ocp-lint
A Plugin-based Style-Checker with Semantic Patches

Çağdaş Bozman
OCamlPro
cagdas.bozman@ocamlpro.com

Théophane Hufschmitt
OCamlPro
theophane.hufschmitt@ocamlpro.com

Michael Laporte
OCamlPro
michael.laporte@ocamlpro.com

Fabrice Le Fessant
INRIA & OCamlPro
fabrice.le_fessant@inria.fr

Abstract

In this talk, we will present ocp-lint, a new style-checker for OCaml projects. ocp-lint can typically be used to check pull-requests in a Github-style workflow. ocp-lint is highly extensible, with a simple API to define new plugins that can be linked dynamically. ocp-lint is easily configurable, with plugin- and analysis- specific options and arguments. ocp-lint can also use semantic patches, a patch-style format to describe code patterns to detect. Warnings found by ocp-lint are stored in a database, to avoid useless recomputations and ease the development of external tools and GUIs to exploit its results.

1 Introduction

Even in languages with strong static typing, style-checkers can be very useful: some coding styles are known to ease the hidden presence of bugs; other coding styles can be inefficient or raise memory issues. Also, having different coding-styles in a project can make code-review more difficult, disturbing the focus of reviewers towards minor style issues instead of looking for algorithmic issues.

A style-checker can solve many of these problems by providing an automatic way to check for coding styles that should be avoided in a project. For that, the style-checker should be fast, to provide feedback almost immediately, configurable, to allow project leaders to express their preferences, and extensible, to allow project developers to design new analysis specific to their needs.

In this talk, we will present ocp-lint, a style-checker for OCaml projects, that aims at satisfying all these needs. An overview of ocp-lint is given in section 2. Of course, ocp-lint builds on our experience learnt from using other style-checkers for OCaml, as shown in section 3. It is easy to use and configure, as depicted in appendix A, and can be extended using both *semantic patches*, in appendix B, and dynamic plugins using a simple API, described in appendix C.

2 Overview

The main design idea behind ocp-lint is to provide a framework for checking OCaml projects for coding errors, instead of just another monolithic tool. For that, we have tried to make it easy to extend ocp-lint, either using *semantic patches* (patterns of code described in a patch-like syntax on OCaml code) or using dynamic plugins (user code linked at runtime). We also tried to make it as configurable as possible: each plugin and each analysis can be enabled or disabled by a simple project configuration file, and analysis settings can be modified in the same configuration file. We wrote a set of plugins and analyses, both to be able to use the tool for our own purposes, and to provide examples for developers of how to extend the tool.

Configuration files in ocp-lint are managed by the `ocplib-config` [1] library. This library provides simple functions to define options, that can be manipulated as simply as references in the program, while being loaded and saved automatically in configu-

ration files. The library also automatically generates command-line arguments to modify the options. ocp-lint can use both user- and directory- specific configuration files.

We already implemented a small set of plugins working on different kinds of inputs: text plugin (length of lines, extra spaces, non-ANSII characters, etc.), indent plugin (correct use of ocp-indent), tokens plugin (non-ASCII characters in comments, in idents, etc.), parsetree plugin (constructor with tuple arguments, local aliases, identifiers lengths, semantic patches, etc.), sempatch plugin (detection of patterns provided by semantic patches), typedtree plugin (non-qualified external idents), parsing plugin (extra-parentheses, use of parentheses instead of begin..end, etc.).

Warnings emitted by these analyses are stored in a project-database. The database is then used when the tool is restarted, to avoid checking files again if they have not been modified, and to allow other tools to display the results (ocp-index or Merlin for example, or on a web interface).

3 Related Work

ocp-lint is not the only tool that can be used to improve the quality of the code of an OCaml project. In this section, we compare ocp-lint with three other tools that can be used for this purpose.

Mascot [2] was probably the most exhaustive style-checker for OCaml. It provided many checks in various categories: code, documentation, interface, metrics, and typography. However, it is not maintained anymore, and hard to extend, especially as analyses are heavily based on using Camlp4 syntax trees.

