Argot 1.1

Reference Manual

http://argot.x9c.fr

August 5, 2012

Contents

Chapter 1

Overview

1.1 Purpose

The OCaml[1]distribution contains a tool named `ocamldoc` whose duty is to produce API documentation by extracting information from source comments. Argot is an enhanced HTML generator for `ocamldoc` providing additional text formatting, additional tags, and support for symbol search. The name "Argot" stems from the following acronym: *Argot is a Raised Generator for the Ocamldoc Tool*.

1.2 License

Argot is distributed under the terms of the GPL version 3. This licensing scheme should not cause any problem, as documentation generation will not contaminate code. Moreover, generated documentation files can be used without any restriction.

1.3 Contributions

In order to improve the project, I am primarily looking for testers and bug reporters. Pointing errors in documentation and indicating where it should be enhanced is also very helpful.
Bugs and feature requests can be made at `http://bugs.x9c.fr`.
Other requetes can be sent to argot@x9c.fr.

[1]The official OCaml website can be reached at `http://caml.inria.fr` and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.

Chapter 2

Building Argot

2.1 Step 0: dependencies

Before starting to build Argot, one first have to check that dependencies are already installed. The following elements are needed in order to build Argot:

- OCaml, version 4.00.0;
- `make`, in its GNU Make 3.81 flavor;
- a classical Unix shell, such as `bash`;
- **optionally:** Findlib[1], version 1.3.3.

2.2 Step 1: configuration

The configuration of Argot is done by executing `sh configure`. One can specify elements if they are not correctly inferred by the `configure` script; the following switches are available:

- `-ocaml-prefix` to specify the prefix path to the OCaml installation (usually `/usr/local`);
- `-ocamlfind` to specify the path to the `ocamlfind` executable;
- `-no-native-dynlink` to disable the build of the native version of the generator, even if native dynamic linking is available.

2.3 Step 2: compilation

The actual build of Argot is launched by executing `make all`. When build is finished, it is possible to run some simple tests by running `make tests`.

2.4 Step 3: installation

Argot is installed by executing `make install`. According to local settings, it may be necessary to acquire privileged accesses, running for example `sudo make install`. The actual installation

[1] Findlib, a library manager for OCaml, is available at http://projects.camlcity.org/projects/findlib.html.

directory depends on the use of `ocamlfind`: if present the files are placed inside the Findlib hierarchy, otherwise they are placed in the directory printed by the `ocamldoc -customdir` command (*i. e.* `$PREFIX/lib/ocaml/ocamldoc/custom`).

Chapter 3

Running Argot

3.1 Through direct call to `ocamldoc`

Without Findlib

Once built and installed, using Argot is as simple as switching from:

`ocamldoc -html -d` *`destination-path files`*

or

`ocamldoc.opt -html -d` *`destination-path files`*

to:

`ocamldoc -g argot.cmo -d` *`destination-path files`*

or

`ocamldoc.opt -g argot.cmxs -d` *`destination-path files`*

With Findlib

Argot execution is akin to the previous section, except that the use of findlib implies to pass the installation directory to the `ocamldoc` tool, thus leading to:

`ocamldoc -i` *`argot-path`* `-g argot.cmo -d` *`destination-path files`*

or

`ocamldoc.opt -i` *`argot-path`* `-g argot.cmxs -d` *`destination-path files`*

where *`argot-path`* is ‘`ocamlfind query argot`‘.

3.2 Through `ocamlbuild`

Generating documentation through `ocamlbuild` implies to first create a *`modules`*`.odocl` file containing the list of modules (one per line) to generate documentation for. Then, documentation can be generated by executing `ocamlbuild` *`modules`*`.docdir/index.html`. In order to make use of Argot, the following line should be added to the `_tags` file:

`"`*`modules`*`.docdir/index.html": argot`

which marks the index file with the `argot` tag.

It is thus necessary to handle the aforementioned tag by defining an `ocamlbuild` plugin. This implies to create a `myocamlbuild.ml` file with the contents presented by code sample 1.

The main code of the plugin calls the `define_argot` to replace the default HTML generator with Argot. The `define_argot` function takes four parameters:

- an optional `search` parameter (defaulting to `true`), indicating whether search information should be generated;
- an optional `frame` parameter (defaulting to `true`), indicating whether search criteria should be presented in a frame;
- an optional `fulltext` parameter (defaulting to `true`), indicating whether full-text search information should be generated;
- a mandatory parameter of type `(string * string) list`, giving the definitions used by the `TOKEN` formatter.

All these parameters correspond to command-line switches whose meaning are detailed in chapter 4 and 6.

