TypeRex user manual

OCamlPro

March 13, 2012

This manual presents the usage of TypeRex, a development environment and toolbox for OCaml developped by OCamlPro and Inria[1].

[1]with support from the Campus Paris Saclay fundation

Introduction

We present a new development environment which improves what was previously available for OCaml, by bringing some of the features which are considered classic in editors for mainstream programming languages.

The TypeRex environment for Emacs is written in OCaml and relies on parts of the OCaml compiler to provide a more accurate semantic view of the OCaml programs which are edited, enabling a set of specific commands and behaviors.

A few additional tools are provided together with the TypeRex environment, which mostly serve to perform the necessary specific processing on the edited programs, namely, provide appropriate *binary annotations* about the source files.

Summary of TypeRex features

- Improved syntax coloring
- Auto-completion of identifiers (experimental)
- Browsing of identifiers: show type and comment, go to definition, cycle between alternate definitions, and "semantic grep"
- Strictly semantic-preserving, local and whole-program refactoring:
  - renaming identifiers and compilation units
  - open elimination and reference simplification
- Robust *w.r.t.* not-recompiled, possibly unsaved buffers
- Scalable (used regularly on a few hundreds of source files)

All the features of the Tuareg mode are also included, even when we provide an equivalent of them.

Feedback

Please report any bugs, unexpected behavior, or unclear documentation, through the Github issue tracker at https://github.com/OCamlPro/typerex/issues. Please let us know also about your most wanted features so that we can optimize our agenda.

Contents

1 TypeRex setup **7**
- 1.1 TypeRex distribution and supported environments . . . 7
  - 1.1.1 System requirements . . . 7
  - 1.1.2 Obtaining TypeRex . . . 7
- 1.2 Installation . . . 7
  - 1.2.1 Build configuration . . . 7
  - 1.2.2 Building . . . 8
  - 1.2.3 Installation . . . 8
  - 1.2.4 Testing the environment . . . 9
- 1.3 TypeRex configuration (optional) . . . 9
  - 1.3.1 TypeRex keys and contextual menu . . . 9
  - 1.3.2 Automatic indentation . . . 9
  - 1.3.3 TypeRex syntax coloring . . . 10
  - 1.3.4 Auto-completion . . . 10
  - 1.3.5 On-the-fly compilation . . . 11
  - 1.3.6 Reconfiguring paths . . . 11

2 TypeRex development environment for Emacs **13**
- 2.1 TypeRex environment setup . . . 13
  - 2.1.1 Generating .cmt(i) files . . . 13
  - 2.1.2 Project configuration: .typerex file . . . 14
- 2.2 Browsing OCaml code with TypeRex . . . 16
  - 2.2.1 Grep (C-o g / C-o t g) . . . 16
  - 2.2.2 Goto-definition (C-o d) . . . 17
  - 2.2.3 Cycle-definitions (C-o a) . . . 17
  - 2.2.4 Comment-definition (C-o c) . . . 17
- 2.3 Refactoring OCaml code with TypeRex . . . 17
  - 2.3.1 Multiple-file undo (C-o u) . . . 18
  - 2.3.2 Renaming (C-o r / C-o t r) . . . 18
  - 2.3.3 Reference pruning (C-o p) . . . 18
  - 2.3.4 Open elimination (C-o q) (for "qualify") . . . 19
- 2.4 Syntax coloring . . . 19
- 2.5 Auto-completion . . . 19

2.6 TypeRex assumptions and supported code . . . 19
2.6.1 Preprocessors . . . 19
2.6.2 Module packs . . . 20
2.6.3 Dealing with outdated binary annotations . . . 20
2.6.4 Permissive behavior . . . 20
2.7 Recovery and debugging . . . 20

3 TypeRex tools **23**
3.1 ocp-type . . . 23
3.2 ocp-wrapper . . . 24

4 Common issues and questions **27**
4.1 General . . . 27
4.2 Project setup / building annotations . . . 28
4.3 Browsing and refactoring . . . 29
4.4 Syntax coloring . . . 30
4.5 Auto completion . . . 30

Chapter 1

TypeRex setup

This chapter describes the process of installing and configuring TypeRex from the sources, on a Unix environment with OCaml already installed.

1.1 TypeRex distribution and supported environments

1.1.1 System requirements

- Linux (32-bit or 64-bit) or MacOS (tested with 10.6.8)
- OCaml $\geq$3.11.2
- Emacs (Tested with 23.2.1, does not work under X-Emacs, reported to work with Aquamacs but with performance issues)
- Gnu diff, present in PATH

1.1.2 Obtaining TypeRex

We recommend using the compressed archive distribution of TypeRex.

Source archive

All versions of TypeRex can be found at http://www.typerex.org/.

