ocamlbuild

a compilation manager for OCaml projects

Berke Durak    Nicolas Pouillard

Berke.Durak@inria.fr    Nicolas.Pouillard@inria.fr

January 26, 2008
Outline

1. Introduction
2. Regular OCaml projects
3. Dealing with exceptions to standard rules
4. Writing an ocamlbuild plugin
5. General features
6. Conclusion
Why such a tool?

- To make our OCaml life easier
Why such a tool?

- To make our OCaml life easier
- To stop writing poor Makefiles
Why such a tool?

- To make our OCaml life easier
- To stop writing poor Makefiles
- To have a tool that Just works™
What does ocamlbuild handle?

Regular OCaml projects of arbitrary size
Trivially handled using the command line options.
What does *ocamlbuild* handle?

**Regular OCaml projects of arbitrary size**
Trivially handled using the command line options.

**Mostly regular OCaml projects with common exceptions**
Requires writing one tag file (_tags_) that declares those exceptions.
What does ocamlbuild handle?

Regular OCaml projects of arbitrary size
Trivially handled using the command line options.

Mostly regular OCaml projects with common exceptions
Requires writing one tag file (_tags) that declares those exceptions.

Almost any project
Accomplished by writing an ocamlbuild plugin.
What does ocamlbuild provide?

- Automated whole-project compilation
- Minimal recompilation
- Lots of useful targets (doc, debugging, profiling...)
- Supports multiple build directories
- Automatic and safe cleaning
- A source directory uncluttered by object files
- A portable tool shipped with OCaml
What does ocamlbuild provide?

- Automated whole-project compilation
- Minimal recompilation
- Lots of useful targets (doc, debugging, profiling...)
- Supports multiple build directories
- Automatic and safe cleaning
- A source directory uncluttered by object files
- A portable tool shipped with OCaml
- Saves time and money!
Outline

1. Introduction
2. Regular OCaml projects
3. Dealing with exceptions to standard rules
4. Writing an ocamlbuild plugin
5. General features
6. Conclusion
What’s a regular OCaml project?

It’s a project that needs no exceptions from the standard rules:

- Has compilation units (ml and mli files)
- May have parsers and lexers (mly and mll files)
- May use packages, libraries and toplevels (ml{pack,lib,top})
- May link with external libraries
- Has one main OCaml unit from which these units are reachable
How difficult is it to build regular projects by hand?

**OCaml has subtle compilation rules**

- Interfaces (.mli) can be absent, yet buildable (.mly)
- Native and bytecode suffixes and settings differ
- Native packages are difficult to do (-for-pack)
- Linkage order must be correctly computed
- Include directories must be ordered
- `ocamldep` gives partial information (too conservative)
How does ocamlbuild manage all that?

It has a lot of hand-crafted Ocaml-specific compilation logic!
How does ocamlbuild manage all that?

It has a lot of hand-crafted Ocaml-specific compilation logic!

A dynamic exploration approach

- Start from the given targets
- Attempt to discover dependencies using ocamldep
- ocamldep cannot always be trusted: backtrack if necessary
- Launch compilations and discover more dependencies
Outline

1 Introduction

2 Regular OCaml projects

3 Dealing with exceptions to standard rules

4 Writing an ocamlbuild plugin

5 General features

6 Conclusion
What’s an exception?

Files that need specific flags

- Warnings to be enabled or disabled
- Debugging (-g), profiling (-p), type annotation, recursive types, -linkall, -thread, -custom...

