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

Çağdaş Bozman[1], Théophane Hufschmitt[1], Michael Laporte[1] and Fabrice Le Fessant[1,2]

[1]OCamlPro
[2]INRIA

September 17, 2016
THIS DOCUMENT IS A DRAFT

Contents

1 Overview

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.

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 configuration 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).

In this document, 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 1. Of course, ocp-lint builds on our experience learnt from using other style-checkers for OCaml, as shown in section 9. It is easy to use and configure, as depicted in appendix 4, and can be extended using both *semantic patches*, in appendix 7, and dynamic plugins using a simple API, described in appendix 8.

2 Use Cases

2.1 Using Occasionally

Without the database

To start to lint your project without the database feature, just run -lint which will output the warnings in your terminal by default:

```
$ cd $PROJECT
$ ocp-lint
```

Without the database feature, nothing is cached, so every time you will run -lint, it will recompute and re-lint every file on your project, even when it is no needed.

With the cache system (database)

First go to your project directory and initialize the linter which will create a default configuration file and an `_olint` directory to store the the results for the cache system.

```
$ cd $PROJECT
$ ocp-lint --init # create _olint directory and .ocplint configuration file
```

Then, edit your .ocplint file or just run the default behavior of the linter.

```
$ ocp-lint # default directory is .
$ ocp-lint --path src/
```

You will get some entries in your database showing the different warning of your project.

```
$ ocp-lint --path $SOURCEDIR --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'.
```

After that, you can edit your source files to fix the issues raised by the linter.

To force the linter to lint a file, even when it is no needed, we can use the `--no-db-cache` option in the command line. The linter will force the lint of your files and will not cache the result in the database.

You can run ocp-lint as often as you want, it will update your database only if you change your current configuration or if a file has changed.

2.2 Using During the Development

The initializing process is the same as explain in the previous section.

ocp-lint can be integrated to your editor as `ocp-index` or `merlin`. You can then have the linter output directly in your editor and then go to the warning line directly. With this feature, every warning could be fixed more quickly and during the development process.

Some hook can also be add to start the linter after each compilation of your project or even each time you save a file. In that case, only this file can be linted to see the output in your editor, it will be more easy to fix the warnings.

2.3 Using in a Git-development process

Another use of ocp-lint is to integrate it during the development process. For example, we can add some hook with `git` which will check the project each time a git command will be executed.

As a Pre-commit Hook

One of these hooks is to check the project before committing. To do so, we can add a *pre-commit hook* which will start ocp-lint before each commit. We already propose the script below that you can add to your git configuration:

```
#!/bin/sh

LINT=ocp-lint

$LINT --warn-error tools > /dev/null

if [ "$?" = 0 ]; then
    exit 0
else
    echo "\n/!\\ Please fix the warnings before commiting./!\\"
    exit 1
fi
```

You can copy this script in your `$PROJECT/.git/hook` directory to enable the pre-commit hook. The linter will warn you about your code before each commit.

As a Pre-push Hook

In a similar way, you can add a pre-push hook which will warn you about the warnings in your code before every `git push` command. Until every warnings are fixed, there will be any chance to push the code to your git repository.

2.4 Using in a CI Process

Using ocp-lint with a CI tool like travis, you can add the hook to start the linter after the compilation process. Then, the CI tool can automatically add comments to your github pull request with all the warnings or directly add comments to the corresponding lines. Developers can then discuss about the warnings and decided the needed to fix them or not to accept the pull request.

ocp-lint can be more strict with the `-warn-error` option. In that case, the linter will return an exit code different from 0 and the CI tool will fail until all warnings are fixed.

3 Configuration, Build, Installation and Testsuite

3.1 Dependencies

There are a few dependencies for ocp-lint: menhir, ocp-indent and ocp-build. You can install them via OPAM, the OCaml PAckage Manager:

```
$ opam install menhir ocp-build ocp-indent
```

3.2 Configuration

First, we need to configure `typerex-lint`. To do so, we use `ocp-autoconf` [2], a simple tool to manage standard project files for OCaml projects. It can manage:

- *./configure*: basic configure.ac files (autoconf) to detect OCaml and its libraries, and set variables to be used in Makefiles and 'ocp-build' files.

- *opam*: generate a standard opam file for your project, and a script to upload new versions of your project to Github
- *Travis files*: standard Travis files to test pull-requests on Github

Basic Usage `ocp-autoconf` will create a 'configure' file and a 'autoconf/' directory at the root of your project, so you should make sure such files do not already exist. The 'configure' file is just a script to call the real 'configure' script inside the 'autoconf/' directory.

```
$ ocp-autoconf --save-template
```

This will create two configuration files for `ocp-autoconf` :

- *ocp-autoconf.config*: a file containing options to describe your project
- *ocp-autoconf.ac*: a file that you can use to insert 'autoconf' instructions inside the 'autoconf/configure.ac' file that will be generated.

