Cafesterol – version 1.3

http://cafesterol.x9c.fr

September 19, 2009

Abstract: This document presents Cafesterol, its purpose and its relation to OCaml-Java. This document explains how to build, and how to run Cafesterol, as well as its compatibility level with the standard Objective Caml compilers.

Introduction

OCaml-Java is an effort to make Objective Caml[1] available on the Java[2] platform, currently supporting version 3.11.1. The project has two concrete objectives: first, the ability to run Objective Caml sources that have been compiled using `ocamlc`; second, the ability to compile Objective Caml sources into executable Java jar files.

Cafesterol is an extension of the Objective Caml compiler suite that generates Java bytecode. Cafesterol provides an `ocamljava` compiler that is the Java counterpart of `ocamlc`/`ocamlopt` compilers shipped with the Objective Caml standard distribution.

Cafesterol, in its 1.3 version builds with the 3.11.1 version of Objective Caml. The produced Java classes need the 1.3 version of Cadmium to run (precisely, the `ocamlrun.jar` file) and can be executed on any Java 1.6 virtual machine.

Build process

Cafesterol is written in the Objective Caml language. It thus needs `ocamlc`/`ocamlopt` to be installed, as well as the Objective Caml standard library. It also depends on Barista[3] and Camlzip[4]. Barista in turn also depends on Camomile[5]. This dependency chain implies that the build process is quite involved.

In the following, the mandatory steps allow to compile `ocamljava` (and `ocamljava.opt`) compilers as well as the standard library and *other* libraries from the standard distribution (bigarray,

[1] The official Caml 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.
[2] The official website for the Java Technology can be reached at `http://java.sun.com`.
[3] Library for Java class file manipulation – available at `http://barista.x9c.fr`.
[4] Library for zip/gzip/jar manipulation – available at `http://cristal.inria.fr/~xleroy/software.html`.
[5] Unicode library – available at `http://camomile.sourceforge.net`.

dbm, dynlink, graph, labltk, num, str, systhreads, threads, and unix). From this point, it is possible to use `ocamljava` like `ocamlc` or `ocamlopt` to produce some Java binaries from `ml`/`mli` sources.

The optional steps allow to produce Cafesterol-compiled version of `ocamlc`, `ocamlopt`, and `ocamljava` (it results in `ocamlc.jar`, `ocamlopt.jar`, and `ocamljava.jar`), as well as some additional tools. Steps from seven to nine are not required to build `ocamlc.jar` or `ocamlopt.jar`; they are only required to build `ocamljava.jar`.

The following steps suppose you want to install all the elements into their default locations.

First step: build Objective Caml 3.11.1 To build Objective Caml from source, after the source archive has been unzipped, it is sufficient to run `./configure` followed by `make word.opt` and then `make install` as root.

Second step: install Camomile 0.7.2 To build Camomile from source, after the source archive has been unzipped, it is sufficient to run `./configure` followed by `make` and then `make install` as root.

Third step: install Camlzip 1.04 To build Camlzip from source, after the source archive has been unzipped, it is sufficient to run `make all allopt` and then `make install installopt` as root. You may need to edit the Makefile to modify the location of the Zlib C library.

Fourth step: install Barista 1.3 To build Barista from source, after the source archive has been unzipped, it is sufficient to run `make all` and then `make install-all` as root. You may need to edit the Makefile to modify the paths.

Fifth step: compile Cafesterol First, uncompress the Cafesterol source distribution. Then, copy all files from its `src` directory into the directory that contain the Objective Caml source distribution you created at step 1 (the simplest is to use `cp -R /path/to/cafesterol/src/* /path/to/ocaml-3.11.1`). You can now build Cafesterol by running `make -f Makefile-cafesterol` from the Objective Caml directory. At last, run `make -f Makefile-cafesterol install` to install the Cafesterol compiler (both `ocamljava` and `ocamljava.opt`) as well as the Objective Caml libraries built with these compilers.

Sixth step: install Cadmium 1.3 To build Cadmium from source, after the source archive has been unzipped, it is sufficient to run `ant deploy-base` and `make all`, followed by `ant install-base` and `make install` as root.

[optional] Seventh step: compile Camomile with Cafesterol First, get the `Makefile-cafesterol` for Camomile from the Cafesterol website. Then, copy this file into the unarchived source directory of Camomile you created at step 2. Finally, run `make -f Makefile-cafesterol all`, and `make -f Makefile-cafesterol install` as root.

[optional] Eighth step: compile Camlzip with Cafesterol First, get the `Makefile-cafesterol` for Camlzip from the Cafesterol website. Then, copy this file into the unarchived source directory of Camlzip you created at step 3. Finally, run `make -f Makefile-cafesterol all`, and `make -f Makefile-cafesterol install` as root.

[optional] Ninth step: compile Barista with Cafesterol Just run `make cafesterol` and `make install-cafesterol` (as root) from the Barista directory.

[optional] Tenth step: compile compilers with Cafesterol Return to the Objective Caml source distribution directory and run `make -f Makefile-cafesterol compilers` and `make -f Makefile-cafesterol install-compilers` as root. It will build and install `ocamlc.jar`, `ocamlopt.jar`, and `ocamljava.jar`.

