Tools

version 2.1

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Chapter 1

Tools User's Guide

The *Tools* application contains a number of stand-alone tools, which are useful when developing Erlang programs.

cover A coverage analysis tool for Erlang.

cprof A profiling tool that shows how many times each function is called. Uses a kind of local call trace breakpoints containing counters to achieve very low runtime performance degradation.

eprof A time profiling tool; measure how time is used in Erlang programs. Erlang programs. Predecessor of *fprof* (see below).

fprof Another Erlang profiler; measure how time is used in your Erlang programs. Uses trace to file to minimize runtime performance impact, and displays time for calling and called functions.

instrument Utility functions for obtaining and analysing resource usage in an instrumented Erlang runtime system.

make A make utility for Erlang similar to UNIX make.

tags A tool for generating Emacs TAGS files from Erlang source files.

xref A cross reference tool. Can be used to check dependencies between functions, modules, applications and releases.

1.1 cover

1.1.1 Introduction

The module `cover` provides a set of functions for coverage analysis of Erlang programs, counting how many times each executable line [page 8] is executed.

Coverage analysis can be used to verify test cases, making sure all relevant code is covered, and may be helpful when looking for bottlenecks in the code.

1.1.2 Getting Started With Cover

Example

Assume that a test case for the following program should be verified:

```
-module(channel).
-behaviour(gen_server).

-export([start_link/0,stop/0]).
-export([alloc/0,free/1]). % client interface
-export([init/1,handle_call/3,terminate/2]). % callback functions

start_link() ->
    gen_server:start_link({local,channel},channel,[],[]).

stop() ->
    gen_server:call(channel,stop).

%%%-Client interface functions-------------------------------------------

alloc() ->
    gen_server:call(channel,alloc).

free(Channel) ->
    gen_server:call(channel,{free,Channel}).

%%%-gen_server callback functions----------------------------------------

init(_Arg) ->
    {ok,channels()}.

handle_call(stop,Client,Channels) ->
    {stop,normal,ok,Channels};

handle_call(alloc,Client,Channels) ->
    {Ch,Channels2} = alloc(Channels),
    {reply,{ok,Ch},Channels2};

handle_call({free,Channel},Client,Channels) ->
    Channels2 = free(Channel,Channels),
    {reply,ok,Channels2}.

terminate(_Reason,Channels) ->
    ok.

%%%-Internal functions---------------------------------------------------

channels() ->
    [ch1,ch2,ch3].

alloc([Channel|Channels]) ->
    {Channel,Channels};
```

```
alloc([]) ->
    false.

free(Channel,Channels) ->
    [Channel|Channels].
```

The test case is implemented as follows:

```
-module(test).
-export([s/0]).

s() ->
    {ok,Pid} = channel:start_link(),
    {ok,Ch1} = channel:alloc(),
    ok = channel:free(Ch1),
    ok = channel:stop().
```

Preparation

First of all, Cover must be started. This spawns a process which owns the Cover database where all coverage data will be stored.

```
1> cover:start().
{ok,<0.30.0>}
```

Before any analysis can take place, the involved modules must be *Cover compiled*. This means that some extra information is added to the module before it is compiled into a binary which then is loaded [page 9]. The source file of the module is not affected and no `.beam` file is created.

```
2> cover:compile_module(channel).
{ok,channel}
```

Each time a function in the Cover compiled module `channel` is called, information about the call will be added to the Cover database. Run the test case:

```
3> test:s().
ok
```

Cover analysis is performed by examining the contents of the Cover database. The output is determined by two parameters, `Level` and `Analysis`. `Analysis` is either `coverage` or `calls` and determines the type of the analysis. `Level` is either `module`, `function`, `clause`, or `line` and determines the level of the analysis.

Coverage Analysis

Analysis of type coverage is used to find out how much of the code has been executed and how much has not been executed. Coverage is represented by a tuple {Cov,NotCov}, where Cov is the number of executable lines that have been executed at least once and NotCov is the number of executable lines that have not been executed.

If the analysis is made on module level, the result is given for the entire module as a tuple {Module,{Cov,NotCov}}:

```
4> cover:analyse(channel,coverage,module).
{ok,{channel,{14,1}}}
```

For channel, the result shows that 14 lines in the module are covered but one line is not covered.

If the analysis is made on function level, the result is given as a list of tuples {Function,{Cov,NotCov}}, one for each function in the module. A function is specified by its module name, function name and arity:

```
5> cover:analyse(channel,coverage,function).
{ok,[{{channel,start_link,0},{1,0}},
     {{channel,stop,0},{1,0}},
     {{channel,alloc,0},{1,0}},
     {{channel,free,1},{1,0}},
     {{channel,init,1},{1,0}},
     {{channel,handle_call,3},{5,0}},
     {{channel,terminate,2},{1,0}},
     {{channel,channels,0},{1,0}},
     {{channel,alloc,1},{1,1}},
     {{channel,free,2},{1,0}}]}
```

For channel, the result shows that the uncovered line is in the function channel:alloc/1.

If the analysis is made on clause level, the result is given as a list of tuples {Clause,{Cov,NotCov}}, one for each function clause in the module. A clause is specified by its module name, function name, arity and position within the function definition:

```
6> cover:analyse(channel,coverage,clause).
{ok,[{{channel,start_link,0,1},{1,0}},
     {{channel,stop,0,1},{1,0}},
     {{channel,alloc,0,1},{1,0}},
     {{channel,free,1,1},{1,0}},
     {{channel,init,1,1},{1,0}},
     {{channel,handle_call,3,1},{1,0}},
     {{channel,handle_call,3,2},{2,0}},
     {{channel,handle_call,3,3},{2,0}},
     {{channel,terminate,2,1},{1,0}},
     {{channel,channels,0,1},{1,0}},
     {{channel,alloc,1,1},{1,0}},
     {{channel,alloc,1,2},{0,1}},
     {{channel,free,2,1},{1,0}}]}
```

For channel, the result shows that the uncovered line is in the second clause of channel:alloc/1.

Finally, if the analysis is made on line level, the result is given as a list of tuples {Line,{Cov,NotCov}}, one for each executable line in the source code. A line is specified by its module name and line number.

```
7> cover:analyse(channel,coverage,line).
{ok,[{{channel,9},{1,0}},
     {{channel,12},{1,0}},
     {{channel,17},{1,0}},
     {{channel,20},{1,0}},
     {{channel,25},{1,0}},
     {{channel,28},{1,0}},
     {{channel,31},{1,0}},
     {{channel,32},{1,0}},
     {{channel,35},{1,0}},
     {{channel,36},{1,0}},
     {{channel,39},{1,0}},
     {{channel,44},{1,0}},
     {{channel,47},{1,0}},
     {{channel,49},{0,1}},
     {{channel,52},{1,0}}]}
```

For `channel`, the result shows that the uncovered line is line number 49.

Call Statistics

Analysis of type `calls` is used to find out how many times something has been called and is represented by an integer `Calls`.

If the analysis is made on module level, the result is given as a tuple {`Module,Calls`}. Here `Calls` is the total number of calls to functions in the module:

```
8> cover:analyse(channel,calls,module).
{ok,{channel,12}}
```

For `channel`, the result shows that a total of twelve calls have been made to functions in the module.

If the analysis is made on function level, the result is given as a list of tuples {`Function,Calls`}. Here `Calls` is the number of calls to each function:

```
9> cover:analyse(channel,calls,function).
{ok,[{{channel,start_link,0},1},
     {{channel,stop,0},1},
     {{channel,alloc,0},1},
     {{channel,free,1},1},
     {{channel,init,1},1},
     {{channel,handle_call,3},3},
     {{channel,terminate,2},1},
     {{channel,channels,0},1},
     {{channel,alloc,1},1},
     {{channel,free,2},1}]}
```

For `channel`, the result shows that `handle_call/3` is the most called function in the module (three calls). All other functions have been called once.

If the analysis is made on clause level, the result is given as a list of tuples {`Clause,Calls`}. Here `Calls` is the number of calls to each function clause:

```
10> cover:analyse(channel,calls,clause).
{ok,[{{channel,start_link,0,1},1},
     {{channel,stop,0,1},1},
     {{channel,alloc,0,1},1},
     {{channel,free,1,1},1},
     {{channel,init,1,1},1},
     {{channel,handle_call,3,1},1},
     {{channel,handle_call,3,2},1},
     {{channel,handle_call,3,3},1},
     {{channel,terminate,2,1},1},
     {{channel,channels,0,1},1},
     {{channel,alloc,1,1},1},
     {{channel,alloc,1,2},0},
     {{channel,free,2,1},1}]}
```

For channel, the result shows that all clauses have been called once, except the second clause of channel:alloc/1 which has not been called at all.

