# Contents

1 Introduction ........................................ 5  
   1.1 Main Features .................................. 5  
   1.2 Comparing ocp-build to others ............... 5  

2 Tutorial: Building an ocp-build project ............ 7  
   2.1 Invoking ocp-build ............................ 7  
   2.2 The _obuild directory .......................... 8  
   2.3 Setting ocp-build default parameters ......... 8  

3 Tutorial: How to use ocp-build in your projects ... 9  
   3.1 Building a simple program ..................... 9  
   3.2 Building libraries ............................. 10  
   3.3 Customization of per-file options .......... 10  

4 Building OCaml Projects with ocp-build .......... 11  
   4.1 Environment Variables .......................... 11  
   4.2 Configuration Files ............................ 12  
   4.3 Compilation Layout ............................ 12  
   4.4 Format of the package description files (.ocp) 13  
      4.4.1 Description of Simple Packages .......... 13  
      4.4.2 OCaml Configuration ....................... 13  
      4.4.3 OCaml options ............................ 13  
      4.4.4 Advanced options ......................... 15  
   4.5 Command line options ......................... 16  

5 Managing tests with ocp-build ................... 17  
   5.1 Running tests .................................. 17  
   5.2 Adding tests to your packages ............... 18  
      5.2.1 Creating an independant “test” package .... 18  
      5.2.2 Including tests in a “program” package ... 19
5.2.3 Adding external tests to a “program” package . . . . . . 20
5.2.4 Complex examples . . . . . . . . . . . . . . . . . . . 20

6 Managing Syntax Extensions with ocp-build 23

7 Installation 25
  7.1 Installing with opam . . . . . . . . . . . . . . . . . . . . 25
  7.2 Installing from GitHub . . . . . . . . . . . . . . . . . . . 26
Chapter 1

Introduction

ocp-build is a tool to build whole OCaml projects, composed of multiple files in multiple directories.

1.1 Main Features

The main features of ocp-build are:

- Simple declarative project description
- Short syntax for packing modules, dependencies between libraries.
- Simple way to customize options for every module.
- Incremental and parallel build system
- Support for camlp4, composition of syntax extensions and optimised compilation
- Portable, run both on Unices and Windows
- Support and generate ocamlfind META files

1.2 Comparing ocp-build to others

Here is a list of other building tools for OCaml projects and how ocp-build compares to them.
ocamlbuild: ocamlbuild is a generic build manager, where configuration files are written in OCaml. ocp-build has a less generic scope, but consequently, its configuration files are much simpler, and still very powerful because of its specialization for OCaml projects. Also, ocp-build works under native Windows.

make: make is a well-known generic build manager for Unices. Makefiles for OCaml projects are notoriously complicated to write, and do not support parallelization well.

omake: omake is an improved make, written in OCaml, and with native support for OCaml projects. omake works well on big projects and under Windows, but it is not supported anymore, and omake has no support for library dependencies.

ocamlfind ocamlfind is not a build manager, but a command-line tool to make the OCaml compilers aware of other OCaml components installed in the system (libraries, syntaxes, etc.), thanks to their META files. ocp-build supports META files, and, since it reads the environment only once, is much faster than using ocamlfind for every module.

oasis: oasis is a simple frontend to describe OCaml projects, inspired from Cabal. Configuration files are simple declarative files, as with ocp-build, but the build process is done by other tools (ocamlbuild usually). ocp-build is very similar to using oasis+ocamlfind+ocamlbuild, but supports multi-components projects, and its declarative language is simpler.

opam: opam is a source package manager for OCaml. It relies on other build managers to compile packages, and only decides in which order the packages should be compiled and installed to meet dependencies between them. opam uses ocp-build to build itself.
Chapter 2

Tutorial: Building an ocp-build project

Even if you are not using ocp-build to build you own projects, you might need some more information to take advantage of ocp-build features when compiling other projects.

