, `$opam/$SWITCH/bin/` and `$opam/$SWITCH/doc/`.

1.3.2 Listing Packages

When an end-user wants to have information on all available packages, he should run:

```
$ opam list
```

This command will parse `$opam/$SWITCH/installed` to know the installed packages, and `$opam/packages/$NAME/$NA` to get all the available packages. It will then build a summary of each packages. The description of each package will be read in `$opam/packages/$NAME/$NAME.$VERSION/descr` if it exists.

For instance, if `batteries` version `1.1.3` is installed, `ounit` version `2.3+dev` is installed and `camomille` is not installed, then running the previous command should display:

```
batteries   1.1.3  Batteries is a standard library replacement
ounit     2.3+dev  Test framework
camomile       --  Unicode support
```

1.3.3 Getting Package Info

In case the end-user wants a more details view of a specific package, he should run:

```
$ opam info $NAME
```

This command will parse `$opam/$SWITCH/installed` to get the installed version of `$NAME`, will process `$opam/repo/index` to get the repository where the package comes from and will look for `$opam/packages/$NAME/$NAME.$VERSION/opam` to get available versions of `$NAME`. It can then display:

```
package: $NAME
version: $VERSION
versions: $VERSION1, $VERSION2, ...
libraries: $LIB1, $LIB2, ...
syntax: $SYNTAX1, $SYNTAX2, ...
repository: $REPO
description:
  $SYNOPSIS

  $LINE1
  $LINE2
  $LINE3
  ...
```

1.3.4 Installing a Package

When an end-user wants to install a new package, he should run:

```
$ opam install $NAME
```

This command will:

1. Compute the transitive closure of dependencies and conflicts of packages using the dependency solver (see §1.3.8). If the dependency solver returns more than one answer, the tool will ask the user to pick one, otherwise it will proceed directly. The dependency solver should also mark the packages to recompile.
2. The dependency solver sorts the collections of packages in topological order. Then, for each of them do:
   (a) Check whether the package is already installed by looking for the line `$NAME $VERSION` in `$opam/$SWITCH/installed`. If not, then:

(b) Look into the archive cache to see whether it has already been downloaded. The cache location is: `$opam/archives/$NAME.VERSION.tar.gz`

(c) If not, process `$opam/repo/index/` (see 3) to get the repository `$REPO` where the archive is available and then ask the repository to download the archive if necessary.

Once this is done, the archive might become available in `$opam/repo/$REPO/archives/`. It it is, copy it in the global state (in `$opam/archives`). If it is not, download it upstream by looking at `$opam/packages/$NAME/$NAME.$VERSION/url` and add the optional overlay files located in `$opam/packages/$NAME/$NAME.$VERSION/files`. Note: this files can be overwritten by anything present in `$opam/$SWITCH/overlay/$NAME/`.

(d) Decompress the archive into `$opam/$SWITCH/build/$NAME.$VERSION/`.

(e) Substitute the required files.

(f) Run the list of commands to build the package with `$opam/$SWITCH/bin` in the path.

(g) Process `$opam/$SWITCH/build/$NAME.$VERSION/$NAME.install` to install the created files. The file format is described in §2.2.3.

(h) Install the installation file `$opam/$SWITCH/build/$NAME.$VERSION/$NAME.install` in `$opam/$SWITCH/instal` and the configuration file `$opam/$SWITCH/build/$NAME.$VERSION/$NAME.config` in `$opam/$SWITCH/config/`

1.3.5 Updating Index Files

When an end-user wants to know what are the latest packages available, he will write:

```
$ opam update
```

This command will follow the following steps:

- Update each repositories in `$opam/config`.
- For each repositories in `$opam/config`, call the corresponding update scripts. Then, look at the difference between the digest of files in the global state and in each repositories, and fix the discrepancies (while notifying the user) if necessary. Here, the order in which the repositories are specified is important: the first repository containing a given version for a package will be the one providing it (this can be changed manually by editing `$opam/repo/index` later).
- For each installed pinned and development packages, look at what changed upstream and if something has changed, update `$opam/$SWITCH/reinstall` accordingly (for each compiler version `$SWITCH`).
- Packages in `$opam/$SWITCH/reinstall` will be reinstalled (or upgraded if a new version is available) on the next `opam upgrade` (see §1.3.6), with `$SWITCH` being the current compiler version when the upgrade command is run.