1.2 Installation

1.2.1 Build configuration

```
1:  ./configure [options]
```

Useful options (examples):

```
--bindir=/opt/typerex
--disable-version-check
    do not check OCaml version (allows, e.g., ocaml-3.12.1-rc1)
--with-lispdir=~/.emacs.d
    default is a system-wide installation path
--with-ocp-build=<command>
    default is 'ocp-build' if found, `pwd`/boot/ocp-build.boot otherwise
--disable-auto-complete
    default is to install Emacs auto-complete, unless found in lispdir
--enable-cmt
    binary-annotate typerex sources (for typerex developpers)
```

You may also specify an `EMACS` variable, as:

```
1:  ./configure [options] EMACS=aquamacs
```

1.2.2 Building

```
2:  make
```

1.2.3 Installation

```
3:  sudo make install
```

Append the contents of emacs.append to your ∼/`.emacs`:

```
4:  cat emacs.append >>~/.emacs
```

(or paste the contents where you like). The next section shows how to customize this Emacs configuration in order to handle non-standard settings and to fine-tune various TypeRex options.

Warning: TypeRex is incompatible with (the original version of) the `tuareg-mode`. Do not enable both in your `.emacs`.

Optional: enabling typerex for additional source file extensions

By default, the `auto-mode-alist` settings in `emacs.append` automatically start the TypeRex mode for standard OCaml source file extensions. You can add more extensions by writing, for example:

```
(add-to-list 'auto-mode-alist '("\\.eliom" . typerex-mode))
```

Note that if you intend to use TypeRex on these source file for more than syntax coloring, you will also need to register these extensions either globally by adding an option to the server:

```
(setq ocp-server-command "ocp-wizard -add-impl-suffixes .eliom")
```

or in the project description (as explain later). This second option is necessary for OCaml scripts which are detected with an interpreter directive on the first line.

1.2.4 Testing the environment

Open a `.ml` file inside emacs, and switch to the `*Messages*` buffer. If you see the line "Connection established with TypeRex server", the installation is OK.

1.3 TypeRex configuration (optional)

The following additional configuration is done through Emacs customizable variables, and can be changed in two ways:

- By directly editing the .emacs file to change the values.
- Through the Emacs customization mechanism, by choosing "Customize TypeRex Mode" under the TypeRex Menu. Remember to check your `.emacs` for duplicate variable customization, because `setq` assignments are ignored when updating the `custom-set-variables` expression. Also note that many changes require restarting Emacs to become effective, or at least running `M-x ocp-restart-server`.

1.3.1 TypeRex keys and contextual menu

You may change the default prefix key `(C-o)` for calling TypeRex commands. Just add the following to your `.emacs`:

```
;; Changing the TypeRex prefix key:
(setq ocp-prefix-key [(control j)])
```

The contextual menu can also be enabled when right-cliking ; it offers the same actions as the keyboard shortcuts.

```
;; Uncomment to enable typerex command menu by right click
;;(setq ocp-menu-trigger [mouse-3])
```

1.3.2 Automatic indentation

TypeRex currently uses the Tuareg indentation mechanism. To get a result closer to the OCaml programming guidelines described at http://caml.inria.fr/resources/doc/guides/guidelines.en.html Some users prefer to indent slightly less, which is achieved with:

```
(setq typerex-let-always-indent nil)
(setq typerex-with-indent 0)
(setq typerex-function-indent 0)
(setq typerex-fun-indent 0)
```

Another reasonable choice regarding `if-then-else` is:

```
(setq typerex-if-then-else-indent 0)
```

1.3.3 TypeRex syntax coloring

By default, TypeRex proposes a new syntax coloration for OCaml source files. You may disable syntax coloring by setting `ocp-syntax-coloring` to `nil` instead of `t`. Alternatively, you can change the coloring theme/implementation by customizing the variable `ocp-theme`. Here are the possible values:

- `"syntactic"` (default): the new TypeRex coloring, providing extended identifier kind distinction, and smarter comment/string handling
- `"tuareg_like"`: The same TypeRex implementation, tuned to look almost like Tuareg mode (with minor improvements and differences, and with Tuareg faces renamed into `typerex-font-lock-...`)
- `"caml_like"`: Same as `tuareg_like`, with Caml-mode colors (not renamed)
- `"tuareg"` The embedded Tuareg-mode implementation of syntax coloring (again with renamed Tuareg faces).
- `"caml"` The Caml-mode implementation of syntax coloring, which must be installed and present in the path (file `caml-font.el`). You may need to add the following to your `.emacs`:

```
(setq load-path (cons "~/.emacs.d/caml-mode" load-path))
```

