Bisect – version 1.0

http://bisect.x9c.fr

December 17, 2009

Abstract: This document presents Bisect, its purpose and the way it works. This document is structured in three parts explaining first how to build, and how to run Bisect. Then, the last part lists known issues. Additionally, an appendix describes the output file formats.

Introduction

Bisect is a code coverage tool for the Objective Caml language[1]. Its name stems from the following acronym: *Bisect is an Insanely Short-sized and Elementary Coverage Tool*. The shortness of the source files can be seen as a tribute to the camlp4 tool and API bundled with the standard Objective Caml distribution.

Code coverage is a mean of software testing. Associated with for example unit or functional testing, the goal of code coverage is to measure the portion of the application source code that has actually been tested. To achieve this goal, the code coverage tool defines *points* in the source code and memorizes at runtime (that is, when tests are run) if the execution path of the program passes at these *points*. The so-called *points* are places of interest in the source code (as an example, the branches of an *if* or *match* construct are interesting *points*), to ensure that all alternatives have been tested. In practice, code coverage is often performed in three steps:

- first, the application is *instrumented*: this means that (the compiled form of) the application is enhanced in such a way that it will count at runtime how many times the application passed at a given *point*;
- then, the tests are actually run, producing some runtime-data about code coverage;
- finally, a report is generated from the data produced at the previous step; this report shows which points were actually passed through during tests.

Bisect can be seen as an improved version of the `ocamlcp`/`ocamlprof` couple (both of these tools being part of the standard Objective Caml distribution). In this respect, Bisect performs statement and condition coverage, but not path coverage. This means that it only counts how many times the application passed at each *point*, independently of which was the statement previously executed (that is, the previously visited point). At the opposite, path coverage is not only interested in *points*

[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.

but also in *paths*, the goal being to ensure that every possible execution path has been followed.

Code coverage is a useful software metric but, being based on tests, it cannot ensure that a program is correct. For program correction, one should consider more involved tools and formalisms such as *model checking*, or *proof systems*. Code coverage is still convenient in practice because it is a much simpler method that require no particular knowledge from the developer.

Bisect, in its 1.0 version, is designed to work with version 3.11.1 of Objective Caml. Bisect is released under the GPL version 3. This licensing scheme should not cause any problem, as instrumented applications are intended to be used during developement but should not be released publicly. The GPL contamination has thus no consequence here.
Bugs should be reported at `http://bugs.x9c.fr`.

Building Bisect

Bisect can be built from sources using `make`, and Objective Caml version 3.11.1. Under usual circumstances, there should be no need to edit the `Makefile`. Bisect is compiled by executing the command `make all` and installed by executing the command `make install` with root privileges. The following targets are available:

`all` compiles all files, and generates html documentation

`common` compiles the 'Common' module

`runtime` compiles the 'Runtime' module

`instrument` compiles the 'Instrument' module

`report` compiles the report executable

`html-doc` generates html documentation

`clean-all` deletes all produced files (including documentation)

`clean` deletes all produced files (excluding documentation)

`clean-doc` deletes documentation files

`install` copies executable and library files

`ocamlfind` installs through ocamlfind

`tests` runs the tests

`depend` populates the dependency files (they are initially empty)

One may notice that the `instrument` module will be compiled into Ojective Caml bytecode only, while the other modules will be compiled into bytecode as well as native formats (and even Java format if the `ocamljava`[2] compiler is present).

[2] OCaml compiler generating Java bytecode, by the same author – `http://ocamljava.x9c.fr`

Running Bisect

As previously stated, using a code coverage tool usually requires to follow three steps: instrumentation, execution, and report. Bisect is no exception in this respect; the following sections discuss each of these three steps.

Instrumenting the application

Bisect instruments the application at compile-time using a camlp4-based preprocessor. It allows the user to choose exactly which module (*i.e.* source file) of the application should be instrumented. Code sample 1 shows how to instrument a file named `source.ml` during compilation (the very same effect can be achieved using either `ocamlopt` or `ocamljava` as a replacement of `ocamlc`). Code sample 2 does the same through `ocamlfind`. During this step, Bisect will produce a file named `source.cmp`[3]. Files with the `cmp` extension contain *point* information for a given source file, that is: identifiers, positions, and kinds of *points*. Of course, the usual `cmi`, `cmo`, `cmx`, and `cmj` files are also produced, depending on the compiler acutally invoked. It is necessary to pass the `-I +bisect` option to the compiler because instrumentation adds calls to functions defined in the runtime modules of Bisect.