1.3.6 Upgrading Installed Packages

When an end-user wants to upgrade the packages installed on his host, he will write:

```
$ opam upgrade
```

This command will:

- Call the dependency solver (see §1.3.8) to find a consistent state where **most** of the installed packages are upgraded to their latest version. Moreover, packages listed in `$opam/$SWITCH/reinstall` will be reinstalled (or upgraded if a new version is available). It will install each non-installed packages in topological order, similar to what it is done during the install step, See §1.3.4.
- Once this is done the command will delete `$opam/$SWITCH/reinstall`.

1.3.7 Removing Packages

When the user wants to remove a package, he should write:

```
$ opam remove $NAME
```

This command will check whether the package `$NAME` is installed, and if yes, it will display to the user the list packages that will be uninstalled (ie. the transitive closure of all forward-dependencies). If the user accepts the list, all the packages should be uninstalled, and the client state should be let in a consistent state.

1.3.8 Dependency Solver

Dependency solving is a hard problem and we do not plan to start from scratch implementing a new SAT solver. Thus our plan to integrate (as a library) the Debian depency solver for CUDF files, which is written in OCaml.

- the dependency solver should run on the client; and
- the dependency solver should take as input a list of packages (with some optional version information) the user wants to install, upgrade and remove and it should return a consistent list of packages (with version numbers) to install, upgrade, recompile and remove.

2 Managing Compiler Switches

OPAM is able to manage concurrent compiler installations.

2.1 State

Compiler descriptions are stored in two files:

- `$opam/compilers/$VERSION/$VERSION[+$TAG]/comp` contains the meta-data for a given compiler.`$VERSION` is the compiler version and `$TAG` an optional tag (such as `4.01+fp`). The format of `.comp` file is described in 2.2.2.
- (optional) `$opam/compilers/$VERSION/$VERSION[+$TAG]/descr` contains the description of that compiler. `$VERSION` is the compiler version and `$TAG` an optional tag (such as `4.01+fp`). The first line of that line is used as synopsis and is displayed with `opam repository list`.

Switch-related meta-data are stored under `$opam/$SWITCH/`:

- `$opam/$SWITCH/installed` is the list of installed packages for the compiler instance `$SWITCH`. The file format is described in §2.2.1.
- `$opam/$SWITCH/installed.root` is the list of installed packages roots for the compiler instance `$SWITCH`. The file format is described in §2.2.1. A package root has been explicitly installed by the user.
- `$opam/$SWITCH/config/$NAME.config` is a platform-specific configuration file of for the installed package `$NAME` with the compiler instance `$SWITCH`. The file format is described in §1.2.2.
- `$opam/$SWITCH/install/$NAME.install` is a platform-specific package installation file for the installed package `$NAME` with the compiler instance `$SWITCH`. The file format is described in §2.2.3.
- `$opam/$SWITCH/lib/$NAME/` contains the libraries associated to the installed package `$NAME` with the compiler instance `$SWITCH`.
- `$opam/$SWITCH/doc/$NAME/` contains the documentation associated to the installed package `NAME` with the compiler instance `$SWITCH`.

- `$opam/$SWITCH/bin/` contains the program files for all installed packages with the compiler instance `$SWITCH`.
- `$opam/$SWITCH/build/$NAME.$VERSION/` is a tempory folder used to build package `$NAME` with version `$VERSION`, with compiler instance `$SWITCH`.
- `$opam/$SWITCH/reinstall` contains the list of packages which has been changed upstream since the last upgrade. This can happen for instance when a packager uploads a new archive or fix the `opam` file for a specific package version. Every package appearing in this file will be reinstalled (or upgraded if a new version is available) during the next upgrade when the current instance of the compiler is `$SWITCH`. The file format is similar to the one described in §2.2.1.
- `$opam/$SWITCH/pinned` contains the list of pinned packages. The file format is described in §2.2.4.
- `$opam/$SWITCH/overlay/$NAME/` contains overlay files for package `$NAME`, for the switch `$SWITCH`. These possible overlay files are `opam`, `url`, `descr` or the directory `files/`. If one of this file (or directory) is present, the file (or directory) will be used instead of the global one.
- `$opam/$SWITCH/packages.dev` contains cached information for dev and pinned packages. OPAM uses it on update to check which package needs to be upgraded.

