Bisect – version 1.0-alpha

http://bisect.x9c.fr

July 6, 2008

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.

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 (in fact, 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 practive, 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 produced from the data produced at the previous step; this report shows which points were actually passed through during tests.

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. At the opposite, path coverage is not only interested in *points* but also in *paths*, the goal being to ensure that every possible execution path has been followed.

[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.

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 practive because it is a much simpler method.

Bisect, in its 1.0-alpha version, is designed to work with version 3.10.2 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. Bugs should be reported at http://bugs.x9c.fr.

Building Bisect

Bisect can be built from sources using `make`, and Objective Caml version 3.10.2. Under usual circumstances, there should be no need to edit the `Makefile`. The following targets are available:

`all` compiles all files, adn 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;

`tests` runs the tests.

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` compiler is present).

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 instrument. 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`). During this step, Bisect will produce a file named `source.cmp`. 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.cma' source.ml
```

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).

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"), 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:

`-version` prints version and exit;

`-verbose` sets verbose mode;

`-tab-size <int>` sets tabulation size in output;

`-html <dir>` sets HTML output mode, and writes html files in given directory;

`-help` or `--help` displays the list of options.

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`.

When the HTML output mode is chosen, a bunch a files is produced: one `index.html` file, and one 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 provides 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*.

Example

Code sample 2 shows the makefile used for the compilation (with instrumentation), run, and report phases for a one-file application.

Known issues

Bisect suffers from the following issues:

- Bisect, being based on camlp4, performs a purely syntaxtic 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 redifined one of these operator in such a way 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[2]. A dual issue would occur is 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.

[2] 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.

Code sample 2 Example makefile.

```
default: clean compile run report

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

compile:
	ocamlc -c -I +bisect \
	    -pp 'camlp4o /usr/local/lib/ocaml/bisect/instrument.cma' source.ml
	ocamlc -o bytecode -I +bisect bisect.cma source.cmo

run:
	./bytecode

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

- some *point* labels (*e.g.* `(*[0]*)`) may appear at the wrong place (*e.g.* inside an expression rather than at its start), it is partly due to some location bug in camlp4 (*cf.* bug #0004521 at url http://caml.inria.fr/mantis/view.php?id=4521) but may also be due to Bisect bugs (including in the clumsy code trying to overcome this camlp4 bug);
- 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 (it is also possible to use `BISECT_FILE` to specify an absolute path);
- for performance reasons, Objective Caml `ints` 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[3];
- when using relative paths at the instrumentation step, the report executable should be launched from the same directory.

[3] 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 in the sense of neither an unvisited will not be considered as visited, nor the opposite) when working at the 32-bit limit.