Then we can start the configure script:

```
$ ./configure
```

3.3 Build

`ocp-build` is a build system for OCaml application, based on simple descriptions of packages. `ocp-build` combines the descriptions of packages, and optimize the parallel compilation of files depending on the number of cores and the automatically-infered dependencies between source files.

For building the project you can either use directly `ocp-build` or the Makefile which will just call `ocp-build` :

```
$ make
```

3.4 Installation

You can install ocp-lint via OPAM:

```
$ opam install ocp-lint
```

Or pin the development package:

```
$ opam pin add ocp-lint $OCPLINTSOURCES
$ opam install ocp-lint
```

You can install ocp-lint directly from the sources:

```
$ make install
```

3.5 Testsuite

Each plugin can define a set of tests. To make all the test easily we use `ocp-build` which will find the .ocp file containing the tests. To define a set of test, create a directory in a plugin source:

```
$ mkdir $OCPLINTPATH/plugins/MYPLUGIN/tests
```

Then create a .ocp file to describe the tests:

```
begin test "test-ocp-lint-MYPLUGIN"
  test_byte = false
  requires = [ "ocp-lint-testsuite" ]
```

```
  test_args = [
    "%{ocp-lint_FULL_DST_DIR}%/ocp-lint.asm" "%{sources}%"
  ]
end
```

TODO cago: michael, il faudrait que tu dÃ©crives le process complet pour ajouter un test avec les histoires de .result

Finally you can use ocp-build to run the tests anywhere in the source tree:

```
$ ocp-build tests
```

Or in the source directory use the Makefile rule:

```
$ make test
```

4 Usage

4.1 Conventions

For each plugins, 2 options are generated automatically:

- `--enable-PLUGIN` which allow to enable the plugin `PLUGIN` (e.g. `--enable-plugin-typedtree`);
- `--disable-PLUGIN` which allow to disable the plugin `PLUGIN` (e.g. `--disable-plugin-typedtree`);

For each linters, 3 options are generated automatically:

- `--enable-PLUGIN-LINTER` which allow to enable the linter `LINTER` (e.g. `--enable-plugin-parsetree.code-identifier-length`);
- `--disable-PLUGIN-LINTER` which allow to disable the linter `LINTER` (e.g. `--disable-plugin-parsetree.code-identifier-length`);
- `--PLUGIN-LINTER-warnings` which allow to choose warnings raised by the linter `LINTER` (e.g. `--plugin-text.code-length.warnings -A+3..5+8`);

In addition to that, the plugin developers can add specific option to a plugin or a linter: `--PLUGIN-OPTION` or `--PLUGIN-LINTER-OPTION`
(e.g. `--plugin-parsetree.code-identifier-length.max-identifier-length 2`).

4.2 Kernel Arguments

```
  --init                  Init a project
  --path DIR              Give a project dir path
  --output-txt FILE       Output results in a text file.
  --list                  List of every plugins and warnings.
  --warn-error            Every warning returns an error status code.
  --load-plugins PLUGINS  Load dynamically plugins with their
                          corresponding 'cmxs' files.
  --save-config           Save ocp-lint default config file.
  --no-db-cache           Ignore the DB feature.
  --print-only-new        Print only new warnings.
```

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.

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.

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.

4.3 Plugins Specific Arguments

Plugin On File System

This plugin contains linters on the file system. For example, we can check if an interface file is missing, or do some checks on files name.

```
--disable-plugin-file-system
                          A plugin with linters on file system like
                          interface missing, etc.
--enable-plugin-file-system
                          A plugin with linters on file system like
                          interface missing, etc.

--disable-plugin-file-system.interface-missing
                          Enable/Disable linter "Missing interface".
--enable-plugin-file-system.interface-missing
                          Enable/Disable linter "Missing interface".
--plugin-file-system.interface-missing.warnings <value>
                          Enable/Disable warnings from
                          "Missing interface" (current: +A)

--disable-plugin-file-system.project-files
                          Enable/Disable linter "File Names".
--enable-plugin-file-system.project-files
                          Enable/Disable linter "File Names".
--plugin-file-system.project-files.warnings <value>
                          Enable/Disable warnings from "File Names"
                          (current: +A)
```