2.2 Files

2.2.1 Package List: `installed` and `reinstall`

The following configuration files: `$opam/$SWITCH/installed`, `$opam/$SWITCH/reinstall`, and `$opam/repo/$REPO/upda` follow a very simple syntax. The file is a list of lines which contains a space-separated name and a version. Each line `$NAME $VERSION` means that the version `$VERSION` of package `$NAME` has been compiled with the compiler instance `$SWITCH` and has been installed on the system in `$opam/$SWITCH/lib/$NAME` and `$opam/$SWITCH/bin/`.

For instance, if `batteries` version `1.0+beta` and `ocamlfind` version `1.2` are installed, then `$opam/$SWITCH/installed` will contain:

```
batteries 1.0+beta
ocamlfind 1.2
```

2.2.2 Compiler Description Files: `comp`

The syntax of `comp` files follows the one described in §1.2.1 with the following restrictions:

```
<file> :=
    opam-version: "1.2"
    name:       STRING
    ?src:       STRING
    ?archive:   STRING
    ?http:      STRING
    ?local:     STRING
    ?git:       STRING
    ?darcs:     STRING
    ?hg:        STRING
    ?make:      STRING+ ]
    ?build:     [[STRING+]]
    ?patches:   [ STRING+ ]
    ?configure: [ STRING+ ]
    ?packages:  <package>+
    ?preinstalled: BOOL
    ?env:       [ <env>+ ]
```

```
<env>    := IDENT <eq> STRING
<eq>     := '=' | '+=' | '=+' | ':=' | '=:'
```

- `name` is the compiler name, it should be identical to the filename.
- `src`, `archive`, `http`, `local`, `git`, `hg`, `darcs` are the location where the compiler sources can be downloaded. It can be:
  - A directory on the local file system (which will be linked or, if file system doesn't support links, copied to the build directory). Use `local`.
  - An archive file on the local file system (which will be unpacked into the build directory). Use `local`.
  - An archive file at a URL that is understood by either curl or wget (which will be fetched using either curl (if that available) or wget (if curl is not available) and unpacked into the build directory). Use `http`.
  - a version-controlled repository under git, darcs or hg, or a specific commit, tag or branch in that repository if the string ends by `#<SHA1>` or `#<tag-name>` or `#<branch-name>`. Use `git`, `hg` or `darcs`.
  - OPAM will try to guess the source kind if you use `src` or `archive`.
- `patches` are optional patch addresses, available via `http` or locally on the filesystem.
- `configure` are the optional flags to pass to the configure script. The order is relevant: `-prefix=$opam/SWITCH/` will be automatically added at the end to these options. Remark that if these flags contain `-bindir`, `-libdir`, and `-mandir`, then every `-prefix` will be ignored by `configure`.
- `make` are the flags to pass to `make`. It must at least contain some target like `world` or `world.opt`. If `make` is not present, OPAM will execute all the commands listed in `build`.
- `packages` is the list of packages to install just after the compiler installation finished.
- `preinstall` is `true` when the version of the compiler available in the path is the same as `name`.
- `env` is the list of environment variables to set in the given compiler switch:
  - `VAR = "value"` set the variable to the given value;
  - `VAR += "value"` prepend the given value to the variable;
  - `VAR =+ "value"` append the given value to the variable;
  - `VAR := "value"` prepend the given value to the variable, separated by a colon. If the variable was empty, add the colon anyway.
  - `VAR =:  "value"` append the given value to the variable, separated by a colon. If the variable was empty, add the colon anyway.