- Units that need external C libraries
- Binaries that need external OCaml libraries
- Directories that must be included or excluded
- Dependencies that cannot be discovered
Make and exceptions

- The `make` tool can’t handle exceptions very well.
- Needs exceptions to be encoded as specific rules.
- This generally makes rules and exceptions tightly bound by variables.
- This creates non-modular makefiles that don’t **scale**.
The tags, our way to specify exceptions

- Tagging is made in _tags files
- Each line is made of a pattern and a list of signed tags
- A line adds or removes tags from matching files
- Patterns are boolean combinations of shell-like globbing expressions

"funny.ml": rectypes
<**/*.ml*>: warn_A, warn_error_A, debug, dtypes
"foo.ml" or "bar.ml": warn_v, warn_error_v
"vendor.ml": -warn_A, -warn_error_A
<main.{byte,native}>: use_unix
"main.byte": use_dynlink, linkall
"test": not_hygienic
<satsolver.cm[io]>: precious
How tags and rules give commands

Files are tagged using tagging rules

"foo/bar.ml": rectypes

Rules then produce commands with **tagged holes**

```
let tagged_hole =
tag_for(ml)++"ocaml"++"compile"++"byte" in
Cmd(S[A"ocamlc";A"-c";T tagged_hole;P ml;A"-o";P cmo])
```

These holes are filled by command fragments (such as flags)

```
flag ["ocaml"; "compile"; "byte"; "rectypes"]
(A"-rectypes")
```
Tags and dependencies

One can define dependencies triggered by combinations of tags

dep ["ocaml"; "link"; "byte"; "program"; "plugin:foo"]
["plugin/pluginlib.cma"; "plugin/plugin_foo.cmo"]

By tagging files we make things happen

"test.byte": plugin:foo
Outline

1. Introduction
2. Regular OCaml projects
3. Dealing with exceptions to standard rules
4. Writing an ocamlbuild plugin
5. General features
6. Conclusion
Not a specific language, but plain OCaml code

- Plugins are compiled on the fly
- Dynamic configuration is feasible

With a plugin one can:

- Extend rules (add new ones, override old ones)
- Add flags and dependencies based on tags
- Tag files
- Change options
- Define the directory structure precisely
- Help ocamldep
- Specify external libraries
Outline

1. Introduction
2. Regular OCaml projects
3. Dealing with exceptions to standard rules
4. Writing an ocamlbuild plugin
5. General features
6. Conclusion
Parallel execution where applicable

- You select the maximum number of jobs (-j N)
- Rules know how to ask for parallel targets
- The system keeps things scheduled correctly
- Example: Separate compilation of byte code
- (Optimal scheduling would require a static graph)
Some supported tools

**Menhir as an ocamlyacc replacement**

- Enabled with the `use_menhir` global tag or the `-use-menhir` option
- Handles implicit dependencies using `-infer`
Some supported tools

**Menhir** as an *ocamlyacc* replacement
- Enabled with the `use_menhir` global tag or the `-use-menhir` option
- Handles implicit dependencies using `-infer`

**OcamlDoc** to build your doc
- Separated construction using `(-dump/-load)`
- Handles HTML, LaTEX, MAN, DOT, TeXi
Some supported tools

**Menhir** as an **ocamlyacc** replacement

- Enabled with the `use_menhir` global tag or the `-use-menhir` option
- Handles implicit dependencies using `--infer`

**Ocamldoc** to build your doc

- Separated construction using `(-dump/-load)`
- Handles HTML, LaTeX, MAN, DOT, TeXI

**Camlp4** aware

- Tags allow to setup any installed Camlp4 preprocessor
- Fine grained dependencies help a lot...
Outline

1. Introduction
2. Regular OCaml projects
3. Dealing with exceptions to standard rules
4. Writing an ocamlbuild plugin
5. General features
6. Conclusion
ocamlbuild can be used in three ways:

- With only command-line options for fully regular projects
- With the _tags file for intermediate projects
- With a plugin for the most complex projects
**ocamlbuild** can be used in three ways:

- With only command-line options for fully regular projects
- With the `_tags` file for intermediate projects
- With a plugin for the most complex projects

**ocamlbuild** saves your time by:

- Building your project gently
- Compiling only as necessary
- Running commands in parallel
- Keeping your house clean
- Letting you concentrate on your code!