1.3.4 Auto-completion

A (still very primitive) contextual identifier completion feature is provided, relying on the Emacs Auto Complete Mode written by Tomohiro Matsuyama. It can be enabled by uncommenting the following lines in `.emacs`:

```
;; AutoComplete
(add-to-list 'load-path "/usr/local/share/emacs/site-lisp/auto-complete-mode")
(setq ocp-auto-complete t)

;; Using <‘> to complete whatever the context, and <C-‘> for ‘
(setq auto-complete-keys 'ac-keys-backquote-backslash)

;; I want immediate menu pop-up
(setq ac-auto-show-menu 0.)

;; Short delay before showing help
(setq ac-quick-help-delay 0.3)

;; Uncomment to disable help showing
;;(setq ac-use-quick-help nil)

;; Number of characters required to start (nil to disable)
(setq ac-auto-start 0)
```

If you like auto-completion, you can enable it for all supported modes (using dictionaries):

```
;;;; Uncomment to enable auto complete mode globally (independently of OCaml)
;;(require 'auto-complete-config)
;;(add-to-list 'ac-dictionary-directories
;;             "/usr/local/share/emacs/site-lisp/auto-complete-mode/ac-dict")
;;(ac-config-default)
```

Warning! Do not set a black background using `M-x invert-face default` when using auto-completion ; instead, start Emacs with `emacs -r`.

Auto completion keys

Changing the default keys and behavior can be useful to fit the user's habits. For example, the provided configuration `'ac-keys-backquote-backslash` keeps normal behavior of `TAB`, `RET`, `<up>`, and `<down>`, and uses respectively `<‘>`, `<\>`, `<C-p>` and `<C-up>` instead, remapping `''` to `<C-‘>`.

On some keyboards, `<²>` and `<$>` may be more appropriate than `<‘>`, `<\>`. This is achieved by setting `auto-complete-keys` to `ac-keys-two-dollar`

Setting `auto-complete-keys` to `nil` will use the default auto-complete configuration, that is, with `TAB`, `RET`, `<up>`, and `<down>`. Alternatively, one may want to use all standard keys, but with auto-start disabled and using a specific trigger key.

```
;; Standard keys but starting only with C-TAB, and no auto-start
(setq ac-auto-start nil)
(setq auto-complete-keys 'ac-keys-default-start-with-c-tab)
```

Finally, you can also define a customized `'ac-keys-`... function by looking at the implementation of predefined ones, or set the auto-complete configuration variables directly, but then make sure to set `auto-complete-keys` to `nil`. See the Auto Complete Mode user manual for more details on the corresponding configuration.

1.3.5 On-the-fly compilation

By setting `ocp-flymake-enabled`, you can enable an on-the fly compilation button, using flymake and ocamlbuild (contributed by Wojciech Meyer). This feature and its assumptions are undocumented.

1.3.6 Reconfiguring paths

The values of all the paths described in the following are determined at configure time, so normally you should not need to change them, unless you didn't perform the install step.

Emacs lisp directory

The directory in which the TypeRex (and auto-complete) lisp code is searched for is configured by the `(add-to-list 'load-path ...)` line in `emacs.append`. Note that if you ran `configure` with default options the value should be `/usr/local/share/emacs/site-lisp` which is in the default load-path, so this line is probably even not required.

TypeRex server command

The TypeRex development environment is implemented by means of a server which is launched by Emacs. The command which is fed to the shell to launch the server is defined by the variable `ocp-server-command`. You can use it for example to pass a particular OCAMLLIB environment variable (see bellow) to the server. In a standard setup, the corresponding line in emacs.append is also not required as the default value is the executable base name, and is searched in the path.

OCaml standard library

Most TypeRex executables (most notably the server) need to access the OCaml standard library at runtime. They look at the following options, in decreasing priority order, to determine the appropriate directory:

1. `typerex-library-path`, if specified in `.emacs`
2. `OCAMLLIB` environment variable
3. `CAMLLIB` environment variable
4. option `ocamllib` in `~/.ocp/ocaml.conf`
5. the value determined by the configure script when building typerex.

Chapter 2

TypeRex development environment for Emacs

This chapter explains how to enable and use the TypeRex environment for editing an OCaml program.

2.1 TypeRex environment setup

Using the TypeRex environment for an OCaml project requires two configuration steps: ensuring the generation of required binary annotations, and providing a minimal description of the project's paths. Those steps are detailed in the following.

Keeping binary annotations up-to-date

The binary annotations must be up-to-date for TypeRex to function properly, in particular for refactoring, and these annotations are not updated by TypeRex itself. This implies that successive refactoring actions require a recompilation of the impacted part of the program at each step.

2.1.1 Generating .cmt(i) files

The most simple way of generating binary annotations is to setup your build process to use the provided `ocp-*` versions of the OCaml compilers, for example `ocp-ocamlc.opt` instead of `ocamlc.opt`. These are wrappers which behave as the original compilers, but additionally run `ocp-type` on the sources.

In some cases, a more expressive solution is required which consists in prefixing the compiler commands with `ocp-wrapper -save-types` with specific arguments (see chapter "Tools" for more details).