2.1 Invoking ocp-build

If you want to compile a project that uses ocp-build, you can use the following commands:

ocp-build root: This command defines the root directory of the project as the current directory. ocp-build will create a file ocp-build.root. When invoked from any sub-directory, it will come back to this directory to compile the full project.

ocp-build configure [OPTIONS]: This command can be used to modify default options of the project, stored in the ocp-build.root file.

ocp-build project [QUERY]: This command can be used to read the project, and query some informations. For example, it can be used to ask which packages are going to be built in the current configuration.

ocp-build build [PACKAGES]: This command can be used to compile all the project, or just a list of packages.

ocp-build install [PACKAGES]: install the specified packages.
2. The _obuild directory

ocp-build creates a directory _obuild at the root of the project, to store the files it creates. The content of the _obuild directory is as follows:

_obuild/cache.cmd: a list of command checksums, to avoid re-executing commands already executed.

_obuild/cache.cmd.log: the log of texts used to compute command checksums.

_obuild/mutable_tree/: a copy of the project hierarchy of directories, used to store temporary sources and annotations (actually, redirections towards annotation files).

_obuild/package/: for each package, a directory containing the object files generated during the compilation.

2.3 Setting ocp-build default parameters
Chapter 3

Tutorial: How to use ocp-build in your projects

3.1 Building a simple program

To build a program, for example composed of two files a.ml and b.ml, and using libraries x and y (where x and y are META packages), you just need to create a file my-program.ocp in the directory containing the sources, with the following content:

```ocp
begin program "my-program"
  files = [ "a.ml" "b.ml"]
  requires = [ "x" "y" ]
end
```

If you don't know in which order a.ml and b.ml should be linked, you can ask ocp-build to sort them for you:

```ocp
begin program "my-program"
  sort = true
  files = [ "a.ml" "b.ml"]
  requires = [ "x" "y" ]
end
```

Finally, you can provide additional information, not necessary for your program:

```ocp
version = "1.2.3"
```
CHAPTER 3. TUTORIAL: HOW TO USE ocp-build IN YOUR PROJECTS

begin program "my-program"
  files = [ "a.ml" "b.ml"
  requires = [ "x" "y" ]
end

ocp-build can still use this information to generate files that will be integrated to your program, for example here the module Version:

version = "1.2.3"
authors = [ "Jon Doe <jon.doe@does-family.net>" ]
license = [ "GPLv3" ]

begin program "my-program"
  files = [
    "version.ml" (ocp2ml)
    "a.ml" "b.ml"
  ]
  requires = [ "x" "y" ]
end

Once your program compiled, the content of file version.ml can be seen in the directory _obuild_/mutable_tree/SUBDIR/version.ml, where SUBDIR is the subdirectory of your sources in the compiled project.

3.2 Building libraries

3.3 Customization of per-file options
Chapter 4

Building OCaml Projects
with ocp-build

ocp-build can be used to compile simple OCaml projects. The tool uses simple configuration files to describe the packages that need to be compiled, and the dependencies between them.

Compared to other OCaml building tools, it provides the following particularities:

- ocp-build supports complete parallel builds. Its improved understanding of OCaml compilation constraints avoids traditional problems, arising from conflicts while compiling interfaces.

- ocp-build configuration files provide a simple and concise way to handle the complexity of OCaml projects.

- ocp-build supports complex compilation rules, such as per-file options, packing and C stubs files.

- ocp-build can use either a set of attributes or a digest of the content of a file to detect files’ modifications to decide which files should be rebuilt.

4.1 Environment Variables

ocp-build uses the following environment variables:

HOME : the user directory (“.” if not defined)
12  \textit{CHAPTER 4. BUILDING OCAML PROJECTS WITH OCP-BUILD}

\textbf{OCP\_HOME} : ocp-build configuration directory ("\textdollar\{HOME}/.ocp" if not defined)

\textbf{PATH} : the path of directories containing commands (separated by ":" on Unix, ";" on Windows)

\textbf{TERM} : if defined, characters are escaped (also on Windows)

\textbf{OCPBUILD\_VERBOSITY} : verbosity before the -v option is parsed.