`ocamllint` [3] is a style-checker that runs as ppx [4] while compiling the project. Thus, it requires minimum effort to be used on an OCaml project. However, the number of analyses is currently very limited, and they can only be applied on the AST, whereas ocp-lint can work also on text files, and on typedtrees.

Dead code analyzer [5] tries to detect useless patterns in an OCaml project. For example, it detects never used values, types fields and constructors (that can thus be removed as dead code), and optional labels either always or never used. The tool assumes that interface files (`.mli`) are compiled with the `-keep-locs` and source files (`.ml`) with `-bin-annot`. The analysis can be quite expensive, but the tool is a good complement to ocp-lint, and could be added as a plugin to benefit from its database and project management.

4 Diffusion

ocp-lint is being developed on Github, as part of Typerex linting tools, and distributed under GPLv3, with several plugins and many useful checks:
`http://github.com/OCamlPro/typerex-lint`

References

[1] OCamlPro. ocplib-config. `https://www.typerex.org/ocp-build.html`.

[2] Xavier Clerc. Mascot. `http://mascot.x9c.fr/`, 2010–2012.

[3] Cryptosense. ocamllint. `https://github.com/cryptosense/ocamllint`, 2015–2016.

[4] Yaron Minsky. "extension points, or how ocaml is becoming more like lisp". `https://blogs.janestreet.com/extension-points-or-how-ocaml-is-becoming-more-lik` 2008.

[5] LexiFi. Ocaml dead code analyzer. `https://github.com/LexiFi/deadcodeanalyzer`, 2015–2016.

[6] Laboratoire d'Informatique de Paris 6 Julia Lawall. coccinelle. `http://coccinelle.lip6.fr/`, 2009–2016.

A Usage

ocp-lint can use a database to make style-checking a project. To create an initial database, the user should call `ocp-lint --init`, otherwise, no database will be used.

Then, here is a typical example of running ocp-lint:

```
$ ocp-lint --path tools/ocp-lint
       --disable-plugin-typedtree
Summary:
* 11 files were linted
* 40 warnings were emitted:
 * 2 "interface_missing" number 1
 * 2 "code_length" number 1
 * 4 "ocp_indent" number 1
File "lint_input.ml", line 1:
  "ocp_indent" number 1:
  File lint_input.ml' is not
     indented correctly.
File "lint_actions.ml", line 1:
  "code_length" number 1:
  This line is too long ('82'): it
      should be at most of size
     '80'.
File "main.ml", line 1:
  "interface_missing" number 1:
  Missing interface for main.ml'.
```

The `--path` argument tells ocp-lint the directory to scan for files to check. ocp-lint will check all the files with extensions .ml, .mli, .cmt and .cmti. We are currently parallelizing the process of checking each file in a different process: each internal process will store its results in the (temporary or persistent) database and the calling process will display the results at the end.

Each plugin can define its own arguments that are added to ocp-lint command-line, to:

- enable/disable each plugin (e.g. `--disable-PLUGIN`);
- enable/disable each plugin analysis (e.g. `--disable-PLUGIN.LINTER`);
- enable/disable each linter set of warnings (e.g. `--disable-PLUGIN.LINTER.warnings -A+3..5+8`);
- set analysis settings a set of parameters (e.g. `--PLUGIN.LINTER.OPTION VALUE`);

The `--load-plugins FILE.cmxs` argument can be used to load a plugin dynamically (ocp-lint comes with a few plugins already statically linked), and the –list argument can be used to list plugins and their analyses.

Options changed on the command-line can be saved in the project configuration file using the `--save-config` argument.

Several output formats can be used to print warnings: plain-text, JSON, HTML, etc. This is easily extensible, and we plan to provide more formats.

B Using Semantic Patches

The `ocplib-sempatch` library was inspired the Semantic Patches from by Coccinelle [6]. It provides a simple text DSL to describe patterns on OCaml syntax in a patch-like style, and these patterns can be used to locate these patterns in source files. For that, it uses a regular expression engine operating on the OCaml AST — the parsing AST or, when needed and possible, the typed AST.