Plugin On Text

This plugins contains linters on the source.

```
--disable-plugin-text     A plugin with linters on the source.
--enable-plugin-text      A plugin with linters on the source.

--disable-plugin-text.code-length
                          Enable/Disable linter "Code Length".
--enable-plugin-text.code-length
                          Enable/Disable linter "Code Length".
--plugin-text.code-length.max-line-length <int>
                          Maximum line length
--plugin-text.code-length.warnings <value>
```

```
                          Enable/Disable warnings from "Code Length"
                          (current: +A)

  --disable-plugin-text.not-that-char
                          Enable/Disable linter "Detect use of unwanted
                          chars in files".
  --enable-plugin-text.not-that-char
                          Enable/Disable linter "Detect use of unwanted
                          chars in files".
  --plugin-text.not-that-char.warnings <value>
                          Enable/Disable warnings from "Detect use of
                          unwanted chars in files" (current: +A)

  --disable-plugin-text.ocp-indent
                          Enable/Disable linter "Indention with ocp-indent".
  --enable-plugin-text.ocp-indent
                          Enable/Disable linter "Indention with ocp-indent".
  --plugin-text.ocp-indent.warnings <value>
                          Enable/Disable warnings from "Indention with
                          ocp-indent" (current: +A)

  --disable-plugin-text.useless-space-line
                          Enable/Disable linter "Useless space
                          character and empty line at the end of file".
  --enable-plugin-text.useless-space-line
                          Enable/Disable linter "Useless space
                          character and empty line at the end of file".
  --plugin-text.useless-space-line.warnings <value>
                          Enable/Disable warnings from "Useless space
                          character and empty line at the end of file."
                          (current: +A)
```

Plugin On Parsetree

This plugins contains linters that take the parsetree as input.

```
  --disable-plugin-parsetree
                          A plugin with linters on parsetree.
  --enable-plugin-parsetree
                          A plugin with linters on parsetree.

  --disable-plugin-parsetree.check-constr-args
                          Enable/Disable linter
                          "Check Constructor Arguments".
  --enable-plugin-parsetree.check-constr-args
                          Enable/Disable linter
                          "Check Constructor Arguments".

  --plugin-parsetree.check-constr-args.warnings <value>
                          Enable/Disable warnings from
                          "Check Constructor Arguments" (current: +A)

  --disable-plugin-parsetree.code-identifier-length
                          Enable/Disable linter
                          "Code Identifier Length".
  --enable-plugin-parsetree.code-identifier-length
                          Enable/Disable linter
```

```
                          "Code Identifier Length".
  --plugin-parsetree.code-identifier-length.max-identifier-length <int>
                          Identifiers with a longer name will
                          trigger a warning
  --plugin-parsetree.code-identifier-length.min-identifier-length <int>
                          Identifiers with a shorter name will
                          trigger a warning
  --plugin-parsetree.code-identifier-length.warnings <value>
                          Enable/Disable warnings from
                          "Code Identifier Length" (current: +A)

  --disable-plugin-parsetree.code-list-on-singleton
                          Enable/Disable linter "List function
                          on singleton".
  --enable-plugin-parsetree.code-list-on-singleton
                          Enable/Disable linter "List function
                          on singleton".
  --plugin-parsetree.code-list-on-singleton.warnings <value>
                          Enable/Disable warnings from "List function
                          on singleton" (current: +A)

  --disable-plugin-parsetree.phys-comp-allocated-lit
                          Enable/Disable linter "Physical comparison
                          between allocated litterals.".
  --enable-plugin-parsetree.phys-comp-allocated-lit
                          Enable/Disable linter "Physical comparison
                          between allocated litterals.".
  --plugin-parsetree.phys-comp-allocated-lit.warnings <value>
                          Enable/Disable warnings from "Physical
                          comparison between allocated litterals."
                          (current: +A)
```

Plugin On Typedtree

This plugin contains linters that take the typedtree as input.

```
  --disable-plugin-typedtree
                          A plugin with linters on typed tree.
  --enable-plugin-typedtree
                          A plugin with linters on typed tree.

  --disable-plugin-typedtree.fully-qualified-identifiers
                          Enable/Disable linter
                          "Fully-Qualified Identifiers".
  --enable-plugin-typedtree.fully-qualified-identifiers
                          Enable/Disable linter
                          "Fully-Qualified Identifiers".
  --plugin-typedtree.fully-qualified-identifiers.ignore-depth <int>
                          Ignore qualified identifiers of that depth,
                          not fully qualified
  --plugin-typedtree.fully-qualified-identifiers.ignore-operators <bool>
                          Ignore symbolic operators
  --plugin-typedtree.fully-qualified-identifiers.warnings <value>
                          Enable/Disable warnings from "Fully-Qualified
                          Identifiers" (current: +A)

  --disable-plugin-typedtree.polymorphic-function
                          Enable/Disable linter "Polymorphic function".
```

```
  --enable-plugin-typedtree.polymorphic-function
                           Enable/Disable linter "Polymorphic function".
  --plugin-typedtree.polymorphic-function.warnings <value>
                           Enable/Disable warnings from
                           "Polymorphic function" (current: +A)
```