Here are examples of how to achieve this depending on your build system.

make

Use as compiler a variable defined by

```
        OCAMLC=ocp-ocamlc.opt
or
        OCAMLC=ocp-wrapper -save-types [<other options>] ocamlc.opt
```

ocamlbuild

Add

```
        Options.ocamlc := S [ A "ocp-ocamlc"]
or
        Options.ocamlc :=
          S [ A"ocp-wrapper"; A"-save-types"; ... ; A"ocamlc"]
```

to your `myocamlbuild.ml` file. Another option is to invoke ocamlbuild with options:

```
ocamlbuild -ocamlc ocp-ocamlc.opt -ocamlopt ocp-ocamlopt.opt
```

Finally, don't forget t add `CMT _build` to your `.typerex` file (see below).

ocamlfind (without ocamlbuild)

Add

```
        ocamlc = "ocp-ocamlc.opt"
```

or

```
        ocamlc(typerex) = "ocp-ocamlc.opt"
```

to your `/etc/findlib.conf` (or `ocamlfind.conf`, or the file pointed to by `$OCAMLFIND_CONF`). The first option tells `ocamlfind` to use `ocp-wrapper` globally ; the second defines a toolchain "typerex" which you then specify by calling

```
 ocamlfind -toolchain typerex ocamlc
```

Using a separate build process

Alternatively, ocp-type provides a file Makefile.ocp-type.template, which is able to perform the ocp-type compilation automatically for simple projects.

2.1.2 Project configuration: .typerex file

Most functionalities of TypeRex rely on some knowledge of the edited program (source files, libraries...) which should be specified in a very simple project file at the "root" of its source tree with name `.typerex`. When TypeRex is invoked on a source file `<file.ml>`, it looks for file `.typerex` in the directory containing `<file.ml>`, or its parent directories, back to the file system's root. This file is read at each command invocation (except syntax coloring and auto-completion) so modifications are taken into account immediately.

Syntax of .typerex files

The `.typerex` file should specify the set of directories to search for OCaml source files, and the set of directories to include in the load path (*i.e.*, libraries). It is also possible to exclude some source files or whole compilation units, or to force files to be included whatever their extension (if any).

The syntax of the `.typerex` file is as follows:

```
<project file> := <line>* // file must end with a newline

<line> := <dirs>        // add all contained source files (non-recursive)
        | I<dirs>       // include <dirs> as external libraries
                        // +<dir> means <stdlib>/<dir>
        | -<files>      // ignore these source files
        | IMPL <files>  // treat these as implementation files
        | INTF <files>  // treat these as interface files
        | CMT <dir>     // search .cmt(i)s here instead of in source dir
        | NOSTDLIB      // do not include the standard library path
        | #<comment>

<dirs> := white-space-separated list of directories
<files> := white-space-separated list of files
```

Relative directory names are interpreted with respect to the directory containing the project file `.typerex`, and the project directory itself may be denoted by '.', but the shortcuts '∼' and '∼`user`' are not supported. Note that `-<prefix>` is a shorthand for `-<prefix>.ml <prefix>.mli ...` See `.typerex` in the TypeRex root directory for an example.

Meaning of project and library directories

Lines starting with `I` indicate that the specified directories are considered as *library* and not as project's directories. The meaning of this distinction, which may change in the future, is currently the following:

- All source files (.ml, .mli, .mll, .mly) in a project directory are considered, whether they have corresponding compiled files (.cmi, .cmti, .cmt) or not, while compiled files without sources are ignored. This is exactly the opposite for libraries: all .cmi, .cmti, .cmt are considered, and uncompiled sources are simply ignored.

- Refactoring and browsing stops at the boundary of libraries, and no binding propagation is performed on the implementation of libraries (see the documentation for renaming and grep). This saves some computation time and is sound unless a library depends on the program (but the same question arises when the considered program is meant to be a library)

Pack modules (experimental)

Pack modules are understood by TypeRex if the source directories contain either a file

- `pack.mlpack` in the ocamlbuild format: a list of module names, possibly qualified (using `/`) by a path relative to the directory containing the `pack.mlpack` file, or
- `pack.cmt`, whose contents is a pack module (such as generated by `ocp-type -pack`. This option only works if the packed modules are in the same directory as the resulting pack, which is not the case when compiling with `ocamlbuild`.

Other options

`CMT <dir>` **syntax:** It is possible to specify a `CMT` directory to search for `.cmt(i)` files when they are not found at the same place as the source files. This is needed if the build system moves the files around, but then if several modules (in different directories) have the same name, then outdated cmts won't be assigned unless there only is one (matching with the source digest to resolve ambiguity).

`IMPL <files>, INTF <files>:` Use this to use TypeRex on source files with special extension (or none). The subdirectory containing these files still needs to be specified in the `.typerex` file. You will also have to enable TypeRex mode for those, either manually with `M-x typerex-mode`, or by extending `auto-mode-alist` or `interpreter-mode-alist` (see `emacs.append`).