Finally, if the analysis is made on line level, the result is given as a list of tuples {Line,Calls}. Here Calls is the number of times each line has been executed:

```
11> cover:analyse(channel,calls,line).
{ok,[{{channel,9},1},
     {{channel,12},1},
     {{channel,17},1},
     {{channel,20},1},
     {{channel,25},1},
     {{channel,28},1},
     {{channel,31},1},
     {{channel,32},1},
     {{channel,35},1},
     {{channel,36},1},
     {{channel,39},1},
     {{channel,44},1},
     {{channel,47},1},
     {{channel,49},0},
     {{channel,52},1}]}
```

For channel, the result shows that all lines have been executed once, except line number 49 which has not been executed at all.

Analysis to File

A line level calls analysis of channel can be written to a file using cover:analysis_to_file/1:

```
12> cover:analyse_to_file(channel).
{ok,"channel.COVER.out"}
```

The function creates a copy of channel.erl where it for each executable line is specified how many times that line has been executed. The output file is called channel.COVER.out.

```
File generated from channel.erl by COVER 2001-05-21 at 11:16:38

****************************************************************************

        |  -module(channel).
        |  -behaviour(gen_server).
        |
        |  -export([start_link/0,stop/0]).
        |  -export([alloc/0,free/1]). % client interface
        |  -export([init/1,handle_call/3,terminate/2]). % callback functions
        |
        |  start_link() ->
     1..|      gen_server:start_link({local,channel},channel,[],[]).
        |
        |  stop() ->
     1..|      gen_server:call(channel,stop).
        |
        |  %%%-Client interface functions------------------------------------
        |
        |  alloc() ->
     1..|      gen_server:call(channel,alloc).
        |
        |  free(Channel) ->
     1..|      gen_server:call(channel,{free,Channel}).
        |
        |  %%%-gen_server callback functions---------------------------------
        |
        |  init(_Arg) ->
     1..|      {ok,channels()}.
        |
        |  handle_call(stop,Client,Channels) ->
     1..|      {stop,normal,ok,Channels};
        |
        |  handle_call(alloc,Client,Channels) ->
     1..|      {Ch,Channels2} = alloc(Channels),
     1..|      {reply,{ok,Ch},Channels2};
        |
        |  handle_call({free,Channel},Client,Channels) ->
     1..|      Channels2 = free(Channel,Channels),
     1..|      {reply,ok,Channels2}.
        |
        |  terminate(_Reason,Channels) ->
     1..|      ok.
        |
        |  %%%-Internal functions--------------------------------------------
        |
        |  channels() ->
     1..|      [ch1,ch2,ch3].
        |
        |  alloc([Channel|Channels]) ->
     1..|      {Channel,Channels};
        |  alloc([]) ->
     0..|      false.
```

```
        |
        |  free(Channel,Channels) ->
     1..|      [Channel|Channels].
```

Conclusion

By looking at the results from the analyses, it can be deducted that the test case does not cover the case when all channels are allocated and `test.erl` should be extended accordingly.
Incidentally, when the test case is corrected a bug in `channel` should indeed be discovered.

When the Cover analysis is ready, Cover is stopped and all Cover compiled modules are unloaded [page 9]. The code for `channel` is now loaded as usual from a `.beam` file in the current path.

```
13> code:which(channel).
cover_compiled
14> cover:stop().
ok
15> code:which(channel).
"./channel.beam"
```

1.1.3 Miscellaneous

Performance

Execution of code in Cover compiled modules is slower and more memory consuming than for regularly compiled modules. As the Cover database contains information about each executable line in each Cover compiled module, performance decreases proportionally to the size and number of the Cover compiled modules.

Executable Lines

Cover uses the concept of *executable lines*, which is lines of code containing an executable expression such as a matching or a function call. A blank line or a line containing a comment, function head or pattern in a `case`- or `receive` statement is not executable.

In the example below, lines number 2,4,6,8 and 11 are executable lines:

```
1: is_loaded(Module,Compiled) ->
2:   case get_file(Module,Compiled) of
3:     {ok,File} ->
4:       case code:which(Module) of
5:         ?TAG ->
6:           {loaded,File};
7:         _ ->
8:           unloaded
9:       end;
10:    false ->
11:      false
12:  end.
```

Code Loading Mechanism

When a module is Cover compiled, it is also loaded using the normal code loading mechanism of Erlang. This means that if a Cover compiled module is re-loaded during a Cover session, for example using `c(Module)`, it will no longer be Cover compiled.

Use `cover:is_compiled/1` or `code:which/1` to see if a module is Cover compiled (and still loaded) or not.

When Cover is stopped, all Cover compiled modules are unloaded.

1.1.4 Using the Web Based User Interface to Cover

Introduction

To ease the use of Cover there is a web based user interface to Cover. The web based user interface to Cover is designed to be started and used via WebTool. It is possible to Cover compile Erlang modules and to generate printable Cover and Call analyses via the web based user interface.

Start the Web Based User Interface to Cover

Configure WebTool to manage the web based user interface to Cover, see WebTool User's Guide for more information. Start WebTool and point a browser to the start page of WebTool. Currently the web based user interface to Cover is only compatible with Internet Explorer and Netscape Navigator 4.0 and higher.

Click on the link marked WebCover in the topmost frame of WebTool. The main frame of the browser will then show the web based user interface to Cover.

Cover Compile

To Cover compile a module or all the modules in a given directory select *Compile* in the left frame. Write the filename or the directory to be Cover compiled into the text field labeled *Module or Directory*. If other compile options than the standard compile options is needed, write them in the field labeled *Compile Options*. Click on the button labeled *Compile*. The module(s) will then be Cover compiled

If the name of the directory or file to Cover compile is unknown, it is possible to list the Erlang files in a directory, and change the working directory for the node from the web based user interface. This is done in the right part of the page.

Create Cover and Call Analyses

To generate Cover or Call analysis, Cover compile the file either from the web based user interface or from the command line. Execute the code and drag the mouse over the module name, in the list of modules in the left frame. A pop-up menu will appear, select the wanted action from the menu and the result of the analyse will show up in the right frame.

To narrow the information in the Cover and Call analysis select one of the radio buttons at the top of the page.

View the source code with Line Level Analyze information

To view the source file of a Cover compiled module with additional cover information for each line, drag the mouse over the module name in the left frame and a pop-up menu appears. Select *Source File* in the popup-menu. The source with line level analysis information will then come up in the left frame.

1.2 cprof - The Call Count Profiler

`cprof` is a profiling tool that can be used to get a picture of how often different functions in the system are called.

`cprof` uses breakpoints similar to local call trace, but containing counters, to collect profiling data. Therfore there is no need for special compilation of any module to be profiled.

`cprof` presents all profiled modules in decreasing total call count order, and for each module presents all profiled functions also in decreasing call count order. A call count limit can be specified to filter out all functions below the limit.

Profiling is done in the following steps:

`cprof:start/0..3` Starts profiling with zeroed call counters for specified functions by setting call count breakpoints on them.

`Mod:Fun()` Runs the code to be profiled.

`cprof:pause/0..3` Pauses the call counters for specified functions. This minimises the impact of code running in the background or in the shell that disturbs the profiling. Call counters are automatically paused when they "hit the ceiling" of the host machine word size. For a 32 bit host the maximum counter value is 2147483647.

`cprof:analyse/0..2` Collects call counters and computes the result.

`cprof:restart/0..3` Restarts the call counters from zero for specified functions. Can be used to collect a new set of counters without having to stop and start call count profiling.

`cprof:stop/0..3` Stops profiling by removing call count breakpoints from specified functions.

Functions can be specified as either all in the system, all in one module, all arities of one function, one function, or all functions in all modules not yet loaded. As for now, BIFs cannot be call count traced.

The analysis result can either be for all modules, or for one module. In either case a call count limit can be given to filter out the functions with a call count below the limit. The all modules analysis does *not* contain the module `cprof` itself, it can only be analysed by specifying it as a single module to analyse.

Call count tracing is very lightweight compared to other forms of tracing since no trace message has to be generated. Some measurements indicates performance degradations in the vicinity of 10 percent.

The following sections show some examples of profiling with `cprof`. See also cprof(3) [page 32].

1.2.1 Example: Background work

From the Erlang shell:

```
1> cprof:start(), cprof:pause(). % Stop counters just after start
3476
2> cprof:analyse().
{30,
 [{erl_eval,11,
           [{{erl_eval,expr,3},3},
            {{erl_eval,'-merge_bindings/2-fun-0-',2},2},
            {{erl_eval,expand_module_name,2},1},
            {{erl_eval,merge_bindings,2},1},
            {{erl_eval,binding,2},1},
            {{erl_eval,expr_list,5},1},
            {{erl_eval,expr_list,3},1},
            {{erl_eval,exprs,4},1}]},
  {orddict,8,
          [{{orddict,find,2},6},
           {{orddict,dict_to_list,1},1},
           {{orddict,to_list,1},1}]},
  {packages,7,[{{packages,is_segmented_1,1},6},
               {{packages,is_segmented,1},1}]},
  {lists,4,[{{lists,foldl,3},3},{{lists,reverse,1},1}]}]}
3> cprof:analyse(cprof).
{cprof,3,[{{cprof,tr,2},2},{{cprof,pause,0},1}]}
4> cprof:stop().
3476
```

The example showed the background work that the shell performs just to interpret the first command line. Most work is done by `erl_eval` and `orddict`.