In ocp-lint, we use this library to allow the user to provide its own OCaml patterns to recognize. For example, the following patch will make ocp-lint raise a warning, proposing to replace a if-then-else constructs with identical expressions in both branches by a simple sequence, ignoring the result of the condition:

```
@ConstantIf
expressions : cond, e1, e2
when: ''e1 = e2''
'''
- if cond then e1 else e2
+ ignore (cond:bool); e1
'''
```

We were able to use these semantic patches to express most of the patterns recognized by ocamllint.

C Extensibility

It is difficult to forecast all the checks that users will want to apply on their code. We implemented a large set of checks, that can be configured on a user basis, or per project, and semantic patches can be used to add more patterns to detect. Nevertheless, we thought it would be useful to allow users to define their own complex checks, and for that, we provided a simple way to develop plugins and link them at runtime.

We sketch here the development of a simple plugin. First, we create a `Plugin` to which analyses will be attached:

```
module Plugin = Lint_plugin_api.
    MakePlugin (struct
  let name = "Text Plugin"
  let short_name = "plugin_text"
  let details = "A plugin..."
end)
```

Then, we attach a first analysis to this plugin:

```
module Linter = Plugin.MakeLint(
    struct
  let name = "Detect use of.."
  let version = 1
  let short_name = "not_that_char"
  let details = "Detect ..."
end)
```

We can now define and attach the warnings that will be raised. For brievity, we just display one definition:

```
let w_non_ascii_char =
  Linter.new_warning ~id:1
  ~short_name:"non_ascii_char"
  ~msg:"Non-ASCII char $char used"
```

In the code, we usually prefer the use of symbolic warnings, so we define them and provide a translation function:

```
type warning =
  | NonAsciiChar of string
  | ..
module Warnings =
 Linter.MakeWarnings(struct
  type t = warning
  let to_warning = function
  | NonAsciiChar s ->
     w_non_ascii_char, ["char", s]
  | ..
  end)
```

We can now define our analysis. We can use several reporting functions defined by the MakeWarnings functor:

```
let check_line lnum line file =
  ..
  Warnings.report_file_line_col
     file lnum i
     (NonAsciiChar (Printf.sprintf
         "\\%03d" c))
  ..
let check_file file =
  FileString.iteri_lines (fun lnum
      line ->
      check_line lnum line file)
         file
```

Finally, we declare that the analysis should be carried out on source files:

```
module MainSRC = Linter.
   MakeInputML(struct
  let main source = check_file
     source
end)
```

We have predefined several kinds of inputs for analysis:

- All files: the input is the list of files to check
- Source file: the input is the name of a source file to check
- Tokens: the input is the list of tokens from the OCaml lexer
- Parsetree: the input is the standard AST
- Typedtree: the input is the typed AST, from the `.cmt` file

Currently, we have created one plugin per input, that gathers together all the analyses on that input. However, we expect users to define their own plugins, mixing analyses on the different inputs, as better suited for their checks.

Finally, we are now working on *custom inputs*, i.e. an input that can be defined by the user, and then used by multiple analyses. A typical example of use is the ocp-lint-plugin-parsing plugin, that includes a full OCaml 4.03 parser, that we have modified to generate an AST closer to the real input. Currently, this plugin is defined as one analysis, taking a source and applying many checks on the AST (for example, to warn for extra parenthesis). However, we would like to define the analysis separately, while still parsing only once the file with the new parser, defined as a user custom input.

D Conclusion

When designing ocp-lint, our goal was to use it on our own Github projects, to check pull-requests both from the project developers and from external contributors. We plan to apply it soon to all our projects, once most of the analyses we need are implemented.

Although we currently use a non-optimized approach in the implementation of the analyses (different checks are often done in different analyses, while they could be done in the same iteration on the AST), performance is good enough for its purpose. For example, running *sequentially* all the current 30 analysis of our 6 plugins, we get the following performances:

Project	Files	LOC	Warnings	Time
ocp-index	12	4333	36	0.14s
ocp-indent	12	5763	44	0.32s
stdlib	35	12957	80	2.18s
opam	64	26906	362	3.09s
flow	119	47833	563	13.25s
hack	386	73715	1213	33.57s

We also took internationnalization into account in the design: the message associated with each warning is a simple string, that will be customized in the future for different languages.

The project sources are hosted on Github, and an OPAM package should be available soon.