For instance the file, `3.12.1+memprof.comp` describes OCaml, version 3.12.1 with the memory profiling patch enabled:

```
opam-version: "1.2"
name:         "3.12.1"
src:          "http://caml.inria.fr/pub/distrib/ocaml-3.12/ocaml-3.12.1.tar.gz"
make:         [ "world" "world.opt" ]
patches:      [ "http://bozman.cagdas.free.fr/documents/ocamlmemprof-3.12.0.patch" ]
env:          [ CAML_LD_LIBRARY_PATH = "%{lib}%/stublibs" ]
```

And the file `trunk-g-notk-byte.comp` describes OCaml from SVN trunk, with no *tk* support and only in bytecode, and all the libraries built with `-g`:

```
opam-version: "1.2"
name:         "trunk-g-notk-byte"
src:          "http://caml.inria.fr/pub/distrib/ocaml-3.12/ocaml-3.12.1.tar.gz"
configure:    [ "-no-tk" ]
make:         [ "world" ]
bytecomp:     [ "-g" ]
bytelink:     [ "-g" ]
env:          [ CAML_LD_LIBRARY_PATH = "%{lib}%/stublibs" ]
```

2.2.3 Package installation files: `*.install`

`$opam/$SWITCH/install/NAME.install` follows the syntax defined in §1.2.1 with the following restrictions:

```
<file> :=
    ?lib:        [ <mv>+ ]
    ?libexec:    [ <mv>+ ]
    ?bin:        [ <mv>+ ]
    ?sbin:       [ <mv>+ ]
    ?toplevel:   [ <mv>+ ]
    ?share:      [ <mv>+ ]
    ?share_root: [ <mv>+ ]
    ?etc:        [ <mv>+ ]
    ?doc:        [ <mv>+ ]
    ?misc:       [ <mv>+ ]
    ?stublibs:   [ <mv>+ ]
    ?man:        [ <mv>+ ]

<mv> := STRING
      | STRING { STRING }
```

- Files listed under `lib` are copied into `$opam/$SWITCH/lib/$NAME/`.
- Files listed under `libexec` are copied into `$opam/$SWITCH/lib/$NAME/`, but with execution permission.
- Files listed under `bin` are copied into `$opam/$SWITCH/bin/`.
- Files listed under `sbin` are copied into `$opam/$SWITCH/sbin/`.
- Files listed under `doc` are copied into `$opam/$SWITCH/doc/$NAME/`.
- Files listed under `share` are copied into `$opam/$SWITCH/share/$NAME/`.
- Files listed under `share_root` are copied into `$opam/$SWITCH/share/`.
- Files listed under `etc` are copied into `$opam/$SWITCH/etc/$NAME/`.
- Files listed under `toplevel` are copied into `$opam/$SWITCH/lib/toplevel/`.
- Files lister under `stublibs` are copied into `$opam/$SWITCH/lib/stublibs/`
- Files listed under `man` are copied into `$opam/$SWITCH/man/manN`, where `N` is taken from the extension of the source file if 1...8 (not considering a `.gz` extension). If the destination is explicit, it will be taken relative to `$opam/$SWITCH/man`, e.g. `man: [ "foo.1" ]` is equivalent to `man: [ "foo.1" {"man1/foo.1"} ]`
- Files listed under `misc` are processed as follows: for each pair `$SRC { $DST }`, the tool asks the user if he wants to install `$SRC` to the absolute path `$DST`.

General remarks:

- You control where the files are copied under the given prefix by using the optional argument. For instance: `doc: [ "_build/foo.html" {"foo/index.html"} ]` will copy the given HTML page under `$opam/$SWITCH/doc/$NAME/foo/index.html`.
- OPAM will try to install all the files in sequence, and it will fail in case a source filename is not available. To tell OPAM a source filename might not be generated (because of byte/native constraints or because of optional dependencies) the source filename should be prefixed with `?`.
- It is much cleaner if the underlying build-system can generate the right `$NAME.install` files, containing the existing files only.
- File permissions are set to 0644, except for `libexec`, `bin`, `sbin`, `stublibs` for which it's 0755. `umask` may apply.
- These files can be used for installation outside of the scope of OPAMusing the `opam-installer` program shipped with OPAM.

2.2.4 Pinned Packages: `pinned`

`$opam/$SWITCH/pinned` contains a list of lines of the form:

```
<name> <kind> <path>
```

- `<name>` is the name of the pinned package
- `<kind>` is the kind of pinning. This could be `version`, `local`, `git` or `darcs`.
- `<path>` is either the version number (if kind is `version`) or the path to synchronize with.