`NOSTDLIB`**:** Do not implicitely include the standard library path. This is required when using TypeRex on the OCaml compiler.

Fallback

If no specific configuration is provided, TypeRex considers as program the set of OCaml source files present in the directory containing the edited source file, with no libraries other than the OCaml stdlib.

2.2 Browsing OCaml code with TypeRex

Note on browsing commands: Each cursor motion incurred by a browsing action (except when clicking on grep results) is undoable with the standard Emacs shortcut (C-_).

2.2.1 Grep (C-o g / C-o t g)

(C-o g) display a click-able list (compile minor mode) of the connected definitions and occurrences of the identifier under the cursor. Invocation is the same as for renaming. Use (C-o t g) to grep the top-level module defined by the current file instead of an identifier.

2.2.2 Goto-definition (C-o d)

Places the cursor on the definition of the identifier under the cursor, opening the appropriate file in the current window if necessary.

2.2.3 Cycle-definitions (C-o a)

Places the cursor on an alternate definition of the identifier declaration under the cursor, opening the appropriate file in the current window if necessary. The typical effect is to switch between .ml and .mli files, but at the right place. This may be used only for top-level let-bindings (i.e. 'let' and not 'let..in', external statements, type declarations, exception declarations, and (recursive) module and module type declarations

2.2.4 Comment-definition (C-o c)

Display a description of the identifier under the cursor, with its lookup path, and any comments associated with it (in the sense of OCamldoc). The description is:

- the type, for a value or field
- the type declaration, for a type constructor
- the argument types (or "constant"), for a constructor or exception
- the module type, for a module or module type.

2.3 Refactoring OCaml code with TypeRex

Note on reverting and undoing: For all refactoring actions, the reverting of modified buffers and the undoing take one of the two following modes:

- If the modification is local to the current buffer, then it is reverted while keeping its history, and renamed if needed. This enables undoing with the standard emacs shortcut (C-_).
- If several files are modified, then all relevant buffers are reverted and their "local" undo-lists are cleared. Instead, the multiple-file modification is added to a global undo list and can only be undone with "C-o u". A call to "C-o u" is also pushed onto the local undo lists of all modified buffers for convenience, so that (C-_) will also work.

2.3.1 Multiple-file undo (C-o u)

Undo the last multiple-file modification. Warning! This discards any subsequent modification of the affected files (a confirmation is asked in this case). All buffers editing one of the affected files are reverted, and their local undo lists are cleared (and then receive a single new "global-undo" item).

2.3.2 Renaming (C-o r / C-o t r)

Rename an identifier through an OCaml program.

(C-o r): The cursor must be placed on an identifier definition or reference (for example, a let binding or a pattern).

(C-o t r): Rename the top-level module defined by the current file instead of an identifier.

Renaming takes care of necessary propagation (e.g., when distinct values with the same name need to be renamed consistently because this name appears in a common interface), and capture is detected.

Renaming is implemented for: values, types, modules (non-recursive), module types, fields, constructors, and exceptions. Aa a convenience, a partial, unsound renaming of classes and class types is supported, but will miss all references to the "secondary" bindings of a class or class type, *i.e.*, the closed and open types, and, for a class, the class type. Type variables, instance variable, methods, argument labels, and polymorphic variants are not supported.

The replacement is intended to be complete, up to the following known bugs:

- labels, e.g. renaming `x` in `let x = .. in f ~x` yields `f ~y` instead of `f ~x:y`, and similarly with `fun ~x -> ..`
- renaming a type which is in fact a class or a class type, or such that its renaming "propagates" to one (through module constraints and functor applications) will not rename the class or class type itself, or its references.

Note also the following limitation:

- including a module where an element is renamed with an afterwards masked name causes a capture error.

2.3.3 Reference pruning (C-o p)

Simplify the identifier references (longidents) by removing unnecessary qualification. This operation ranges on the current buffer.

2.3.4 Open elimination (C-o q) (for "qualify")

Remove (if possible) the open statement under the cursor and qualify the subsequent references as required. the `let open .. in` syntax is also supported by open elimination. This operation is currently slightly conservative, when the same module is opened again inside one of the items in the elimination scope (sub-modules, `let open`, and `M.(...)`) but a duplicate open at the same level will be correctly handled.

2.4 Syntax coloring

TypeRex implements its own version of syntax coloring. It is not yet fully stable, but already has some new features such as the inline marking of lexing errors (with help-info) and a smarter treatment of unterminated strings and comments.

Syntax coloring is not specialized for ocamlyacc/ocamllex files, but will usually give an acceptable result except for C-style comments.

2.5 Auto-completion

An experimental completion feature is proposed in typerex, currently only for identifiers (including methods, tags, labels and type variables). Once enabled, a menu of candidates is triggered when typing test or with the appropriate key (`<‘>` by default) which also completes the longest common prefix. Other keys allow to select a candidate and insert it (`<C-n>`, `<C-p>`, and `<\>` by default), or to cycle between them (with `<TAB>`, see the Auto Complete Mode user manual).