What is captured in this example is the part of the work the shell does while interpreting the command line that occurs between the actual calls to `cprof:start()` and `cprof:analyse()`.

1.2.2 Example: One module

From the Erlang shell:

```
1> cprof:start(),R=calendar:day_of_the_week(1896,4,27),cprof:pause(),R.
1
2> cprof:analyse(calendar).
{calendar,9,
          [{{calendar,df,2},1},
           {{calendar,dm,1},1},
           {{calendar,dy,1},1},
           {{calendar,last_day_of_the_month1,2},1},
           {{calendar,last_day_of_the_month,2},1},
           {{calendar,is_leap_year1,1},1},
           {{calendar,is_leap_year,1},1},
           {{calendar,day_of_the_week,3},1},
           {{calendar,date_to_gregorian_days,3},1}]}
```

```
3> cprof:stop().
3271
```

The example tells us that "Aktiebolaget LM Ericsson & Co" was registered on a Monday (since the return value of the first command is 1), and that the `calendar` module needed 9 function calls to calculate that.

Using `cprof:analyse()` in this example also shows approximately the same background work as in the first example.

1.2.3 Example: In the code

Write a module:

```
-module(sort).

-export([do/1]).

do(N) ->
    cprof:stop(),
    cprof:start(),
    do(N, []).

do(0, L) ->
    R = lists:sort(L),
    cprof:pause(),
    R;
do(N, L) ->
    do(N-1, [random:uniform(256)-1 | L]).
```

From the Erlang shell:

```
1> c(sort).
{ok,sort}
2> l(random).
3> sort:do(1000).
[0,0,1,1,1,1,1,1,2,2,2,3,3,3,3,3,4,4,4,5,5,5,5,6,6,6,6,6,6|...]
4> cprof:analyse().
{9050,
 [{lists_sort,6047,
              [{{lists_sort,merge3_2,6},923},
               {{lists_sort,merge3_1,6},879},
               {{lists_sort,split_2,5},661},
               {{lists_sort,rmerge3_1,6},580},
               {{lists_sort,rmerge3_2,6},543},
               {{lists_sort,merge3_12_3,6},531},
               {{lists_sort,merge3_21_3,6},383},
               {{lists_sort,split_2_1,6},338},
               {{lists_sort,rmerge3_21_3,6},299},
               {{lists_sort,rmerge3_12_3,6},205},
               {{lists_sort,rmerge2_2,4},180},
               {{lists_sort,rmerge2_1,4},171},
               {{lists_sort,merge2_1,4},127},
               {{lists_sort,merge2_2,4},121},
               {{lists_sort,merge1,2},79},
               {{lists_sort,rmerge1,2},27}]},
  {random,2001,
          [{{random,uniform,1},1000},
           {{random,uniform,0},1000},
           {{random,seed0,0},1}]},
  {sort,1001,[{{sort,do,2},1001}]},
  {lists,1,[{{lists,sort,1},1}]}]}
5> cprof:stop().
5369
```

The example shows some details of how `lists:sort/1` works. It used 6047 function calls in the module `lists_sort` to complete the work.

This time, since the shell was not involved, no other work was done in the system during the profiling. If you retry the same example with a freshly started Erlang emulator, but omit the command `l(random)`, the analysis will show a lot more function calls done by `code_server` and others to automatically load the module `random`.

1.3 fprof - The File Trace Profiler

`fprof` is a profiling tool that can be used to get a picture of how much processing time different functions consumes and in which processes.

`fprof` uses tracing with timestamps to collect profiling data. Therfore there is no need for special compilation of any module to be profiled.

`fprof` presents wall clock times from the host machine OS, with the assumption that OS scheduling will randomly load the profiled functions in a fair way. Both *own time* i.e the time used by a function for its own execution, and *accumulated time* i.e execution time including called functions.

Profiling is essentially done in 3 steps:

1 Tracing; to file, as mentioned in the previous paragraph.

2 Profiling; the trace file is read and raw profile data is collected into an internal RAM storage on the node. During this step the trace data may be dumped in text format to file or console.

3 Analysing; the raw profile data is sorted and dumped in text format either to file or console.

Since `fprof` uses trace to file, the runtime performance degradation is minimized, but still far from negligible, especially not for programs that use the filesystem heavily by themselves. Where you place the trace file is also important, e.g on Solaris `/tmp` is usually a good choice, while any NFS mounted disk is a lousy choice.

Fprof can also skip the file step and trace to a tracer process of its own that does the profiling in runtime.

The following sections show some examples of how to profile with Fprof. See also the reference manual fprof(3) [page 38].

1.3.1 Profiling from the source code

If you can edit and recompile the source code, it is convenient to insert `fprof:trace(start)` and `fprof:trace(stop)` before and after the code to be profiled. All spawned processes are also traced. If you want some other filename than the default try `fprof:trace(start, "my_fprof.trace")`.

Then read the trace file and create the raw profile data with `fprof:profile()`, or perhaps `fprof:profile(file, "my_fprof.trace")` for non-default filename.

Finally create an informative table dumped on the console with `fprof:analyse()`, or on file with `fprof:analyse(dest, [])`, or perhaps even `fprof:analyse([{dest, "my_fprof.analysis"}, {cols, 120}])` for a wider listing on non-default filename.

See the fprof(3) [page 38] manual page for more options and arguments to the functions trace [page 40], profile [page 42] and analyse [page 43].

1.3.2 Profiling a function

If you have one function that does the task that you want to profile, and the function returns when the profiling should stop, it is convenient to use `fprof:apply(Module, Function, Args)` and related for the tracing step.

If the tracing should continue after the function returns, for example if it is a start function that spawns processes to be profiled, you can use `fprof:apply(M, F, Args, [continue | OtherOpts])`. The tracing has to be stopped at a suitable later time using `fprof:trace(stop)`.

1.3.3 Immediate profiling

It is also possible to trace immediately into the profiling process that creates the raw profile data, that is to short circuit the tracing and profiling steps so that the filesystem is not used.

Do something like this:

```
{ok, Tracer} = fprof:profile(start),
fprof:trace([start, {tracer, Tracer}]),
%% Code to profile
fprof:trace(stop);
```

This puts less load on the filesystem, but much more on the Erlang runtime system.

1.4 xref - The Cross Reference Tool

xref is a cross reference tool that can be used for finding dependencies between functions, modules, applications and releases. It does so by analyzing the defined functions and the function calls.

In order to make xref easy to use, there are predefined analyses that perform some common tasks. Typically, a module or a release can be checked for calls to undefined functions. For the somewhat more advanced user there is a small, but rather flexible, language that can be used for selecting parts of the analyzed system and for doing some simple graph analyses on selected calls.

The following sections show some features of xref, beginning with a module check and a predefined analysis. Then follow examples that can be skipped on the first reading; not all of the concepts used are explained, and it is assumed that the reference manual [page 58] has been at least skimmed.

1.4.1 Module Check

Assume we want to check the following module:

```
-module(my_module).

-export([t/1]).

t(A) ->
    my_module:t2(A).

t2(_) ->
    true.
```

Cross reference data are read from BEAM files, so the first step when checking an edited module is to compile it:

```
1> c(my_module, debug_info).
./my_module.erl:10: Warning: function t2/1 is unused
{ok, my_module}
```

The `debug_info` option ensures that the BEAM file contains debug information, which makes it possible to find unused local functions.

The module can now be checked for calls to undefined functions [page 59] and unused local functions:

```
2> xref:m(my_module)
[{undefined,[{{my_module,t,1},{my_module,t2,1}}]},
 {unused,[{my_module,t2,1}]}]
```

`m/1` is also suitable for checking that the BEAM file of a module that is about to be loaded into a running a system does not call any undefined functions. In either case, the code path of the code server (see the module `code`) is used for finding modules that export externally called functions not exported by the checked module itself, so called library modules [page 59].

1.4.2 Predefined Analysis

In the last example the module to analyze was given as an argument to `m/1`, and the code path was (implicitly) used as library path [page 59]. In this example an xref server [page 58] will be used, which makes it possible to analyze applications and releases, and also to select the library path explicitly.