\textbf{OCP\_DEBUG\_MODULES} : which modules to debug (need more info...)

\textbf{OCAMLLIB} : not directly used by ocp-build, but used by OCaml, from which ocp-build computes its own configuration.

\subsection*{4.2 Configuration Files}

ocp-build uses two different kind of files to describe a project:

- Each package (or set of packages) should be described in a file with an \texttt{.ocp} extension. When ocp-build is run with the \texttt{-scan} option, it scans the directory to find all such configuration files, and adds them to the project.

- The project should be described in a file \texttt{ocp-build.root}. This file should be at the root of the project, and ocp-build will try to find it by recursively scanning all the parents directories. If it does not exist, it should be created using the \texttt{-init} option.

\subsection*{4.3 Compilation Layout}

ocp-build generates files both in the source directories and in a special \texttt{.obuild} directory, depending on the nature of the files:

- Temporary source files and compilation garbage are stored in the source directories. This set includes implementation and interfaces files generated by \texttt{ocamllex} and \texttt{ocamlyacc}, and other special files such as \texttt{.annot} files.

- Binary object files are stored in the \texttt{.obuild} directory, where a sub-directory is created for each package.
4.4 Format of the package description files (.ocp)

4.4.1 Description of Simple Packages

A simple package description looks like this:

```ocp
begin library "ocplib-system"
  files = [ "file.ml" "process.ml" ]
  requires = [ "unix" ]
end
```

This description explains to ocplib-system that a library ocplib-system should be built from source files file.ml and process.ml (and possibly file.mli and process.mli), and that this library depends on the unix library to be built.

Another simple description is:

```ocp
begin program "file-checker"
  files = [ "checkFiles.ml" "checkMain.ml" ]
  requires = [ "ocplib-system" ]
end
```

This description tells ocplib-system that it should build an executable file-checker from the provided source files, and with a dependency towards ocplib-system. ocplib-system will automatically add the dependency towards unix required by ocplib-system.

4.4.2 OCaml Configuration

The following variables are automatically defined by ocplib-system from OCaml configuration:

- `ocaml_major_version`
- `ocaml_minor_version`
- `ocaml_point_version`

4.4.3 OCaml options

Per-package options

- `dirname(string)` The directory where the package files are located.
generated(bool) If true, the package is already installed

has_byte(bool)

has_asm(bool)

Per-file options

ml(bool) The file is an implementation source (.ml file)

mli(bool) The file is an interface source (.mli file)

cflags(list) Options to be passed to the C compiler

ccopt(list) ???

nopervasives(bool)

nodeps(list) A list of false dependencies

nocmxdeps(list) A list of false dependencies for native code

bytelink(list)

bytecomp(list)

asmlink(list)

asmcomp(list)

dep(list)

rule_sources(list)

pp(list) ???

pp_requires(list) ???

sort(bool)
4.4. Advanced options

Per-file options

Options can be specified on a per-file basis:

```ocp
begin library "ocplib-fast"
  files = [
    "fastHashtbl.ml" (asmcomp = ["-inline"; "30"])
    "fastString.ml"
  ]
end
```

They can also be specified for a group of files:

```ocp
begin library "ocplib-fast"
  files = [
    begin (asmcomp = ["-inline"; "30"])
    "fastHashtbl.ml"
    "fastMap.ml"
    end
    "fastString.ml"
  ]
end
```

Configurations

Preprocessor requirements: pp Requires

The **pp Requires** option can be used to declare a dependency between one or more source files and a preprocessor that should thus be built before. The preprocessor must be specified as a program package in a projet, plus the target (bytecode byte or native asm):

```ocp
begin library "ocplib-doc"
  files = [
    "docHtml.ml" (  
      pp = ["./_obuild/ocp-pp/ocp-pp.byte"]  
      pp_requires = ["ocp-pp:byte"]  
    )
    "docInfo.ml"
  ]
end
```
requires = ["ocp-pp"]
end

Note that you still need:

- To specify the package in the requires directive, to ensure that this package will be available when your package will need it for processing.