Code sample 1 Example of `ocamlbuild` plugin.

```
open Ocamlbuild_plugin
open Ocamlbuild_pack

let define_argot ?(search=true) ?(frame=true) ?(fulltext=true) defs =
  let rec is_native spec = (* heuristics *)
    match spec with
    | S (hd :: _) -> is_native hd
    | A x | P x | Px x | Sh x -> Filename.check_suffix x ".opt"
    | V "OCAMLDOC" ->
        (try ignore (Command.search_in_path "ocamldoc.opt"); true with _ -> false)
    | Quote x -> is_native x
    | _ -> true in
  let gen =
    if is_native !Options.ocamldoc then
      [A"-g"; A"argot.cmxs"]
    else
      [A"-g"; A"argot.cmo"] in
  let switches =
    if search then
      [A"-search"]
      @ (if frame then [A"-search-frame"] else [])
      @ (if fulltext then [A"-full-text"] else [])
    else
      [] in
  let defs =
    List.map
      (fun (k, v) -> [(A"-define"); (A k); (A v)])
      defs in
  let defs = List.flatten defs in
  flag ["argot"] (S (gen @ switches @ defs));
  let myocamldoc tags =
    Ocaml_tools.ocamldoc_l_dir (tags -- "extension:html") in
  rule "ocamldoc: argot"
    ~prod:"%.docdir/index.html"
    ~dep:"%.odocl"
    ~stamp:"%.docdir/html.stamp"
    ~insert:`top
    (Ocaml_tools.document_ocaml_project ~ocamldoc:myocamldoc
       "%.odocl"
       "%.docdir/index.html"
       "%.docdir")

let () =
  dispatch begin function
    | After_rules ->
        define_argot ["a", "b"]
    | _ -> ()
  end
```

Chapter 4

Argot formatting

4.1 Text formatting

In addition to the already available {`b ...`} (for bold), {`i ...`} (for italic), and {`e ...`} (for emphasized), Argot provides the following text formatting modifiers:

- {`s ...`} for stroke;
- {`u ...`} for underline;
- {`h ...`} for highlight.

4.2 Tables

In order to define tables, the following elements can be used:

- {`table ...`} to actually define a table;
- {`caption ...`} to define its associated caption;
- {`row ...`} to add a row to the table;
- {`header ...`} to add an header cell to the row;
- {`data ...`} to add a data cell to the row;
- {`span` *`n`* `...`} (where n should be a positive integer) to add a data cell spanning n columns to the row.

Code sample 2 give an example of a complete table definition.

Code sample 2 Example of a table.

```
{table {caption the caption}
       {row {header key} {header value}}
       {row {data key1}  {data {i data1}}}
       {row {data key2}  {data {i data2}}}
       {row {span 2 summary}}}
```

4.3 Token substitution

Token substitution allows one to use the value of either an environment variable or a command-line switch into an HTML page. This may for example be useful to insert the current date, or to specify the path of an element at documentation generation time:

```
{token DATE}
file {token FILE_PATH}/file.ext
```

The value of a token is first searched in `-define` *`id value`* switches passed to the `ocamldoc` tool, and then searched among the environment variables.

Instead of passing multiple `-define` switches, it is also possible to load a bunch of definitions from a file through the `-definitions` *`file`* switch. Each line should have the following form: *`id=value`*, where the `#` character introduces comments (ending with the end of the line). Lines that do not conform to the given format are ignored.

Finally, it is possible to refer to other variables when defining a new one (as long as this does not create a circular definition). As an example, the file represented by code sample 3 could be passed to the `-definitions` switch.

Code sample 3 Example of definition file.

```
BASE=/path/to # prefix
NAME=file
FILE=$(BASE)/$(NAME).txt
```

4.4 Images

It is possible to include images into the generated pages through {`image` *`path`*}. The image data will be directly embedded into the page using base64 format, in such a way that no external link remains in the generated HTML. There is thus no need to package the image along with the HTML pages.

To avoid to use the full path to the image, it is possible to use the aforementioned token substitution inside the `image` formatter:

```
{image {token IMAGE_PATH}/img.png}
```

Then, one has to specify the value for `IMAGE_PATH` on the command-line through the `-define` switch seen above:

```
ocamldoc -define IMAGE_PATH /path/to/images ...
```

4.5 Folding

When some explanation, albeit useful, is long and/or may appear as a digression, it is possible to *fold* it. It means that the texte inside a {`fold` *digression*} will appear as an ellipsis (*i.e.* ...)

and will be *unfolded* (that is revealed) when the ellipsis will be clicked. At the opposite, clicking on the ellipsis while the *foldable* text is visible will make it disappear. Any formatting instruction can be used in the digression.

Chapter 5

Argot tags

Argot also defines a bunch of new tags that can be used to enhance documentation. Some of these tags come with image icons; these have been designed by Mark James, released under the Creative Commons Attribution 2.5 License, and are available at http://www.famfamfam.com/lab/icons/silk/.