The candidates computation takes into account the load path which is configured for the project, the open and include statements and unqualified identifiers until the current position in the edited file (in a very approximative and simplistic way) and the module qualification possibly prefixing the identifier to be completed.

2.6 TypeRex assumptions and supported code

2.6.1 Preprocessors

The browsing commands of TypeRex support ocamlyacc/ocamllex sources, and should work with other pre-processors which generate OCaml source files with appropriate line numbers directives. More precisely, the identifiers in a pre-processed source file which are actual identifiers of the source (i.e., not generated or transformed during pre-processing) should be OK to grep or jump from and to, if no generated code has the same location.

For ocamlyacc and ocamllex files, these "actual" identifiers correspond to the quoted OCaml code (between braces). Jumping to ocamlyacc entry points is not supported however, because the generated interface has no line number directives. Renaming may work

in pre-processed or ocamllex/ocamlyacc source files, but has not been thoroughly tested. Other refactoring commands won't work on ocamlyacc/ocamllex sources.

The `camlp4` pre-processor (version 3.12.1) is supported, but only partially because its output is an ast which has insufficient location information (or a source file but without line numbers directives). `ocp-type` (or `ocp-wrapper`) can generate binary annotations with `camlp4`, but the result of TypeRex commands will sometimes be inaccurate on camlp4-processed sources (in particular, renaming should only be attempted for local or unexported value bindings).

2.6.2 Module packs

Module packing is supported to the extent of its treatment in the project description (see above), but is still experimental (and with the limitation that "goto" does not go through packs while "grep" does, as for include directives).

2.6.3 Dealing with outdated binary annotations

TypeRex is usually able to overcome sparse changes to the edited files (saved or not) *w.r.t* the last compiled version, and to recompute the right positions. This feature relies on the source snapshots which are embedded in .cmt files. This works also for refactoring commands, but in this case a confirmation will be asked before proceeding.

2.6.4 Permissive behavior

Some internal errors which could occur while processing some files (for example due to unhandled language features) may be caught and reported to the user (asking for a confirmation in the case of refactoring). This avoids giving up too soon on errors which are clearly harmless to a specific action.

2.7 Recovery and debugging

Except for restarting the server, this section is more intended to developing and debugging TypeRex.

Errors and server restart

If the OCP server crashes for any reason (or becomes crazy), it is possible to restart it using

```
M-x ocp-restart-server
```

Logging:

First, the TypeRex environment for Emacs will echo minimal information as messages in the mini-buffer, the history of which is kept in the special buffer `*Messages*`. This includes the startup procedure, feedback about the executed commands, and in case of unexpected error (which is a bug), a complete exception backtrace.

You may enable logging of debug information in `~/.ocp-wizard-log` by setting the `ocp-debug` variable to `t` (the trace will be huge and hard to read). The value of `ocp-debug` may also be a string, which is a comma-separated list (without whitespace) of uncapitalized module names in the TypeRex code.

Fail fast

In the context of debugging, it is usually easier to disable most exception handling to get a backtrace closer to the real problem. This can be done by setting `ocp-dont-catch-errors` to `t`. Note however that this will lead TypeRex to fail in cases which would normally have triggered tolerant behavior.

Profiling

TypeRex may dump profile information in `~/.ocp-wizard-profile.out` if `ocp-profile` is set to the name of a TypeRex command (see `tools/ocp-wizard/main/owzServer.ml`). Run `profile ~/.ocp-wizard-profile.out` to generate a dot file (note that `profile` is not compiled or installed by default).

Chapter 3

TypeRex tools

This chapter summarizes the command-line tools which are provided together with the TypeRex environment.

3.1 ocp-type

The `ocp-type` command-line tool is the "type-only" OCaml compiler which is used to extract the binary annotations needed by the TypeRex environment. It may be invoked directly or through the `ocp-wrapper` tools (see below). It accepts the same options and arguments as `ocamlc`, and the specific option `-save-types`, to actually write the binary data to a file.

Pre-processing

`ocp-type` also accepts pre-processors through the `-pp` option, which should output either

- an OCaml source file (if possible with line-number directives to allow an accurate use of TypeRex), or
- an OCaml dumped AST of one of the supported versions (this is the default behavior of `camlp4`), or
- a Camlp4 dumped AST of one of the supported versions, which can be achieved by passing the option `-printer Camlp4AstDumper` to `camlp4` (this is done automatically if you use `ocp-wrapper`).

Note that the third option is better than the second one with respect to locations (the accuracy of which is instrumental to TypeRex working).

Libraries

`ocp-type` accepts the special form

```
ocp-type -save-types -a <units> -o <target>
```

where

- the argument units may be `cmi`, `cmo`, `cmti`, `cmt`, `mli`, or `ml` files, and
- the target is ignored.