- To specify the command pp to call the preprocessor

4.5 Command line options
Chapter 5

Managing tests with ocp-build

ocp-build can run tests on packages.

5.1 Running tests

Running tests is simple:

ocp-build -tests

ocp-build will compile all the tests, and run them. ocp-build will then display how many tests failed and succeeded, which tests failed, and timings for benchmarks:

Parallel tests ran in 0.00s (max 6 jobs)
Serial tests ran in 0.08s
FAILED: 0/7
SUCCESS: 7/7
  0.08s ocp-build.test/tests/cycle

ocp-build runs tests in two phases: in the first parallel phase, it runs as many tests as possible in parallel, depending on the “njobs” option; in the second sequential phase, it runs the tests in sequential. Tests are moved to the sequential phase when they are benchmarks or serialized.
5.2 Adding tests to your packages

5.2.1 Creating an independant “test” package

A “test” package is a program that is only compiled when ocp-build is ran with argument -tests.

```ocp
begin test "my-lib-test"
  files = [ "test.ml" ]
  requires = [ "my-lib" ]
end
```

Within a “test” package, it is possible to run several times the program with different parameters. For that, a “tests” field can define a list of test names. If the “tests” field of a “test” package is not specified, a test name “default” is automatically defined.

Every test can define a set of options:

- **test_exit** : the correct exit status for the test (0 by default)
- **test_dir** : the directory in which the test program should be run (default is empty, for the project root)
- **test_args** : a list of arguments for the test program (default is empty)
- **test_benchmark** : true if the time spent running the test should be displayed. Benchmarks are not run in parallel with other tests (default is false).
- **test_stdout** : the name of a file that should be used, when the exit status is correct, to compare with what the test printed on stdout.
- **test_stderr** : the name of a file that should be used, when the exit status is correct, to compare with what the test printed on stderr.
- **test_stdin** : the name of a file that should be passed on the standard input of the test.
- **test_serialized** : true if the test should not be run in parallel with other tests. Useful for tests that are themselves using several cores (default is false).
- **test_variants** : a list of strings on which a test should iterate (default is "").
5.2. ADDING TESTS TO YOUR PACKAGES

**test_asm**: true if the native version of the test should be run (default is true)

**test_byte**: true if the bytecode version of the test should be run (default is true).

**test_cmd**: the name of the file that should be run (default is %{binary}%, see substitutions later).

When defining these options, substitutions are available:

- %{test}% the name of the current test.
- %{binary}% the name of the current executable being run
- %{sources}% the current source directory of the test
- %{tests}% a sub-directory “tests” within the source directory of the test
- %{variant}% the current variant, when “test_variants” was specified.
- %{results}% the directory where test results will be stored (can be used to store other files).

### 5.2.2 Including tests in a “program” package

It is also possible to define tests directly in a “program” package. For that, you should need to define the “tests” field:

```
begin program "my-program"
    files = [ "main.ml" ]
    requires = [ "my-lib" ]

    (* for all the tests, the program should receive the test name as first argument *)
    test_args = [ "%{test}%"]

    (* we want to run 3 tests, with names "1", "2" and "3" *)
    tests = [ "1" "2"
             (* in the test "3", the expected exit status should be 2 *)
             "3" (test_exit = 2) ]
end
```
5.2.3 Adding external tests to a “program” package

It is also possible to define tests for a “program” package in a separate “test” package. For that, the “test” package should have an empty “files” field, and have the “program” package in its “requires” field. The previous tests would now be written:

```
begin test "my-program-test"
    files = []
    requires = [ "my-program"
    (* for all the tests, the program should receive the test name
        as first argument *)
    test_args = [ "%{test}"
    (* we want to run 3 tests, with names "1", "2" and "3" *)
    tests = [ "1" "2"
        (* in the test "3", the expected exit status should be 2 *)
        "3" (test_exit = 2) ]
end
```