Each xref server is referred to by a unique name. The name is given when creating the server:

```
1> xref:start(s).
{ok,<0.27.0>}
```

Next the system to be analyzed is added to the xref server. Here the system will be OTP, so no library path will be needed. Otherwise, when analyzing a system that uses OTP, the OTP modules are typically made library modules by setting the library path to the default OTP code path (or to `code_path`, see the reference manual [page 76]). By default, the names of read BEAM files and warnings are output when adding analyzed modules, but these messages can be avoided by setting default values of some options:

```
2> xref:set_default(s, [{verbose,false}, {warnings,false}]).
ok
3> xref:add_release(s, code:lib_dir(), {name, otp}).
{ok,otp}
```

`add_release/3` assumes that all subdirectories of the library directory returned by `code:lib_dir()` contain applications; the effect is that of reading all applications' BEAM files.

It is now easy to check the release for calls to undefined functions:

```
4> xref:analyze(s, undefined_function_calls).
{ok, [...]}
```

We can now continue with further analyses, or we can delete the xref server:

```
5> xref:stop(s).
```

The check for calls to undefined functions is an example of a predefined analysis, probably the most useful one. Other examples are the analyses that find unused local functions, or functions that call some given functions. See the analyze/2,3 [page 68] functions for a complete list of predefined analyses.

Each predefined analysis is a shorthand for a query [page 65], a sentence of a tiny language providing cross reference data as values of predefined variables [page 60]. The check for calls to undefined functions can thus be stated as a query:

```
4> xref:q(s, "(XC - UC) || (XU - X - B)").
{ok,[...]}
```

The query asks for the restriction of external calls except the unresolved calls to calls to functions that are externally used but neither exported nor built-in functions (the `||` operator restricts the used functions while the `|` operator restricts the calling functions). The `-` operator returns the difference of two sets, and the `+` operator to be used below returns the union of two sets.

The relationships between the predefined variables `XU`, `X`, `B` and a few others are worth elaborating upon. The reference manual mentions two ways of expressing the set of all functions, one that focuses on how they are defined: `X + L + B + U`, and one that focuses on how they are used: `UU + LU + XU`. The reference also mentions some facts [page 61] about the variables:

- `F` is equal to `L + X` (the defined functions are the local functions and the external functions);
- `U` is a subset of `XU` (the unknown functions are a subset of the externally used functions since the compiler ensures that locally used functions are defined);
- `B` is a subset of `XU` (calls to built-in functions are always external by definition, and unused built-in functions are ignored);
- `LU` is a subset of `F` (the locally used functions are either local functions or exported functions, again ensured by the compiler);
- `UU` is equal to `F - (XU + LU)` (the unused functions are defined functions that are neither used externally nor locally);
- `UU` is a subset of `F` (the unused functions are defined in analyzed modules).

Using these facts, the two small circles in the picture below can be combined.

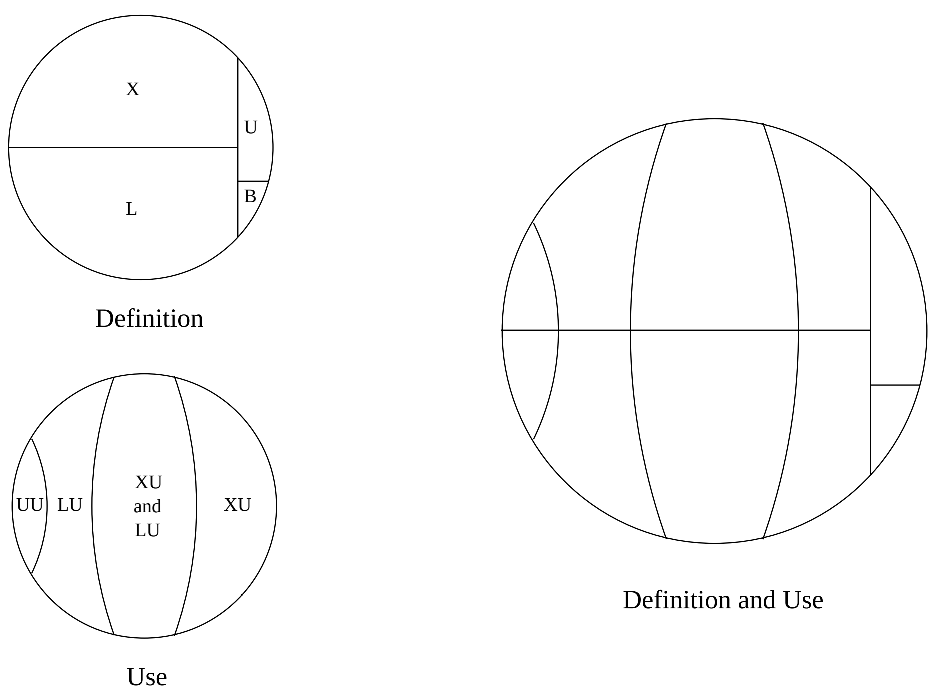

Figure 1.1: Definition and use of functions

It is often clarifying to mark the variables of a query in such a circle. This is illustrated in the picture below for some of the predefined analyses. Note that local functions used by local functions only are not marked in the `locals_not_used` circle.

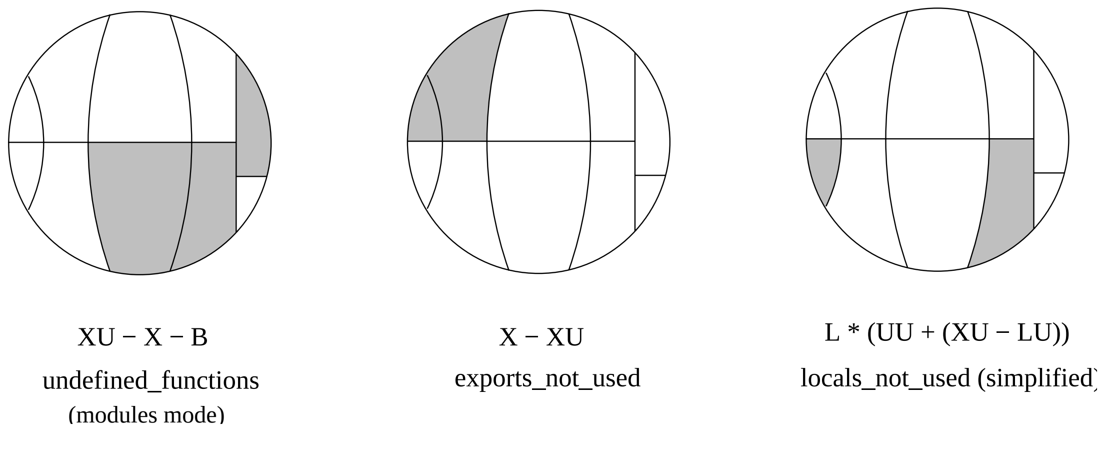

Figure 1.2: Some predefined analyses as subsets of all functions

1.4.3 Expressions

The module check and the predefined analyses are useful, but limited. Sometimes more flexibility is needed, for instance one might not need to apply a graph analysis on all calls, but some subset will do equally well. That flexibility is provided with a simple language. Below are some expressions of the language with comments, focusing on elements of the language rather than providing useful examples. The analyzed system is assumed to be OTP, so in order to run the queries, first evaluate these calls:

```
    xref:start(s).
    xref:add_release(s, code:root_dir()).
```

`xref:q(s, "(Fun) xref : Mod").` All functions of the `xref` module.

`xref:q(s, "xref : Mod * X").` All exported functions of the `xref` module. The first operand of the intersection operator `*` is implicitly converted to the more special type of the second operand.

`xref:q(s, "(Mod) tools").` All modules of the `tools` application.

`xref:q(s, '"xref_.*" : Mod').` All modules with a name beginning with `xref_`.

`xref:q(s, "# E | X ").` Number of calls from exported functions.

`xref:q(s, "XC || L ").` All external calls to local functions.

`xref:q(s, "XC * LC").` All calls that have both an external and a local version.

`xref:q(s, "(LLin) (LC * XC)").` The lines where the local calls of the last example are made.

`xref:q(s, "(XLin) (LC * XC)").` The lines where the external calls of the example before last are made.

`xref:q(s, "XC * (ME - strict ME)").` External calls within some module.

`xref:q(s, "E ||| kernel").` All calls within the `kernel` application.

`xref:q(s, "closure E | kernel || kernel").` All direct and indirect calls within the `kernel` application. Both the calling and the used functions of indirect calls are defined in modules of the kernel application, but it is possible that some functions outside the kernel application are used by indirect calls.

`xref:q(s, "{toolbar,debugger}:Mod of ME").` A chain of module calls from `toolbar` to `debugger`, if there is such a chain, otherwise `false`. The chain of calls is represented by a list of modules, `toolbar` being the first element and `debugger` the last element.

`xref:q(s, "closure E | toolbar:Mod || debugger:Mod").` All (in)direct calls from functions in `toolbar` to functions in `debugger`.