Code sample 1 Compiling and instrumenting a file.

```
ocamlc -c -I +bisect -pp 'camlp4o /path/to/instrument.cmo' source.ml
```

Code sample 2 Compiling and instrumenting a file through `ocamlfind`.

```
ocamlfind ocamlc -package bisect -syntax camlp4o -c source.ml
```

It is possible to choose which language constructs should actually be implemented by passing `-enable` and/or `disable` command-line switches to `instrument.cmo`. Both switches are followed by a string describing the kinds of points the user wants to either enable or disable. The possible characters are:

`b` for *binding*

`s` for *sequence*

`f` for *for*

`i` for *if/then*

`t` for *try*

`w` for *while*

`m` for *match/function*

`c` for *class expression*

`d` for *class initializer*

[3]This file will be strored in the very same directory as the `cmo`, `cmx`, or `cmj` file produced by the compiler.

`e` for *class method*

`v` for *class value*

`p` for *toplevel expression*

`l` for *lazy operator*

By default, all point kinds are enabled. As an example, `-disable cdev` will disable instrumentation of all class constructs.

Unsafe mode

When compiling in *unsafe* mode[4], the `-unsafe` switch should be passed to camlp4 instead of the compiler. Indeed, as camlp4 is building a syntax tree that is passed to the compiler, issuing the `-unsafe` switch to the compiler has no effect because it is too late: the code has been built by camlp4 in *safe* mode. In such a case, the compiler warns the user with the following message: `Warning: option -unsafe used with a preprocessor returning a syntax tree`. The correct command-line invocations are shown by code samples 3 and 4.

Code sample 3 Compiling and instrumenting a file using unsafe mode.

```
ocamlc -c -I +bisect -pp 'camlp4o -unsafe /path/to/instrument.cmo' source.ml
```

Code sample 4 Compiling and instrumenting a file using unsafe mode through `ocamlfind`.

```
ocamlfind ocamlc -package bisect -syntax camlp4o -ppopt -unsafe -c source.ml
```

Linking

Linking a program containing instrumented modules is not different from *classical* linking, except that one should link the Bisect library to the produced executable. This is usually done by adding one of the following to the linking command-line:

- `-I +bisect bisect.cma` (for `ocamlc` version);
- `-I +bisect bisect.cmxa` (for `ocamlopt` version);
- `-I +bisect bisect.cmja` (for `ocamljava` version).

In order, to use Bisect in multithread applications, it is necessary to also link with the `BisectThread` module. This also implies to pass the `-linkall` option to the compiler.

[4] One should remind that the usefulness of using the *unsafe* mode in an instrumented application is questionable, as the instrumentation of an application results in very degraded performances.

Running the instrumented application

Running an instrumented application is not different from running any application compiled with an Objective Caml compiler. However, Bisect will produce runtime data in a file each time the application is run. A new file will be created at each invocation, the first one being `bisect0001.out`, the second one `bisect0002.out`, and so on. It is also possible to define the scheme used for file names by setting the `BISECT_FILE` environment variable. If `BISECT_FILE` is equal to *file*, files will be named *file**n**.out* where ***n*** is a natural value padded with zeroes to 4 digits (*i.e.* "0001", "0002", and so on).

Bisect can also be parametrized using another environment variable: `BISECT_SILENT`. If this variable is set to either "YES" or "ON" (defaulting to "OFF", case being ignored), then Bisect will not output any message at runtime. If not silent, Bisect will output a message on the standard error in two situations:

- the output file for runtime data cannot be created at program initialization;
- the runtime data cannnot be written at program termination.

Generating the coverage report

In order to generate the coverage report for the instrumented application, it is sufficient to invoke the `bisect-report` executable (alternatively either `bisect-report.opt`, or `bisect-report.jar`). This program recognizes the following command-line switches:

`-csv <file>` Set output to csv, data being written to given file

`-dump-dtd <file>` Dump the DTD to the given file

`-html <dir>` Set output to html, files being written in given directory

`-I <dir>` Add the directory to the search path

`-no-folding` Disable code folding (HTML only)

`-no-navbar` Disable navigation bar (HTML only)

`-separator <string>` Set the seprator for generated output (CSV only)

`-tab-size <int>` Set tabulation size in output (HTML only)

`-text <file>` Set output to text, data being written to given file

`-title <string>` Set the title for generated output (HTML only)

`-verbose` Set verbose mode

`-version` Print version and exit

`-xml <file>` Set output to xml, data being written to given file

`-xml-emma <file>` Set output to EMMA xml, data being written to given file

`-help` Display this list of options

`--help` Display this list of options

Wherever a destination file is waited, the use of `-` (*i.e.* minus sign) is interpreted as the standard output. The user should also provide on the command-line the list of the runtime data files that should be used to produce the report. As a result, a typical invocation is: `bisect-report -html report bisect*.out` to process all data files in the current directory and generate an HTML report into the `report` directory.