The additional tags are:

- `@obvious`, a bare placeholder;
- `@typevar`, to document type variables in the same way `@param` is used for parameters;
- `@copyright` and `@license` to be used along with the `@author` parameter;
- `@alias`, `@synonym`, and `@equivalent`, to define synonyms or equivalences;

- `@todo`, or `@unimplemented`, to mark an implementation-in-progress;
- `@todoc`, or `@docme`, to mark a documentation-in-progress;
- `@tofix`, or `@fixme`, to mark a fix-in-progress;

- `@stateful`, to mark that a given function relies on a state;
- `@threadsafe`, to mark that a given function can be used in a multithread context;
- `@threadunsafe`, to mark that a given function cannot be used in a multithread context;

- `@attention`, to introduce text by the  icon;
- `@bug`, to introduce text by the  icon;
- `@error`, to introduce text by the  icon;
- `@info`, to introduce text by the  icon;
- `@new`, to introduce text by the icon;

- `@note`, to introduce text by the  icon;
- `@remark`, to introduce text by the  icon;
- `@warning`, to introduce text by the  icon.

The `@license` tag will only print its argument if it does not recognize the license, but will create a link to the actual webpage of the license otherwise. The list of predefined licenses is the following (the recognition being case insensitive):

- `gpl`, `gpl1`, and `gplv1` will point to http://www.gnu.org/licenses/old-licenses/gpl-1.0.html
- `gpl2`, and `gplv2` will point to http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
- `gpl3`, and `gplv3` will point to http://www.gnu.org/licenses/gpl.html
- `lgpl`, and `lgplv2` will point to http://www.gnu.org/licenses/old-licenses/lgpl-2.0.html
- `lgpl21`, `lgpl2.1`, `lgpl2_1`, `lgplv21`, `lgplv2.1`, and `lgplv2_1` will point to http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html
- `lgpl3`, and `lgplv3` will point to http://www.gnu.org/licenses/lgpl.html
- `agpl` will point to http://www.gnu.org/licenses/agpl.html
- `bsd` will point to http://www.freebsd.org/copyright/license.html
- `mit` will point to http://www.opensource.org/licenses/mit-license.php
- `apache` will point to http://www.apache.org/licenses/
- `qpl` will point to http://doc.trolltech.com/3.0/license.html
- `cecill`, and `cecill-a` will point to http://www.cecill.info/licences/Licence_CeCILL_V2-en.html
- `cecill-b` will point to http://www.cecill.info/licences/Licence_CeCILL-B_V1-en.html
- `cecill-c` will point to http://www.cecill.info/licences/Licence_CeCILL-C_V1-en.html

Chapter 6

Search

Argot is able to embed search facilities[1] into the generated pages when passed the `-search` command-line switch. The default behaviour is to open a search window by clicking on the magnifying glass (*i.e.* ) that appears on the `index.html` page as well as on each module page. However, it is possible to have the search pane in a frame by passing the `-search-frame` command-line switch; `argot_index.html` is then the main file.

The search panel is composed of two parts: the upper part allowing to select search criteria, and the lower part displaying results. Clicking on a search result will make the main window point to the clicked element.

Five search modes are provided:

- search by exact name ;
- search by regular expression over names ;
- search by type;
- search by type using manifests;
- full-text search.

Search by name can be made on simple names as well as on fully qualified names. Moreover, all names can be used : values, types, modules, constructors, *etc.*

Search by regular expression is akin to search by name except that all names matching the passed regular expression are returned. The regular expression should be written using the JavaScript syntax[2]. As an example, it is possible to retrieve all the elements starting with `fold` by using the following regular expression: `fold.*`.

Search by type allows to search for a value by giving its type[3]. It is based on the ideas put forward by Roberto Di Cosmo (in particular the `ocamlsearch` tool for CamlLight). This means

[1] Search features need JavaScript to be available and enabled in the browser used to consult the documentation.
[2] See `http://www.w3schools.com/jsref/jsref_obj_regexp.asp` for reference.
[3] As of version 1.1, type-based search is still experimental and does not support the following features: objects, polymorphic variants, and modules.

that the search does not retrieve values with *exactly* the same type, but with values whose types are *isomorphic*[4]. Practically this means that for example:

- `int -> float -> int`, `float -> int -> int`, and `float * int -> int` are isomorphic (because the differences are only in the way and order parameters are passed);
- `string` and `unit -> string` are isomorphic (because one always has a `unit` value at hand);
- `'from list -> ('from -> 'to) -> 'to list` and `('a -> 'b) -> 'a list -> 'b list` are isomorphic (because type variables can be renamed, and parameters reordered).

note 1: labels are just ignored during search
note 2: a signature containing optional arguments is considered as the set of all signatures that can be generated by including/discarding the optional arugments

Search by type using manifests is similar to bare search by type, except that types are replaced by their manifests if provided. More, in this context, record types are mapped to tuples by droping the field names. Practically, this means that searching for type `float * float -> float` will return (among others) `Complex.norm` whose type is `Complex.t -> float`. Indeed, as `Complex.t` has manifest { `re : float; im : float;` }, it is clearly isomorphic to `float * float`.

Full-text search is the ability to search words appearing in the documentation. Words should be separated by spaces, and the result set will present the elements whose documentation contains all the words.

[4]See http://www.dicosmo.org/ResearchThemes/ISOS/ISOShomepage.html for more information.