[optional] Eleventh step: compile tools with Cafesterol Still from the Objective Caml source distribution directory, run `make -f Makefile-cafesterol tools` and `make -f Makefile-cafesterol install-tools` as root. It will build and install `ocamllex.jar`, `ocamldoc.jar`, `ocamldep.jar`, and `ocamlbuild.jar`.
The `ocamllex.jar` and `ocamldoc.jar` tools are identical to their *classical* counterparts. The `ocamldep.jar` tool is almost identical to its counterpart, except that the `cmx` extension is replaced with `cmj`. Finally, the `ocamlbuild.jar` tool is an enhanced version, compared to the original one: it has builtin rules to compile files using the `ocamljava` compiler. In this version, support for `ocamlbuild.jar` is experimental[6], and one should use the target `module-standalone.jar` to compile with `ocamljava`, where `module.byte` and `module.native` would be used with the original `ocamlbuild`.

[optional] Twelfth step: compile camlp4 with Cafesterol From the Objective Caml source distribution directory, run `make -f Makefile-cafesterol camlp4` and `make -f Makefile-cafesterol install-camlp4` as root. It will build and install the camlp4 libraries and tools.

Overview of the compiler

The `ocamljava` compiler tries to mimic the behaviour of the standard compiler as much as possible. It accepts the same source files as input. Table 1 shows the file types used and produced by the various compilers. On a side note, one may notice that `.jo` files follow the `jar` file format. `.jo` and `.jar` files contain two entries per Objective Caml module: a Java class file and a `.consts` file (it represents the module constants in Objective Caml marshalled format). An executable `jar` file also contain a `cafesterolMain` class that is the entry point of the executable.

File kind	`ocamlc`	`ocamlopt`	`ocamljava`
interface source	`.mli`	`.mli`	`.mli`
implementation source	`.ml`	`.ml`	`.ml`
compiled interface	`.cmi`	`.cmi`	`.cmi`
compiled implementation	`.cmo`	`.cmx`	`.cmj`
implementation binary	-	`.o`	`.jo`
compiled library	`.cma`	`.cmxa`	`.cmja`
library binary	-	`.a`, `.so`, ...	`.jar`
plugin	-	`.cmxs`	`.cmjs`
additional primitives	`.c`	`.c`	`.java`

Table 1: Files types for the various compilers.

[6] Which means, among other things, that its behaviour and rules may change in future releases.

Detailed information about runtime is available in the Cadmium documentation. In particular, the Cadmium documentation explains how to enhance the standard runtime with additional primitives (for Objective Caml `external` declarations).

Options

The `ocamljava` compiler recognizes the following options:

`-a` build a library (`.cmja` and `.jar` files) from passed `.cmj` files;

`-additional-class` *file* add the class to the created jar file;

`-additional-file` *file*[: *path*] add the file to the created jar file (*path* is the path of the file inside the jar file);

`-additional-jar` *file* add the file to the dependency list;

`-applet` link as an applet (the applet class is `cafesterolApplet` in the package set by `-java-package`);

`-annot` dump type information (in `.annot` file);

`-c` compile only (no link);

`-cadmium-parameter` *name=value* add the binding to the Cadmium runtime parameters for the produced executable (the following section lists such parameters);

`-classpath` *cp* add the passed element to the classpath (used to find primitive providers beside the builtin ones);

`-compact` optimize for space rather than for speed;

`-config` print configuration and exit;

`-dtypes` same as `-annot` (deprecated);

`-for-pack` *module* generate an object file that will be later used as a submodule;

`-g` generate debugging information;

`-i` print inferred signature for module;

`-I` *dir* add the given directory to the list of search directories;

`-impl` *file* treat the given file as an implementation source;

`-inline` *n* set inlining hint to *n*;

`-intf` *file* treat the given file as an interface source;

`-intf-suffix` *s* set the interface suffix to *s*;

`-intf_suffix` *s* set the interface suffix to *s*;

`-java-package` *pkg* set the java package for the produced class files;

`-javac` *comp* set the java compiler[7];

`-jopt` *opt* pass additional option to Java compiler;

`-labels` use commuting label mode;

`-noassert` disable assertion checks;

`-nobuiltin` do not use builtin primitive list;

`-nolabels` ignore non-optional labels in types;

`-nomerge` do not merge service descriptors;

`-nostackmap` do not generate stack maps;

`-nostdlib` do not use the standard library;

`-o` *file* set output file to *file* (by default, the executable `jar` files are named `camlprog.jar`);

`-pack` package the passed `.cmj` files;

`-pp` *command* use preprocessor;

`-provider` *fully.qualified.ClassName* adds the passed class to the list of primitive providers;

`-principal` check principality of type inference;

`-rectypes` allow arbitrary recursive types;

`-scripting` compile for Java scripting (internal use);

`-servlet` *file* link as a servlet (passed file is `web.xml` servlet descriptor);

`-shared` produce a dynlinkable plugin;

`-standalone` link in standalone mode (no dependency, the contents of all referenced jar files is copied into the produced jar);

`-thread` generate code supporting threads;

`-unsafe` disable bound checks;

`-v` print compiler version, standard library location and exit;

`-version` print compiler version and exit;

`-verbose` print external calls before execution;

`-w` *list* enable/disable warnings according to list (supports the same convention and the same warnings as the standard compilers);

`-warn-error` *list* treats passed warnings are errors;

`-where` print location of standard library and exit.