2.3 Commands

2.3.1 Switching Compiler Version

If the user wants to switch to another compiler version, he should run:

```
$ opam switch [-alias-of $COMPILER] $SWITCH
```

This command will:

- If `$COMPILER` is not set, set it to `$SWITCH`
- Look for an existing `$opam/$COMPILER` directory.
  - If it exists, then change the `ocaml-version` content to `$COMPILER` in `$opam/config`.
  - If it does not exist, look for an existing `$opam/compilers/$VERSION/$VERSON[+$TAG]/comp`, where `$COMPILER = $VERSION[+TAG]` If the file does not exists, the command will fail with a well-defined error.
  - If the file exist, then build the new compiler with the right options (and pass `--prefix $opam/$SWITCH` to `./configure`) and initialize everything in `$opam/` in a consistent state as if "`opam init`" has just been called.
  - Update the file `$opam/aliases` with the line `$SWITCH $COMPILER`

3 Managing Repositories

3.1 State

Configuration files for the remote repository `REPO` are stored in `$opam/repo/$REPO`. Repositories can be of different kinds (stored on the local filesystem, available via HTTP, stored under git, ...); they all share the same filesystem hierachy, which is updated by different operations, depending on the repository kind.

- `$opam/repo/$REPO/version` contains the minimum version of OPAM which can read that repository meta-data.
- `$opam/repo/$REPO/config` contains the configuration of the repository `$REPO`.
- `$opam/repo/$REPO/packages/$PREFIX/$NAME.$VERSION/opam` is the `opam` file for the package `$NAME` with version `$VERSION` (which might not be installed) with any possible `$PREFIX`. The format of `opam` files is described in §1.2.3.
- (optional) `$opam/repo/$REPO/packages/$PREFIX/$NAME.$VERSION/descr` contains the textual description for the version `$VERSION` of package `$NAME` (which might not be installed) with any possible `$PREFIX` – it should be in the same loacation as the corresponding `opam` file. The first line of this file is the package synopsis.
- (optional) `$opam/repo/$REPO/packages/$PREFIX/$NAME.$VERSION/url` contains the upstream location for the version `$VERSION` of package `$NAME` (which might not be installed) with any possible `$PREFIX` – it should be in the same loacation as the corresponding `opam` file. The format of `url` files is described in §1.2.4.
- (optional) `$opam/repo/$REPO/packages/$PREFIX/$NAME.$VERSION/files/` contains the overlay files for the version `$VERSION` of package `$NAME` (which might not be installed) with any possible `$PREFIX` – it should be in the same loacation as the corresponding `opam` file. The overlay files are added to the build directory when a package is built and installed.
- (optional) `$opam/repo/$REPO/archives/$NAME.$VERSION.tar.gz` contains the source archives for the version `$VERSION` of package `$NAME`. This folder is populated when a package needs to be downloaded and that the given repository expose such a file. If the file is not present, OPAM will download the package from the upstream sources.

3.2 Files

3.2.1 Index of packages

`$opam/repo/index` follows a very simple syntax: each line of the file contains a space separated list of words `$NAME.$VERSION $REPO+` specifying that all the version `$VERSION` of package `$NAME` is available in the remote repositories `$REPO+`. The file contains information on all available packages (e.g. not only on the installed one).

For instance, if `batteries` version `1.0+beta` is available in the `testing` repository and `ocamlfind` version `1.2` is available in the `default` and testing repositories (where `default` is one being used), then `$opam/repo/index` will contain:

```
batteries.1.0+beta testing
ocamlfind.1.2 default testing
```

3.3 Commands

3.3.1 Managing Remote Repository

An user can manage remote repositories, by writing:

```
$ opam repository list # 'opam repository' works as well
$ opam repository add [--kind $KIND] $REPO $ADRESS
$ opam repository remove $REPO
```

- `list` lists the current repositories by looking at `$opam/config`
- `add [--kind $KIND] $REPO $ADDRESS` initializes `$REPO` as described in §1.3.1.
- For distributed version controlled repository, the user can use `url#hash` or `url#name` where `hash` is a given revision name and `name` it the name of a tag or a branch.
- `remove $REPO` deletes `$opam/repo/$REPO` and removes `$REPO` from the `repositories` list in `$opam/config`. Then, for each package in `$opam/repo/index` it updates the link between packages and repositories (ie. it either deletes packages or symlink them to the new repository containing the package).