5.2.4 Complex examples

In the following example, we generate tests for the program “compile-and-run”: we define 4 tests (the “tests” field), and for each test, we will run the 4 variants (the “test_variants” option). In each run, the variant value is passed as the first argument, while the second argument is a file within the test directory. The program “compile-and-run” is only tested in native code (we set “test_byte” to false). For each test, we compare the output with a file in the test directory with the “.reference” extension (the “test_stdout” option):

```
begin test "basic"
    files = []
    requires = [ "compile-and-run"
    test_byte = false
    test_stdout = [ "%{sources}/%{test}.reference" ]
    test_variants = [
        "-ocamlc" "-ocamlopt"
        "-ocamlc.opt" "-ocamlopt.opt"
```
test_args = [ "%{variant}" "%{sources}/%{test}.ml" ]
tests = [
  "arrays"  "equality"  "maps"  "sets"
]
end
Chapter 6

Managing Syntax Extensions with ocp-build
Chapter 7

Installation

ocp-build is part of TypeRex. The simplest way to install it is to use opam\(^1\), the source package manager for OCaml. If for some reasons, you are not satisfied by this way, you will want to try to install it from its source repository, on GitHub.

7.1 Installing with opam

ocp-build is available in opam. It is a meta-package (an empty package) that triggers the installation of TypeRex, with version greater than 1.99. Indeed, ocp-build is compiled and installed by the TypeRex package. Any previous version of ocp-build (especially version 0.1) should be uninstalled before installing TypeRex.

First, let's check if ocp-build is already installed:

```
peerocaml:~% opam info ocp-build
package: ocp-build
installed-version: ocp-build.0.1 [4.00.1]
available-version: 1.99.2-beta
description: Project manager for OCaml
```

The output of the command shows that ocp-build is already installed, with version 0.1. We should remove it immediately:

```
peerocaml:~% opam remove ocp-build
The following actions will be performed:
- remove ocp-build.0.1
```

\(^1\)http://opam.ocamlpro.com
CHAPTER 7. INSTALLATION

0 to install | 0 to reinstall | 0 to upgrade | 0 to downgrade | 1 to remove

Note that some other packages depending on ocp-build can need to be uninstalled too. You can keep a list of these packages, so that you can install them again after installing the new version.

If you only ask opam to install ocp-build, opam might decide to reinstall ocp-build 0.1 because it has a shorter chain of dependencies than ocp-build 1.99. To force it to install the new version, we can ask for both ocp-build and TypeRex:

```
peerocaml:% opam install ocp-build typerex
```

The following actions will be performed:
- install ocp-build.1.99.2-beta
- install typerex.1.99.2-beta

Do you want to continue ? [Y/n]

7.2 Installing from GitHub

ocp-build sources can be retrieved from GitHub. The latest version is developed in the typerex2 branch of the OCamlPro/typerex repository:

```
peerocaml:% git clone git@github.com:OCamlPro/typerex.git
peerocaml:% git checkout typerex2
```

In the source directory (typerex), We can now configure, compile and install:

```
peerocaml:% ./configure --prefix /usr/local/
peerocaml:% make
peerocaml:% make install
```

The last command will install all TypeRex commands and libraries. If you just want to install ocp-build, you can use:

```
peerocaml:% sudo ./_obuild/ocp-build/ocp-build.asm -install ocp-build \ 
     -install-bin /usr/local/bin -install-lib /usr/local/lib/ocaml
```

Note that we used sudo since the install paths we specified require administrator privileges.

It is also possible to uninstall files installed by make install using ocp-build:
peerocaml:~% ocp-build -uninstall typerex

We can also use ocp-build to uninstall packages installed by ocp-build (but it would be a bad idea to use that to uninstall packages installed by opam):

peerocaml:~% sudo ocp-build -uninstall ocp-build

If you want to modify ocp-build, sources specific to ocp-build are located in the tools/ocp-build directory.