Plugin Complex

This plugin contains linters which takes different inputs to do their checks (e.g. parsetree and typedtree, lexer tokens and source, etc.)

```
  --disable-plugin-complex  A plugin with linters on different inputs.
  --enable-plugin-complex   A plugin with linters on different inputs.

  --disable-plugin-complex.interface-module-type-name
                           Enable/Disable linter "Checks on
                           module type name".
  --enable-plugin-complex.interface-module-type-name
                           Enable/Disable linter "Checks on
                           module type name".

  --plugin-complex.interface-module-type-name.warnings <value>
                           Enable/Disable warnings from "Checks on
                           module type name." (current: +A)
```

Plugin On Semantic Patches

```
  --disable-plugin-patch    Detect pattern with semantic patch.
  --enable-plugin-patch     Detect pattern with semantic patch.

  --disable-plugin-patch.sempatch-lint-default
                           Enable/Disable linter "Lint from semantic
                           patches (default)".
  --enable-plugin-patch.sempatch-lint-default
                           Enable/Disable linter "Lint from semantic
                           patches (default)".

  --plugin-patch.sempatch-lint-default.warnings <value>
                           Enable/Disable warnings from "Lint from
                           semantic patches (default)" (current: +A)

  --disable-plugin-patch.sempatch-lint-user-defined
                           Enable/Disable linter "Lint from semantic
                           patches (user defined).".
  --enable-plugin-patch.sempatch-lint-user-defined
                           Enable/Disable linter "Lint from semantic
                           patches (user defined).".
  --plugin-patch.sempatch-lint-user-defined.warnings <value>
                           Enable/Disable warnings from "Lint from
                           semantic patches (user defined)." (current: +A)
```

4.4 A Short Example

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'.
```

5 Configuration File

5.1 `ocplib-config` Library

`ocplib-config` is a library provided by `ocp-build` . It allows to create options for a project by a specific syntax which will create the corresponding command line option and an entry to a configuration file.

To use them, user can either change the value with the command line option or by setting the configuration file. Command line option will always override the configuration file option.

ocp-lint provide `Lint_config` module which is a high level interface over the `ocplib-config` library.

```
val create_option :
    string list ->
    string ->
    string ->
    int ->
    'a SimpleConfig.option_class ->
    'a ->
    'a SimpleConfig.config_option
```

For example, to create a new option in a plugin we use:

```
let option = MyPlugin.create_option
  "option_short_name"      (* Option short name*)
  "Option short details"   (* Short details *)
  "Option long details"    (* Long details *)
  SimpleConfig.int_option  (* Type of option *)
  default_value            (* Default value *)
```

This will create an entry *option_short_name* in the configuration and the option `--my-plugin-option-short-name` for the command line.

Modifying an existing option To change the current value of an existing option, you can use the `ocplib-config` syntax with the `=:=` operator:

```
option_short_name =:= new_value
```

Getting the value of an option To get the current value of an option, you can use the `get_value_option` function :

```
val get_option_value :  string list -> string
```

For example:

```
let current_val = Lint_config.get_option_value options in ...
```

5.2 The Project Configuration File

Above, a short example of the ocp-lint configuration:

```
(* Module to ignore during the lint. *)
ignored_files = [
]

plugin_text = {

  (* A plugin with linters on the source. *)
  enabled = true

  code_length = {

    (* Enable/Disable linter "Code Length". *)
    enabled = true

    (* Module to ignore durint the lint of "Code Length" *)
    ignored_files = [ ]

    (* Enable/Disable warnings from "Code Length" *)
    warnings = "+A"

    (* Maximum line length *)
    max_line_length = 80
  }

  ocp_indent = {

    (* Enable/Disable linter "Indention with ocp-indent". *)
    enabled = true

    (* Module to ignore durint the lint of "Indention with ocp-indent" *)
    ignored_files = [ ]

    (* Enable/Disable warnings from "Indention with ocp-indent" *)
    warnings = "+A"
  }
}
```

We can enable or disable from the configuration by setting the **enable** variable to `true` or `false`, we can add some module to ignore for each plugin, linter or globally by adding the filename to the corresponding list.

Finally, we can customizable as we want in the configuration file even disable a specific warning from a linter.

6 Database

TODO **cache system**

7 Using Semantic Patches

The `ocplib-sempatch` library was inspired the Semantic Patches from by Coccinelle [3]. 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.

8 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.

9 Related Works

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 [4] 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` [5] is a style-checker that runs as ppx [6] 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 [7] 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.

10 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.

References

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

[2] OCamlPro. ocp-autoconf. `https://www.typerex.org/ocp-build.html`.

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

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

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

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

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