4 Managing Configurations

4.1 State

4.2 Variables

Variables can appear in some fields of `opam` files (see §1.2.3), and in substition files (see below). There are three kinds. Note that this list is often out of date, you should check `opam config list` for a complete overview.

1. Global variables, corresponding to the current configuration
   - `opam-version` the running version of OPAM
   - `ocaml-version` the currently used version of OCaml
   - `switch` the user-defined name (alias) of the current switch
   - `compiler` the full OPAM name of the current OCaml compiler
   - `preinstalled` whether the compiler was preinstalled on the system or managed by OPAM
   - `ocaml-native` whether native compilation is available (`ocamlopt`)
   - `ocaml-native-tools` whether the `.opt` variant of the compilers are available (**not** the `ocamlopt` native compiler itself)
   - `ocaml-native-dynlink` whether native dynlink is available
   - `jobs` the configured number of parallel jobs to run
   - `arch` the host architecture, as returned by `uname -m`
2. Package-local variables. Variable `$VAR` of package `$NAME` is normally accessed using `%{$NAME:$VAR}%`, which can be abbreviated as `%{$VAR}%` within `$NAME`'s own opam file (but global variables take precedence, in particular directory names. Check `opam config list`). Some are only available from the opam file or when the corresponding package is installed.
   - `installed` whether the package is installed
   - `enable` same as `installed`, but takes the value `"enable"` or `"disable"`. To be used in configure scripts
   - `pinned` whether the package is pinned
   - `bin`, `sbin`, `lib`, `man`, `doc`, `share`, `etc`, the installation directories used by the package
   - `name` the name of the package
   - `version` the version of the package (the one corresponding to the current opam file if relevant, the installed one otherwise)
   - `build` the temporary build directory of this package

- `depends` the list of current direct dependencies of this package. Optional dependencies are included if they are currently available
- `hash` the md5 of the opam archive of the package, if available. For cache mechanisms

Boolean variables can also be written as `foo+bar:var` as a shortcut to `foo:var & bar:var`.

3. User-defined variables, as defined in `*.config` files (see §??). This can also be used to override values of the above variables.

As an experimental feature in OPAM 1.2.1 (*i.e.* not to be used on the main package repository), converters of boolean variables to strings can be specified with the following syntax: `"%{var?string-if-true:string-if-fal`
Either string may be empty.

4.3 Files

4.3.1 Substitution files: `*.in`

Any file can be processed using generated using a special mode of `opam` which can perform tests and substitutes variables (see §4 for the exact command to run). Substitution files contains some templates which will be replaced with some contents. The syntax of templates is the following:

- templates such as `%{$NAME:$VAR}%` are replaced by the value of the variable `$VAR` defined at the root of the file `$opam/$SWITCH/config/NAME.config`.
- templates such as `%{$NAME.$LIB:$VAR}%` are replaced by the value of the variable `$VAR` defined in the `$LIB` section in the file `$opam/$SWITCH/config/$NAME.config`

Local and global variables The definitions "`IDENT: BOOL`", "`IDENT: STRING`" and "`IDENT: [ STRING+ ]`", are used to defined variables associated to this package, and are used to substitute variables in template files (see §4.4.1):

- `%{$NAME:$VAR}%` will refer to the variable `$VAR` defined at the root of the configuration file `$opam/$SWITCH/config/N`
- Run `opam config var` for the full list of available variables.

4.4 Commands

4.4.1 Getting Package Configuration

OPAM contains the minimal information to be able to use installed libraries. In order to do so, the end-user (or the packager) should run:

```
$ opam config list
$ opam config var $NAME:$VAR
$ opam config var $NAME.$LIB:$VAR
$ opam config subst $FILENAME+
```

- `var $var` will return the value associated to the variable `$var`
- `subst $FILENAME` replace any occurrence of `%{$NAME:$VAR}%` and `%{$NAME.$LIB:$VAR}%` as specified in §4.3.1 in `$FILENAME.in` to create `$FILENAME`.