If relative file paths are used at the instrumentation step, the report executable should be launched from the same directory. Another option is of course to use absolute paths. Using absolute path is also useful when playing with the `-pack` option. Indeed, it is possible in this case to have several source files with the same name in different directories and packed to different enclosing modules. In the case of packed modules, absolute paths allows to avoid ambiguities but are not necessary. It is in fact sufficient to have *discriminating* paths, that is: paths that always allow to distinguish files packed in different enclosing modules.
It is also possible to use the `-I` command-line switch to specify a search path for source files.
When the HTML output mode is chosen, a bunch a files is produced: one `index.html` file, and one HTML file per instrumented module. The `index.html` file provides application-wide statistics about coverage, as well as links to the other files. The module files provide module-wide statistics, as well as a duplicate of the module source, enhanced with *point* information. *Points* are represented in the source as special comments having the form `(*[`*n*`]*)` where *n* indicates how many times the *point* was passed at runtime. For easier appreciation, colors are also used to annotate source lines:

- a line will be green-colored if it contains *points* whose values are all strictly positive;
- a line will be red-colored if it contains *points* whose values are all equal to zero;
- a line will be yellow-colored if it contains some *points* whose values are all equal to zero, and some others whose values are strictly positive;
- a line will not be colored at all if it contains no *point*.

When another output mode is chosen, only one file is produced (or none, if `-` is used) containing the whole coverage information. The appendix details the various file formats.

Example

Code sample 5 shows the makefile used for the compilation (with instrumentation), run, and report phases for a one-file application: `source.ml`. Code sample 6 shows the same information when relying on `ocamlfind`.

Known issues

Bisect suffers from the following issues:

- Bisect, being based on camlp4, performs a purely syntactic treatment. It can thus sometimes produce unaccurate results due to semantics subtleties. For a concrete example consider lazy operators: in expressions such as `e1 && e2` or `e1 || e2`, Bisect adds *points* to both *e1* and *e2* to allow the user to know which parts of the whole expression were actually evaluated. However, it is possible that the programmer redefined one of these operator in such a way

Code sample 5 Example makefile.

```
default: clean compile run report

clean:
        rm -fr report
        rm -f *.cm* *.out bytecode

compile:
        ocamlc -c -I +bisect \
            -pp 'camlp4o path/to/bisect/instrument.cmo' source.ml
        ocamlc -o bytecode -I +bisect bisect.cma source.cmo

run:
        ./bytecode

report:
        path/to/bin/bisect-report -html report bisect*.out
```

Code sample 6 Example makefile (`ocamlfind`-based).

```
default: clean compile run report

clean:
        rm -fr report
        rm -f *.cm* *.out bytecode

compile:
        ocamlfind ocamlc \
            -package bisect -linkpkg -syntax camlp4o -o bytecode source.ml

run:
        ./bytecode

report:
        ocamlfind bisect/bisect-report -html report bisect*.out
```

that its new semantics is no more lazy (*e.g.* `let (&&) = (+)`). In this case, Bisect will still add points to subexpressions even if they appear useless[5]. A dual issue would occur if a programmer defined a new operator with lazy semantics (*e.g.* `external (++) : bool -> bool -> = "%sequor"`), in this case Bisect will not define *points* for subexpressions while they would clearly be of interest.

- when linking the tested application, the `Bisect` module should be linked as (one of) the first ones; indeed, the Bisect runtime performs some operations at initialization, such as determining the target file for runtime information: the current working directory should hence not have been modified by another module or should have been modified purposely (it is also possible to use `BISECT_FILE` to specify an absolute path);
- for performance reasons, Objective Caml `int`s are used store store *point* data; it implies that one should not use the report executable on a 32-bit architecture if the tested application has been instrumented and run on a 64-bit architecture[6].

[5] One may notice that it could not be possible to overcome this problem by keeping track of local (*i.e.* file) redefinitions, as the redefinition may occur in another module that has been opened.

[6] This is indeed an over-cautious recommendation, as the Objective Caml gracefully handles platforms differences; one should only get unaccurate results (but not false results: neither an unvisited will be considered as visited, nor the opposite) when working at the 32-bit limit.