`xref:q(s, "(Fun) xref -> xref_base").` All function calls from `xref` to `xref_base`.

`xref:q(s, "E * xref -> xref_base").` Same interpretation as last expression.

`xref:q(s, "E || xref_base | xref").` Same interpretation as last expression.

`xref:q(s, "E * [xref -> lists, xref_base -> digraph]").` All function calls from `xref` to `lists`, and all function calls from `xref_base` to `digraph`.

`xref:q(s, "E | [xref, xref_base] || [lists, digraph]").` All function calls from `xref` and `xref_base` to `lists` and `digraph`.

`xref:q(s, "components EE").` All strongly connected components of the Inter Call Graph. Each component is a set of exported or unused local functions that call each other (in)directly.

`xref:q(s, "X * digraph * range (closure (E | digraph) | (L * digraph))").` All exported functions of the `digraph` module used (in)directly by some function in `digraph`.

`xref:q(s, "L * yeccparser:Mod - range (closure (E |`

`yeccparser:Mod) | (X * yeccparser:Mod))").` The interpretation is left as an exercise.

1.4.4 Graph Analysis

The list representation of graphs [page 59] is used analyzing direct calls, while the `digraph` representation is suited for analyzing indirect calls. The restriction operators (`|`, `||` and `|||`) are the only operators that accept both representations. This means that in order to analyze indirect calls using restriction, the `closure` operator (which creates the `digraph` representation of graphs) has to been applied explicitly.

As an example of analyzing indirect calls, the following Erlang function tries to answer the question: if we want to know which modules are used indirectly by some module(s), is it worth while using the function graph [page 59] rather than the module graph? Recall that a module M1 is said to call a module M2 if there is some function in M1 that calls some function in M2. It would be nice if we could use the much smaller module graph, since it is available also in the light weight `modules` mode [page 58] of xref servers.

```
t(S) ->
  {ok, _} = xref:q(S, "Eplus := closure E"),
  {ok, Ms} = xref:q(S, "AM"),
  Fun = fun(M, N) ->
            Q = io_lib:format("# (Mod) (Eplus | ~p : Mod)", [M]),
            {ok, N0} = xref:q(S, lists:flatten(Q)),
            N + N0
        end,
  Sum = lists:foldl(Fun, 0, Ms),
  ok = xref:forget(S, 'Eplus'),
  {ok, Tot} = xref:q(S, "# (closure ME | AM)"),
  100 * ((Tot - Sum) / Tot).
```

Comments on the code:

- We want to find the reduction of the closure of the function graph to modules. The direct expression for doing that would be `(Mod) (closure E | AM)`, but then we would have to represent all of the transitive closure of E in memory. Instead the number of indirectly used modules is found for each analyzed module, and the sum over all modules is calculated.
- A user variable is employed for holding the `digraph` representation of the function graph for use in many queries. The reason is efficiency. As opposed to the `=` operator, the `:=` operator saves a value for subsequent analyses. Here might be the place to note that equal subexpressions within a query are evaluated only once; `=` cannot be used for speeding things up.
- `Eplus | ~p : Mod`. The `|` operator converts the second operand to the type of the first operand. In this case the module is converted to all functions of the module. It is necessary to assign a type to the module (`: Mod`), otherwise modules like `kernel` would be converted to all functions of the application with the same name; the most general constant is used in cases of ambiguity.
- Since we are only interested in a ratio, the unary operator `#` that counts the elements of the operand is used. It cannot be applied to the `digraph` representation of graphs.
- We could find the size of the closure of the module graph with a loop similar to one used for the function graph, but since the module graph is so much smaller, a more direct method is feasible.

When the Erlang function `t/1` was applied to an xref server loaded with the current version of OTP, the returned value was close to 84 (percent). This means that the number of indirectly used modules is approximately six times greater when using the module graph. So the answer to the above stated question is that it is definitely worth while using the function graph for this particular analysis. Finally, note that in the presence of unresolved calls, the graphs may be incomplete, which means that there may be indirectly used modules that do not show up.

Tools Reference Manual

Short Summaries

cover

The following functions are exported:

- `analyse(Module) -> {ok,Answer} | {error,Error}`
  [page 30] Analyse a Cover compiled module.
- `analyse(Module, Analysis) -> {ok,Answer} | {error,Error}`
  [page 30] Analyse a Cover compiled module.
- `analyse(Module, Level) -> {ok,Answer} | {error,Error}`
  [page 30] Analyse a Cover compiled module.
- `analyse(Module, Analysis, Level) -> {ok,Answer} | {error,Error}`
  [page 30] Analyse a Cover compiled module.
- `analyse_to_file(Module) ->`
  [page 30] Detailed coverage analysis of a Cover compiled module.
- `analyse_to_file(Module, OutFile) -> {ok,OutFile} | {error,Error}`
  [page 30] Detailed coverage analysis of a Cover compiled module.
- `modules() -> [Module]`
  [page 31] Return all Cover compiled modules.
- `is_compiled(Module) -> {file,File} | false`
  [page 31] Check if a module is Cover compiled.
- `reset(Module) -> ok`
  [page 31] Reset coverage data for Cover compiled modules.
- `reset() -> ok`
  [page 31] Reset coverage data for Cover compiled modules.
- `stop() -> ok`
  [page 31] Stop Cover.

cprof

The following functions are exported:

- `analyse() -> {AllCallCount, ModAnalysisList}`
  [page 32] Collect and analyse call counters.
- `analyse(Limit) -> {AllCallCount, ModAnalysisList}`
  [page 32] Collect and analyse call counters.
- `analyse(Mod) -> ModAnlysis`
  [page 32] Collect and analyse call counters.
- `analyse(Mod, Limit) -> ModAnalysis`
  [page 32] Collect and analyse call counters.
- `pause() -> integer()`
  [page 33] Pause running call count trace for all functions.
- `pause(FuncSpec) -> integer()`
  [page 33] Pause running call count trace for matching functions.
- `pause(Mod, Func) -> integer()`
  [page 33] Pause running call count trace for matching functions.
- `pause(Mod, Func, Arity) -> integer()`
  [page 33] Pause running call count trace for matching functions.
- `restart() -> integer()`
  [page 33] Restart existing call counters for matching functions.
- `restart(FuncSpec) -> integer()`
  [page 33] Restart existing call counters for matching functions.

- `restart(Mod, Func) -> integer()`
  [page 33] Restart existing call counters for matching functions.
- `restart(Mod, Func, Arity) -> integer()`
  [page 33] Restart existing call counters for matching functions.
- `start() -> integer()`
  [page 34] Start call count tracing for all functions.
- `start(FuncSpec) -> integer()`
  [page 34] Start call count tracing for matching functions.
- `start(Mod, Func) -> integer()`
  [page 34] Start call count tracing for matching functions.
- `start(Mod, Func, Arity) -> integer()`
  [page 34] Start call count tracing for matching functions.
- `stop() -> integer()`
  [page 34] Stop call count tracing for all functions.
- `stop(FuncSpec) -> integer()`
  [page 34] Stop call count tracing for matching functions.
- `stop(Mod, Func) -> integer()`
  [page 34] Stop call count tracing for matching functions.
- `stop(Mod, Func, Arity) -> integer()`
  [page 34] Stop call count tracing for matching functions.

eprof

The following functions are exported:

- `start() -> {ok,Pid} | {error,Reason}`
  [page 36] Start Eprof.
- `start_profiling(Rootset) -> profiling | error`
  [page 36] Start profiling.
- `profile(Rootset) -> profiling | error`
  [page 36] Start profiling.
- `stop_profiling() -> profiling_stopped | profiling_already_stopped`
  [page 36] Stop profiling.
- `profile(Rootset,Fun) -> {ok,Value} | {error,Reason} | error`
  [page 36] Start profiling.
- `profile(Rootset,Module,Function,Args) -> {ok,Value} | {error,Reason} | error`
  [page 36] Start profiling.
- `analyse()`
  [page 37] Display profiling results per process.
- `total_analyse()`
  [page 37] Display profiling results per function call.
- `log(File) -> ok`
  [page 37] Activate logging of `eprof` printouts.
- `stop() -> stopped`
  [page 37] Stop Eprof.

fprof

The following functions are exported:

- `start() -> {ok, Pid} | {error, {already_started, Pid}}`
  [page 39] Starts the `fprof` server.
- `stop() -> ok`
  [page 39] Same as `stop(normal)`.
- `stop(Reason) -> ok`
  [page 39] Stops the `fprof` server.
- `apply(Func, Args) -> term()`
  [page 39] Same as `apply(Func, Args, [])`.
- `apply(Module, Function, Args) -> term()`
  [page 39] Same as `apply({Module, Function}, Args, [])`.
- `apply(Func, Args, OptionList) -> term()`
  [page 39] Calls `erlang:apply(Func, Args)` surrounded by `trace([start | OptionList])` and `trace(stop)`.
- `apply(Module, Function, Args, OptionList) -> term()`
  [page 40] Same as `apply({Module, Function}, Args, OptionList)`.
- `trace(start, Filename) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 40] Same as `trace([start, {file, Filename}])`.
- `trace(verbose, Filename) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 40] Same as `trace([start, verbose, {file, Filename}])`.
- `trace(OptionName, OptionValue) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 40] Same as `trace([{OptionName, OptionValue}])`.
- `trace(verbose) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 41] Same as `trace([start, verbose])`.
- `trace(OptionName) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 41] Same as `trace([OptionName])`.
- `trace({OptionName, OptionValue}) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 40] Same as `trace([{OptionName, OptionValue}])`.
- `trace([Option]) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 41] Starts or stops tracing.
- `profile() -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 42] Same as `profile([])`.
- `profile(OptionName, OptionValue) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 42] Same as `profile([{OptionName, OptionValue}])`.
- `profile(OptionName) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 42] Same as `profile([OptionName])`.