[7]The Java compiler is only used if `ocamljava` is presented a `.java` file on the command line, just the same way `ocamlc`/`ocamlopt` uses the C compiler.

Cadmium parameters

The `-cadmium-parameter` switch allows to specify runtime parameters for a Cafesterol-compiled program. The following parameters are recognized:

- `backtrace` either `on` or `off` (defaulting to `off`):
  whether exception backtrace should be written;
- `exitStoppingJVM` either `on` or `off` (defaulting to `on`) :
  whether to stop JVM upon program exit (disable to have multiple programs within the same JVM);
- `awt` either `on` or `off` (defaulting to `off`):
  whether to use AWT for `Graphics` frame (otherwise, Swing is used);
- `javaxSound` either `on` or `off` (defaulting to `off`):
  whether to use `javax.sound` package for `Graphics` beeps (otherwise, mono-tone system beeps are used);
- `jdbm` either `on` or `off` (defaulting to `off`):
  whether to use the `jdbm` package for `Dbm` implementation (otherwise, `java.util` classes are used);
- `os` any of `Unix`, `Cygwin`, `Win32`, `MacOS`, or `Cadmium` (defaulting to `Unix`):
  OS value returned by `Sys` module;
- `unixEmulation` either `on` or `off` (defaulting to `off`):
  whether to enable unix emulation (use of command-line utilities to replace missing primitives);
- `embedded` either `on` or `off` (defaulting to `off`):
  whether to enable embedded mode;
- `embeddedBase` either `on` or `off` (defaulting to `""`):
  base class for embedded mode;
- `simplifiedBacktrace` either `on` or `off` (defaulting to `on`):
  whether to enable simplified backtrace (does not print methods of classes from the Cadmium runtime, of `java.lang` or `java.lang.reflect` packages).

Running executables produced by ocamljava

Following the Java philosophy, the `jar` file generated for a program does only contain the code for this program but not the code of any linked library. In this respect, `ocamljava` is very different from both `ocamlc` and `ocamlopt` that generate statically-linked standalone binary files.
In order to run a `jar` file generated for a program, it is thus necessary to provide the JVM with all the libraries the program depends on. According to the manifest file contained by the program `jar` file, the best way to provide the libraries to the JVM is to copy the `jar` files corresponding to the libraries into the directory of the program `jar` file. The `jar` files of the libraries are generated along with the `cmja` files when using the `-a` switch of the compiler.
At last, the produced program `jar` files also need `ocamlrun.jar` (available at http://cadmium.

x9c.fr) to run.

However, a second linking mode is provided: a static linking mode following the Objective Caml philosophy. This mode is triggered by the `-standalone` switch. In this linking mode, the contents of all referenced jar files is copied into the produced jar file. The produced jar file is thus really standalone (no dependency), at the expense of a bigger produced jar file.

Compatibility with the standard compilers

`ocamljava` is almost fully compatible with the compilers from the standard distribution. However it differs on the following points:

- tail calls are optimized only for direct recursion, not for calls to another function (due to a Java limitation forbidding cross-method jumps);
- object cache is not implemented (it should not alter the behaviour of a program except under race conditions between threads, as the lack of object cache is essentially a performance issue);
- evaluation order is not guaranteed to be the same as in `ocamlc`/`ocamlopt` (however, it should not be a major problem as evaluation order is not specified in the Objective Caml language);
- pending signals are checked at given points in the generated code, which may result in a worse reactivity to signals compared to standard compilers;
- stack overflow as well as memory shortage are not diagnosed neither by Cafesterol nor the Cadmium runtime (it is the JVM that will encounter these limits and raise a Java error);
- backtrace support is rudimentary;
- generated code is subject to compatibillity issues of Cadmium compared to the standard runtime support (one should refer to the Cadmium documentation for more information);
- `ocamljava` may raise a compilation error that has no equivalent in `ocamlc`/`ocamlopt`: "Cannot compile x (Java method is too long)". Such an error is raised when the generated Java method would be more than 65535-byte long[8] (the limit in the current Java specification). To overcome this error, there are mainly two solutions: (i) split the corresponding function into several functions, or (ii) decrease the parameter passed to the `-inline` command-line switch, if any. Due to the way inlining works, one may have to decrease the inlining aggressiveness not (only) for the file failing to compile but (also) for files it depends upon.

Some incompatibilities with the standard Objective Caml distribution arise from primitive implementations. The detailed compatibility information, on a per-primitive basis can be found in the `cadmium-compatibility.pdf` file available at http://cadmium.x9c.fr/downloads.html.

[8] Moreover, this error may be raised when the method size would be between 32768 and 65535 byte long. In this case, the error is not caused by a Java limitation but by an `ocamljava` restriction added to keep compilation simple.