Appendix: file formats

CSV file format

The CSV mode outputs statistics line by line: first for the whole application, and then for each file. Each line has the following format: first the path of the source file (`-` being used for the overall application), then 14×2 integer values (13 for the various point kinds, plus one for the total). Each integer couple consists, for each point kind, of (*i*) the number of visited points and (*ii*) the total number of points. The point kinds are output in the following order: *let bindings*, *sequence*, *for loops*, *if/then constructs*, *try/with constructs*, *while loops*, *match/function constructs*, *class expressions*, *class initializers*, *class methods*, *class values*, *top level expressions*, *lazy operators*. Listing 7 shows such an output.

Code sample 7 CSV file format.

```
-;3;3;5;5;1;1;0;0;0;0;0;0;2;2;0;0;0;0;0;0;0;0;0;2;2;13;13
source.ml;3;3;5;5;1;1;0;0;0;0;0;0;2;2;0;0;0;0;0;0;0;0;0;2;2;13;13
```

Text file format

The text mode outputs statistics first for the overall application, and then for each file. The statistics always take the same form, that is the ratio *number of visited points over total number of points* for each point kind, followed by the ratio for all point kind. Listing 8 shows such an output.

XML file format

The XML mode outputs both statistics and information for each of the points in the source files. Listing 9 shows the DTD for produced XML files (it can be generated using the `-dump-dtd` command-line option). Statistics are output for the whole application and for each file inside `<summary` elements, while information relative to each point is encoded into `<point` elements.

XML EMMA-compatible format

This mode outputs only overall statistics, in a format that is compatible with EMMA[7]. This compatibility allows to use Bisect output in tools that provide support for EMMA, notably giving an easy way to use Bisect with continuous integration servers like Hudson[8].

EMMA defines only four categories for coverage: classes, methods, blocks, and lines. Bisect defining more point kinds, the following mapping is used:

- *class expressions*, *class initializers*, and *class values* are merged into the *class* category;
- *class methods* are mapped to the *method* category;
- *let bindings*, *sequence*, *for loops*, *if/then constructs*, *try/with constructs*, *while loops*, *match/function constructs*, and*lazy operators* are merged into the *block* category;

[7] EMMA is a code coverage tool for Java - http://emma.sourceforge.net/
[8] http://hudson-ci.org/

Code sample 8 Text file format.

```
Summary:
 - 'binding' points: 3/3 (100.00 %)
 - 'sequence' points: 5/5 (100.00 %)
 - 'for' points: 1/1 (100.00 %)
 - 'if/then' points: none
 - 'try' points: none
 - 'while' points: none
 - 'match/function' points: 2/2 (100.00 %)
 - 'class expression' points: none
 - 'class initializer' points: none
 - 'class method' points: none
 - 'class value' points: none
 - 'toplevel expression' points: none
 - 'lazy operator' points: 2/2 (100.00 %)
 - total: 13/13 (100.00 %)
File 'source.ml':
 - 'binding' points: 3/3 (100.00 %)
 - 'sequence' points: 5/5 (100.00 %)
 - 'for' points: 1/1 (100.00 %)
 - 'if/then' points: none
 - 'try' points: none
 - 'while' points: none
 - 'match/function' points: 2/2 (100.00 %)
 - 'class expression' points: none
 - 'class initializer' points: none
 - 'class method' points: none
 - 'class value' points: none
 - 'toplevel expression' points: none
 - 'lazy operator' points: 2/2 (100.00 %)
 - total: 13/13 (100.00 %)
```

Code sample 9 DTD for produced XML files.

```
<!ELEMENT bisect-report (summary,file*)>

<!ELEMENT file (summary,point*)>
<!ATTLIST file path CDATA #REQUIRED>

<!ELEMENT summary (element*)>

<!ELEMENT element EMPTY>
<!ATTLIST element kind CDATA #REQUIRED>
<!ATTLIST element count CDATA #REQUIRED>
<!ATTLIST element total CDATA #REQUIRED>

<!ELEMENT point EMPTY>
<!ATTLIST point offset CDATA #REQUIRED>
<!ATTLIST point count CDATA #REQUIRED>
<!ATTLIST point kind CDATA #REQUIRED>
```

- *top level expressions* are mapped to the *line* category.

Another point should be noted regarding this output mode: for all the categories, any 0/0 value is replaced by a 1/1 value. This replacement is justified by the fact that 0/0 results in 0% while 1/1 results in 100%, and one would not want to have a build failure in Hudson due to low coverage.