- `profile({OptionName, OptionValue}) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 42] Same as `profile([{OptionName, OptionValue}])`.
- `profile([Option]) -> ok | {ok, Tracer} | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 42] Compiles a trace into raw profile data held by the `fprof` server.
- `analyse() -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 43] Same as `analyse([])`.
- `analyse(OptionName, OptionValue) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 43] Same as `analyse([{OptionName, OptionValue}])`.
- `analyse(OptionName) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 43] Same as `analyse([OptionName])`.
- `analyse({OptionName, OptionValue}) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 43] Same as `analyse([{OptionName, OptionValue}])`.
- `analyse([Option]) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`
  [page 43] Analyses raw profile data in the `fprof` server.

instrument

The following functions are exported:

- `holes(AllocList) -> ok`
  [page 52] Print out the sizes of unused memory blocks
- `mem_limits(AllocList) -> {Low, High}`
  [page 52] Return lowest and highest memory address used
- `memory_data() -> AllocList`
  [page 52] Return current memory allocation list
- `read_memory_data(File) -> {ok, AllocList} | {error, Reason}`
  [page 52] Read memory allocation list
- `sort(AllocList) -> AllocList`
  [page 52] Sort a memory allocation list
- `store_memory_data(File) -> ok`
  [page 53] Store the current memory allocation list on a file
- `sum_blocks(AllocList) -> int()`
  [page 53] Return the total amount of memory used
- `type_string(Type) -> string()`
  [page 53] Translate a memory block type number to a string

make

The following functions are exported:

- `all() -> up_to_date | error`
  [page 54] Compile a set of modules.

- `all(Options) -> up_to_date | error`
  [page 54] Compile a set of modules.
- `files(ModFiles) -> up_to_date | error`
  [page 54] Compile a set of modules.
- `files(ModFiles, Options) -> up_to_date | error`
  [page 55] Compile a set of modules.

tags

The following functions are exported:

- `file(File [, Options])`
  [page 56] Create a `TAGS` file for the file `File`.
- `files(FileList [, Options])`
  [page 56] Create a TAGS file for the files in the list `FileList`.
- `dir(Dir [, Options])`
  [page 56] Create a TAGS file for all files in directory `Dir`.
- `dirs(DirList [, Options])`
  [page 56] Create a TAGS file for all files in any directory in `DirList`.
- `subdir(Dir [, Options])`
  [page 56] Descend recursively down the directory `Dir` and create a `TAGS` file based on all files found.
- `subdirs(DirList [, Options])`
  [page 56] Descend recursively down all the directories in `DirList` and create a `TAGS` file based on all files found.
- `root([Options])`
  [page 56] Create a `TAGS` file covering all files in the Erlang distribution.

xref

The following functions are exported:

- `add_application(Xref, Directory [, Options]) -> {ok, application()} | Error`
  [page 66] Add the modules of an application.
- `add_directory(Xref, Directory [, Options]) -> {ok, Modules} | Error`
  [page 66] Add the modules in a directory.
- `add_module(Xref, File [, Options]) -> {ok, module()} | Error`
  [page 67] Add a module.
- `add_release(Xref, Directory [, Options]) -> {ok, release()} | Error`
  [page 67] Add the modules of a release.
- `analyze(Xref, Analysis [, Options]) -> {ok, Answer} | Error`
  [page 68] Evaluate a predefined analysis.
- `d(Directory) -> [DebugInfoResult] | [NoDebugInfoResult] | Error`
  [page 69] Check the modules in a directory using the code path.
- `forget(Xref) -> ok`
  [page 69] Remove user variables and their values.

- forget(Xref, Variables) -> ok | Error
  [page 69] Remove user variables and their values.
- format_error(Error) -> character_list()
  [page 69] Return an English description of an Xref error reply.
- get_default(Xref) -> [{Option, Value}]
  [page 70] Return the default values of options.
- get_default(Xref, Option) -> {ok, Value} | Error
  [page 70] Return the default values of options.
- get_library_path(Xref) -> {ok, LibraryPath}
  [page 70] Return the library path.
- info(Xref) -> [Info]
  [page 70] Return information about an Xref server.
- info(Xref, Category) -> [{Item, [Info]}]
  [page 70] Return information about an Xref server.
- info(Xref, Category, Items) -> [{Item, [Info]}]
  [page 70] Return information about an Xref server.
- m(Module) -> [DebugInfoResult] | [NoDebugInfoResult] | Error
  [page 73] Check a module using the code path.
- m(File) -> [DebugInfoResult] | [NoDebugInfoResult] | Error
  [page 73] Check a module using the code path.
- q(Xref, Query [, Options]) -> {ok, Answer} | Error
  [page 73] Evaluate a query.
- remove_application(Xref, Applications) -> ok | Error
  [page 74] Remove applications and their modules.
- remove_module(Xref, Modules) -> ok | Error
  [page 74] Remove analyzed modules.
- remove_release(Xref, Releases) -> ok | Error
  [page 75] Remove releases and their applications and modules.
- replace_application(Xref, Application, Directory [, Options]) -> {ok, application()} | Error
  [page 75] Replace an application's modules.
- replace_module(Xref, Module, File [, Options]) -> {ok, module()} | Error
  [page 75] Replace an analyzed module.
- set_default(Xref, Option, Value) -> {ok, OldValue} | Error
  [page 75] Set the default values of options.
- set_default(Xref, OptionValues) -> ok | Error
  [page 75] Set the default values of options.
- set_library_path(Xref, LibraryPath [, Options]) -> ok | Error
  [page 76] Set the library path and finds the library modules.
- start(Xref [, Options]) -> Return
  [page 76] Create an xref server.
- stop(Xref)
  [page 77] Delete an xref server.
- update(Xref [, Options]) -> {ok, Modules} | Error
  [page 77] Replace newly compiled analyzed modules.
- variables(Xref [, Options]) -> {ok, [VariableInfo]}
  [page 77] Return the names of variables.

cover

Erlang Module

The module `cover` provides a set of functions for coverage analysis of Erlang programs, counting how many times each *executable line* of code is executed when a program is run.
An executable line contains an Erlang expression such as a matching or a function call. A blank line or a line containing a comment, function head or pattern in a `case`- or `receive` statement is not executable.

Coverage analysis can be used to verify test cases, making sure all relevant code is covered, and may also be helpful when looking for bottlenecks in the code.

Before any analysis can take place, the involved modules must be *Cover compiled*. This means that some extra information is added to the module before it is compiled into a binary which then is loaded. The source file of the module is not affected and no `.beam` file is created.

Each time a function in a Cover compiled module is called, information about the call is added to an internal database of Cover. The coverage analysis is performed by examining the contents of the Cover database. The output `Answer` is determined by two parameters, `Level` and `Analysis`.

- `Level = module`

  `Answer = {Module,Value}`, where `Module` is the module name.
- `Level = function`

  `Answer = [{Function,Value}]`, one tuple for each function in the module. A function is specified by its module name `M`, function name `F` and arity `A` as a tuple `{M,F,A}`.
- `Level = clause`

  `Answer = [{Clause,Value}]`, one tuple for each clause in the module. A clause is specified by its module name `M`, function name `F`, arity `A` and position in the function definition `C` as a tuple `{M,F,A,C}`.
- `Level = line`

  `Answer = [{Line,Value}]`, one tuple for each executable line in the module. A line is specified by its module name `M` and line number in the source file `N` as a tuple `{M,N}`.
- `Analysis = coverage`

  `Value = {Cov,NotCov}` where `Cov` is the number of executable lines in the module, function, clause or line that have been executed at least once and `NotCov` is the number of executable lines that have not been executed.
- `Analysis = calls`

  `Value = Calls` which is the number of times the module, function, or clause has been called. In the case of line level analysis, `Calls` is the number of times the line has been executed.