Only arguments ending in `ml` or `mli` will be considered, and typed into `cmt` or `cmti` files.

Module packs

`ocp-type` accepts the special form

```
ocp-type -save-types -pack <units> -o <target>
```

where

- the argument units may be `cmi`, `cmo`, `cmti`, `cmt`, `mli`, or `ml` files, and
- the target may be a `cmo` or `cmt` file.

Howerver, input will always look at `cmt` or `cmti` (or `cmi` as a fallback) files (possibly generating them if `ml` or `mli` are given), and the output will always be written in a `cmt` file. If a `mli` file exists for the name of the pack, then `ocp-type` will look for a compiled interface file for it in `cmti` format (or `cmi` as a fallback) and match the result of packing the arguments against this signature.

3.2 ocp-wrapper

The command `ocp-wrapper` and the specialized commands `ocp-ocamlc`, `ocp-ocamlopt`, `ocp-ocamlc.opt`, and `ocp-ocamlopt.opt` simplify the generation of binary annotations by invoking `ocp-type` with the appropriate options, as part of the usual compilation commands. Calling

```
ocp-wrapper -save-types <command> <options and arguments>
```

where `<command>` is one of the OCaml compilers first invokes this compiler with the exact same options and arguments, and then runs `ocp-type` (unless the command-line was a linking-only phase) with the right options and arguments which are deduced from the original command ones.

Shortcuts

The four `ocp-*` commands are shortcuts for `ocp-wrapper -save-types <command>` which are useful when a single executable program is required as compiling command.

Options

- `-with-ocp-type`, `-with-ocamlc`, ...: allow to customize the `ocp-type`, `ocamlc`, ... commands which are run.
- `-v` print the `ocp-type` command which is executed on stderr.

Pre-processing

Any `-pp` option appearing in the command line is transformed as follows when passed to `ocp-type`:

- if the pre-processor command is (a variant of) `camlp4`, and unless option `-no-wrap-camlp4` is passed to `ocp-wrapper`, then the option `-printer Camlp4AstDumper` is added, which yields more accurate location information,
- otherwise, the command is left unchanged.

Libraries and Module Packs

`ocp-wrapper` accepts the special forms

```
ocp-wrapper -save-types <compiler> -a <units> -o <target>
ocp-wrapper -save-types <compiler> -pack <units> -o <target>
```

Arguments and targets ending in `cmx` are converted into `cmo` and targets into `cmt` as appropriate (see the documentation for `ocp-type`).

Chapter 4

Common issues and questions

4.1 General

Reporting bugs

Bugs reports should be sent through the issue tracker at https://github.com/OCamlPro/typerex/issues (or by mail). If the problem is an uncaught exception (of the form "Error:. . . "), make sure to provide the full backtrace (available in the `*Messages*` buffer). In addition to the action which triggered the problem, an accurate description of your code (and configuration) will be helpful, especially for a wrong result (for example, incomplete).

How are caml-mode, tuareg-mode, and TypeRex related

- `caml-mode` is the original Emacs mode for OCaml, providing indentation, syntax coloring, interactive interpreter and debugger support, and some semantic-level functions using `.annot` files.

- `tuareg-mode` (mostly) improves the `caml-mode`, re-implementing several features, but may use some functions of `caml-mode` if present (from caml-types and caml-help).

- The Emacs version of TypeRex (which is currently the only one available) builds on the `tuareg-mode` (which is included up to a few modifications and a systematic renaming), providing additional functions (and offering an alternative for syntax coloring). It should not be enabled as the same time as the original `tuareg-mode` in your `.emacs`, or conflicts will arise. Optional interaction with the `caml-mode` should behave exactly the same, but has not been tested, except for syntax coloring.

What about prior Caml-mode/Tuareg-mode customization

As explained above, since TypeRex embeds the Tuareg mode, with functions typically renamed from `tuareg-...` to `typerex-...`, you should make sure to disable Tuareg when using TypeRex, to avoid risking conflicts. Furthermore, if you previously customized these modes (*e.g.*, indentation or coloring settings), you can probably just replace the appropriate names in your existing configuration. See the manual about getting Tuareg-mode colors. To use the Caml-mode for coloration instead, you can just add the following to your `.emacs` (replacing the caml-mode path):

```
(setq ocp-syntax-coloring nil)
(setq load-path (cons "/path/to/the/caml-mode" load-path))
(if window-system (require 'caml-font))
(add-hook 'typerex-mode-hook 'caml-font-set-font-lock)
```

What about other editors and operating systems

Eclipse support is planed in a near future. Some people have also expressed interest in VIM support, so we pay attention to this too. Windows support is also planed (there may not be a lot of work to do for that).