Exports

`start() -> {ok,Pid} | {error,Reason}`

Types:

- Pid = pid()
- Reason = {already_started,Pid}

Starts the Cover server which owns the Cover internal database. This function is called automatically by the other functions in the module.

`compile(ModFile) -> Result`
`compile(ModFile, Options) -> Result`
`compile_module(ModFile) -> Result`
`compile_module(ModFile, Options) -> Result`

Types:

- ModFile = Module | File
- Module = atom()
- File = string()
- Options = [Option]
- Option = {i,Dir} | {d,Macro} | {d,Macro,Value}
  See `compile:file/2`.
- Result = {ok,Module} | {error,File}

Compiles a module for Cover analysis. The module is given by its module name `Module` or by its file name `File`. The `.erl` extension may be omitted. If the module is located in another directory, the path has to be specified.

`Options` is a list of compiler options which defaults to `[]`. Only options defining include file directories and macros are passed to `compile:file/2`, everything else is ignored.

If the module is successfully Cover compiled, the function returns `{ok,Module}`. Otherwise the function returns `{error,File}`. Errors and warnings are printed as they occur.

Note that the internal database is (re-)initiated during the compilation, meaning any previously collected coverage data for the module will be lost.

`compile_directory() -> [Result] | {error,Reason}`
`compile_directory(Dir) -> [Result] | {error,Reason}`
`compile_directory(Dir, Options) -> [Result] | {error,Reason}`

Types:

- Dir = string()
- Options = [Option]
  See `compile_module/1,2`
- Result = {ok,Module} | {error,File}
  See `compile_module/1,2`
- Reason = eacces | enoent

Compiles all modules (`.erl` files) in a directory `Dir` for Cover analysis the same way as `compile_module/1,2` and returns a list with the return values.

`Dir` defaults to the current working directory.

The function returns `{error,eacces}` if the directory is not readable or `{error,enoent}` if the directory does not exist.

```
analyse(Module) -> {ok,Answer} | {error,Error}
analyse(Module, Analysis) -> {ok,Answer} | {error,Error}
analyse(Module, Level) -> {ok,Answer} | {error,Error}
analyse(Module, Analysis, Level) -> {ok,Answer} | {error,Error}
```

Types:

- Module = atom()
- Analysis = coverage | calls
- Level = line | clause | function | module
- Answer = {Module,Value} | [{Item,Value}]
- Item = Line | Clause | Function
- Line = {M,N}
- Clause = {M,F,A,C}
- Function = {M,F,A}
- M = F = atom()
- N = A = C = integer()
- Value = {Cov,NotCov} | Calls
- Cov = NotCov = Calls = integer()
- Error = {not_cover_compiled,Module}

Performs analysis of a Cover compiled module `Module`, as specified by `Analysis` and `Level` (see above), by examining the contents of the internal database.

`Analysis` defaults to `coverage` and `Level` defaults to `function`.

If `Module` is not Cover compiled, the function returns `{error,{not_cover_compiled,Module}}`.

```
analyse_to_file(Module) ->
analyse_to_file(Module, OutFile) -> {ok,OutFile} | {error,Error}
```

Types:

- Module = atom()
- OutFile = string()
- Error = {not_cover_compiled,Module} | {file,File,Reason}
- File = string()
- Reason = term()

Makes a copy `OutFile` of the source file for a module `Module`, where it for each executable line is specified how many times it has been executed.

The output file `OutFile` defaults to `Module.COVER.out`.

If `Module` is not Cover compiled, the function returns `{error,{not_cover_compiled,Module}}`.

If the source file and/or the output file cannot be opened using `file:open/2`, the function returns `{error,{file,File,Reason}}` where `File` is the file name and `Reason` is the error reason.

`modules() -> [Module]`

Types:

- Module = atom()

Returns a list with all modules that are currently Cover compiled.

`is_compiled(Module) -> {file,File} | false`

Types:

- Module = atom()
- Beam = string()

Returns `{file,File}` if the module `Module` is Cover compiled, or `false` otherwise. `File` is the `.erl` file used by `cover:compile_module/1,2`.

`reset(Module) -> ok`
`reset() -> ok`

Types:

- Module = atom()

Resets all coverage data for a Cover compiled module `Module` in the Cover database. If the argument is omitted, the coverage data will be reset for all modules known by Cover.

If `Module` is not Cover compiled, the function returns `{error,{not_cover_compiled,Module}}`.

`stop() -> ok`

Stops the Cover server and unloads all Cover compiled code.

SEE ALSO

code(3), compile(3)

cprof

Erlang Module

The `cprof` module is used to profile a program to find out how many times different functions are called. Breakpoints similar to local call trace, but containing a counter, are used to minimise runtime performance impact.

Since breakpoints are used there is no need for special compilation of any module to be profiled. For now these breakpoints can only be set on BEAM code so *BIF* s cannot be call count traced.

The size of the call counters is the host machine word size. One bit is used when pausing the counter, so the maximum counter value for a 32-bit host is 2147483647.

The profiling result is delivered as a term containing a sorted list of entries, one per module. Each module entry contains a sorted list of functions. The sorting order in both cases is of decreasing call count.

Call count tracing is very lightweight compared to other forms of tracing since no trace message has to be generated. Some measurements indicates performance degradation in the vicinity of 10 percent.

Exports

```
analyse() -> {AllCallCount, ModAnalysisList}
analyse(Limit) -> {AllCallCount, ModAnalysisList}
analyse(Mod) -> ModAnlysis
analyse(Mod, Limit) -> ModAnalysis
```

Types:

- Limit = integer()
- Mod = atom()
- AllCallCount = integer()
- ModAnalysisList = [ModAnalysis]
- ModAnalysis = {Mod, ModCallCount, FuncAnalysisList}
- ModCallCount = integer()
- FuncAnalysisList = [{{Mod, Func, Arity}, FuncCallCount}]
- Func = atom()
- Arity = integer()
- FuncCallCount = integer()

Collects and analyses the call counters presently in the node for either module `Mod`, or for all modules (except `cprof` itself), and returns:

`FuncAnalysisList` A list of tuples, one for each function in a module, in decreasing `FuncCallCount` order.

`ModCallCount` The sum of `FuncCallCount` values for all functions in module `Mod`.

`AllCallCount` The sum of `ModCallCount` values for all modules concerned in `ModAnalysisList`.

`ModAnalysisList` A list of tuples, one for each module except `cprof`, in decreasing `ModCallCount` order.

If call counters are still running while `analyse/0..2` is executing, you might get an inconsistent result. This happens if the process executing `analyse/0..2` gets scheduled out so some other process can increment the counters that are being analysed, Calling `pause()` before analysing takes care of the problem.

If the `Mod` argument is given, the result contains a `ModAnalysis` tuple for module `Mod` only, otherwise the result contains one `ModAnalysis` tuple for all modules returned from `code:all_loaded()` except `cprof` itself.

All functions with a `FuncCallCount` lower than `Limit` are excluded from `FuncAnalysisList`. They are still included in `ModCallCount`, though. The default value for `Limit` is `1`.

`pause() -> integer()`

Pause call count tracing for all functions in all modules and stop it for all functions in modules to be loaded. This is the same as `(pause({'_','_','_'})+stop({on_load}))`.

See also pause/1..3 [page 33] below.

`pause(FuncSpec) -> integer()`
`pause(Mod, Func) -> integer()`
`pause(Mod, Func, Arity) -> integer()`

Types:

- FuncSpec = Mod | {Mod,Func,Arity}, {FS}
- Mod = atom()
- Func = atom()
- Arity = integer()
- FS = term()

Pause call counters for matching functions in matching modules. The `FS` argument can be used to specify the first argument to `erlang:trace_pattern/3`. See erlang(3).

The call counters for all matching functions that has got call count breakpoints are paused at their current count.

Return the number of matching functions that can have call count breakpoints, the same as `start/0..3` with the same arguments would have returned.

`restart() -> integer()`
`restart(FuncSpec) -> integer()`
`restart(Mod, Func) -> integer()`
`restart(Mod, Func, Arity) -> integer()`

Types:

- FuncSpec = Mod | {Mod,Func,Arity}, {FS}
- Mod = atom()
- Func = atom()

- Arity = integer()
- FS = term()

Restart call counters for the matching functions in matching modules that are call count traced. The `FS` argument can be used to specify the first argument to `erlang:trace_pattern/3`. See erlang(3).

The call counters for all matching functions that has got call count breakpoints are set to zero and running.

Return the number of matching functions that can have call count breakpoints, the same as `start/0..3` with the same arguments would have returned.

`start() -> integer()`

Start call count tracing for all functions in all modules, and also for all functions in modules to be loaded. This is the same as `(start({'_','_','_'})+start({on_load}))`.