How does it compare to other similar tools

Many other tools exist to improve the development in OCaml, and it is not possible to describe each of them here. Generally, we believe the refactoring and semantic grep capabilities of TypeRex to be the most advanced, (except for the particular case of Oug, whose very expressive graph description allows similar queries). Identifier querying is slightly more powerful and robust than its equivalents in, *e.g.*, Tuareg, OCamlSpotter, or OcaIDE, with some minor differences in user interface choices. TypeRex does not yet have graphical summaries as in ODT or OCaIDE, or build management. Syntax coloring is more detailed and systematic than other solutions. Auto completion is currently restricted to identifiers (and very approximate for local identifiers), and does not support syntactic constructs such as pattern matching (as was proposed in OcamlWizard and the latest version of OCamlSpotter), but it is already quite accurate and responsive on identifiers defined in other modules.

4.2 Project setup / building annotations

Findlib configuration does not work

We are investigating possible issues with findlib itself.

How to use TypeRex with Ocamlbuild projects

First, see the setup instructions on how to generate the `.cmt(i)` files (this usually amounts to calling `ocamlbuild -ocamlc ocp-ocamlc.opt -ocamlopt ocp-ocamlopt.opt`). Then add a line reading `CMT _build` to your `.typerex` file.

What are .cmt(i) files exactly required for

The binary annotations provide a semantic descriptions of (type-able) source code, together with accurate location information. They are used to know about binding in a large sense, for example applying a functor to an argument somehow "binds" the functor's parameter's signature members to the actual argument's members. Navigation and grep will only range over binary-annotated code, which can generally depend

on libraries without requiring annotations for them, as long as cmi files are available (with some loss of completeness though). Completion can use .cmi or .cmt(i) files, but the latter (and access to the source code) will enable comment showing. Syntax coloring does not require any annotation at all, of course.

How to enable TypeRex for developping the OCaml compiler

Just copy this patch into the main directory, and run `patch -p0 -i ocaml-typerex.patch`, then rebuild the compiler starting from the standard library (mandatory): `make clean`, `make world`, ... Please note the following limitations:

- `ocp-type` will fail on camlp4 except for version 3.12.1 (for binary compatibility reasons). This does not impact the use of TypeRex for the remaining of the compiler.
- Including Camlp4 in the program is currently not possible.
- Generated files (other than ocamllex/ocamlyacc) are not detected, so don't expect a fully automatic renaming of *e.g.* `List.iter`.
- The `Dynlinkaux` pack module is not correctly understood by TypeRex (because its components are in another directory).

4.3 Browsing and refactoring

{module, value, ...} x not found [in load-path]

This means that some identifiers could not be resolved, and can occur in many situations. For a toplevel module, this probably means a configuration problem (check your `.typerex`).

Grep, Cycle, and renaming sometimes lag

These command can take up to several seconds on large projects, because some computations have to be done over the whole code. Most of it is cached, though, to speed up subsequent invocations (caching should be correctly invalidated on a cmt file basis).

"Goto definition" fails, but "Grep" finds the definition

This is expected if the identifier comes from, for example, an included module, or a pack module (more generally if the identifier is internally renamed by the OCaml compiler during type inference). The grep (or renaming) algorithm takes these renamings into account to collect the full set of relevant identifiers, so it is currently more powerful.

Locations are shifted, TypeRex complains about unsaved files

TypeRex uses the Emacs auto-save mechanism to know about modified buffers, so that it can usually re-align shifted positions correctly. This is not a perfect solution though, and in particular, possible auto-save files (`#file.ml#`) from older sessions will confuse TypeRex, if they are more recent than the file itself. In this case you should delete them.

How does TypeRex deal with multiple toplevel modules with the same name

TypeRex was designed with this issue in mind, and uses full-path identification of toplevel-modules, together with a careful multiple load-paths management and digest-based cmt assignment. However, the

currently limited project configuration file `typerex` lacks expressiveness to accurately describe such settings (the load path is the same for all source files) so this is not yet fully supported (and more testing is required). For example, TypeRex should typically be able to deal correctly with multiple `Main` modules (if no other module depends on them), but not with multiple `Misc` or `Util` modules.

4.4 Syntax coloring

Coloring is sometimes inaccurate

The current implementation has exact lexing information, but only uses heuristics for approximate syntax computation (because it needs to work for syntaxically incorrect buffers). This solution is not perfect and also suffers from some thresholds which are introduced to keep it responsive enough.

4.5 Auto completion

Completion is inaccurate for definitions in the current buffer

The current implementation of completion is semantic for external compilation units (i.e., with a .cmt(i) or .cmi, but only lexical for the current buffer, so this is expected.

Completion stops working

We still have to spot the cause of this problem. Reverting the current buffer with `M-x revert-buffer` (after saving the file, of course) should bring it back.

Emacs deadlocks

This was a known bug, which used to happen when typing during a buffer's initialization unless disabling `ocp-pre-cache`, but should be solved now (let us know if you see it again). The solution is to run `killall ocp-wizard` to kill the TypeRex server process (which unlocks Emacs) and then restart it from the TypeRex menu.