See also start/1..3 [page 34] below.

`start(FuncSpec) -> integer()`
`start(Mod, Func) -> integer()`
`start(Mod, Func, Arity) -> integer()`

Types:

- FuncSpec = Mod | {Mod,Func,Arity}, {FS}
- Mod = atom()
- Func = atom()
- Arity = integer()
- FS = term()

Start call count tracing for matching functions in matching modules. The `FS` argument can be used to specify the first argument to `erlang:trace_pattern/3`, for example `on_load`. See erlang(3).

Set call count breakpoints on the matching functions that has no call count breakpoints. Call counters are set to zero and running for all matching functions.

Return the number of matching functions that has got call count breakpoints.

`stop() -> integer()`

Stop call count tracing for all functions in all modules, and also for all functions in modules to be loaded. This is the same as `(stop({'_','_','_'})+stop({on_load}))`.

See also stop/1..3 [page 34] below.

`stop(FuncSpec) -> integer()`
`stop(Mod, Func) -> integer()`
`stop(Mod, Func, Arity) -> integer()`

Types:

- FuncSpec = Mod | {Mod,Func,Arity}, {FS}
- Mod = atom()
- Func = atom()
- Arity = integer()

- FS = term()

Stop call count tracing for matching functions in matching modules. The `FS` argument can be used to specify the first argument to `erlang:trace_pattern/3`, for example `on_load`. See erlang(3).

Remove call count breakpoints from the matching functions that has call count breakpoints.

Return the number of matching functions that can have call count breakpoints, the same as `start/0..3` with the same arguments would have returned.

See Also

eprof [page 36](3), fprof [page 38](3), erlang(3), User's Guide [page 10]

eprof

Erlang Module

The module `eprof` provides a set of functions for time profiling of Erlang programs to find out how the execution time is used. The profiling is done using the Erlang trace BIFs. Tracing of local function calls for a specfied set of processes is enabled when profiling is begun, and disabled when profiling is stopped.

When using Eprof, expect a significant slowdown in program execution, in most cases at least 100 percent.

Exports

`start() -> {ok,Pid} | {error,Reason}`

Types:

- Pid = pid()
- Reason = {already_started,Pid}

Starts the Eprof server which owns the Eprof internal database.

`start_profiling(Rootset) -> profiling | error`
`profile(Rootset) -> profiling | error`

Types:

- Rootset = [atom() | pid()]

Starts profiling for the processes in `Rootset` (and any new processes spawned from them). Information about activity in any profiled process is stored in the Eprof database.

`Rootset` is a list of pids and registered names.

The function returns `profiling` if tracing could be enabled for all processes in `Rootset`, or `error` otherwise.

`stop_profiling() -> profiling_stopped | profiling_already_stopped`

Stops profiling started with `start_profiling/1` or `profile/1`.

`profile(Rootset,Fun) -> {ok,Value} | {error,Reason} | error`
`profile(Rootset,Module,Function,Args) -> {ok,Value} | {error,Reason} | error`

Types:

- Rootset = [atom() | pid()]
- Fun = fun() -> term()
- Module = Function = atom()

- Args = [term()]
- Value = Reason = term()

This function first spawns a process `P` which evaluates `Fun()` or `apply(Module,Function,Args)`. Then, it starts profiling for `P` and the processes in `Rootset` (and any new processes spawned from them). Information about activity in any profiled process is stored in the Eprof database.

`Rootset` is a list of pids and registered names.

If tracing could be enabled for `P` and all processes in `Rootset`, the function returns {`ok,Value`} when `Fun()/apply` returns with the value `Value`, or {`error,Reason`} if `Fun()/apply` fails with exit reason `Reason`. Otherwise it returns `error` immediately.

The programmer must ensure that the function given as argument is truly synchronous and that no work continues after the function has returned a value.

`analyse()`

Call this function when profiling has been stopped to display the results per process, that is:

- how much time has been used by each process, and
- in which function calls this time has been spent.

Time is shown as percentage of total time, not as absolute time.

`total_analyse()`

Call this function when profiling has been stopped to display the results per function call, that is in which function calls the time has been spent.

Time is shown as percentage of total time, not as absolute time.

`log(File) -> ok`

Types:

- File = atom() | string()

This function ensures that the results displayed by `analyse/0` and `total_analyse/0` are printed both to the file `File` and the screen.

`stop() -> stopped`

Stops the Eprof server.

fprof

Erlang Module

This module is used to profile a program to find out how the execution time is used. Trace to file is used to minimize runtime performance impact.

The `fprof` module uses tracing to collect profiling data, hence there is no need for special compilation of any module to be profiled. When it starts tracing, `fprof` will erase all previous tracing in the node and set the necessary trace flags on the profiling target processes as well as local call trace on all functions in all loaded modules and all modules to be loaded. `fprof` erases all tracing in the node when it stops tracing.

`fprof` presents both *own time* i.e how much time a function has used for its own execution, and *accumulated time* i.e including called functions. All presented times are collected using trace timestamps. `fprof` tries to collect cpu time timestamps, if the host machine OS supports it. Therefore the times may be wallclock times and OS scheduling will randomly strike all called functions in a presumably fair way.

If, however, the profiling time is short, and the host machine OS does not support high resolution cpu time measurements, some few OS schedulings may show up as ridicously long execution times for functions doing practically nothing. An example of a function more or less just composing a tuple in about 100 times the normal execution time has been seen, and when the tracing was repeated, the execution time became normal.

Profiling is essentially done in 3 steps:

1. Tracing; to file, as mentioned in the previous paragraph. The trace contains entries for function calls, returns to function, process scheduling, other process related (spawn, etc) events, and garbage collection. All trace entries are timestamped.
2. Profiling; the trace file is read, the execution call stack is simulated, and raw profile data is calculated from the simulated call stack and the trace timestamps. The profile data is stored in the `fprof` server state. During this step the trace data may be dumped in text format to file or console.
3. Analysing; the raw profile data is sorted, filtered and dumped in text format either to file or console. The text format intended to be both readable for a human reader, as well as parsable with the standard erlang parsing tools.

Since `fprof` uses trace to file, the runtime performance degradation is minimized, but still far from negligible, especially for programs that use the filesystem heavily by themselves. Where you place the trace file is also important, e.g on Solaris `/tmp` is usually a good choice since it is essentially a RAM disk, while any NFS (network) mounted disk is a bad idea.

`fprof` can also skip the file step and trace to a tracer process that does the profiling in runtime.

Exports

`start() -> {ok, Pid} | {error, {already_started, Pid}}`

Types:

- Pid = pid()

Starts the `fprof` server.

Note that it seldom needs to be started explicitly since it is automatically started by the functions that need a running server.

`stop() -> ok`

Same as `stop(normal)`.

`stop(Reason) -> ok`

Types:

- Reason = term()

Stops the `fprof` server.

The supplied `Reason` becomes the exit reason for the server process. Default Any `Reason` other than `kill` sends a request to the server and waits for it to clean up, reply and exit. If `Reason` is `kill`, the server is bluntly killed.

If the `fprof` server is not running, this function returns immediately with the same return value.

Note:
When the `fprof` server is stopped the collected raw profile data is lost.

`apply(Func, Args) -> term()`

Types:

- Func = function() | {Module, Function}
- Args = [term()]
- Module = atom()
- Function = atom()

Same as `apply(Func, Args, [])`.

`apply(Module, Function, Args) -> term()`

Types:

- Args = [term()]
- Module = atom()
- Function = atom()

Same as `apply({Module, Function}, Args, [])`.

`apply(Func, Args, OptionList) -> term()`

Types:

- Func = function() | {Module, Function}
- Args = [term()]
- OptionList = [Option]
- Module = atom()
- Function = atom()
- Option = continue | start | {procs, PidList} | TraceStartOption

Calls `erlang:apply(Func, Args)` surrounded by `trace([start, ...])` and `trace(stop)`.

Some effort is made to keep the trace clean from unnecessary trace messages; tracing is started and stopped from a spawned process while the `erlang:apply/2` call is made in the current process, only surrounded by `receive` and `send` statements towards the trace starting process. The trace starting process exits when not needed any more.

The `TraceStartOption` is any option allowed for `trace/1`. The options `[start, {procs, [self() | PidList]} | OptList]` are given to `trace/1`, where `OptList` is `OptionList` with `continue`, `start` and `{procs, _}` options removed.

The `continue` option inhibits the call to `trace(stop)` and leaves it up to the caller to stop tracing at a suitable time.

`apply(Module, Function, Args, OptionList) -> term()`

Types:

- Module = atom()
- Function = atom()
- Args = [term()]

Same as `apply({Module, Function}, Args, OptionList)`.

`OptionList` is an option list allowed for `apply/3`.

`trace(start, Filename) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- Reason = term()

Same as `trace([start, {file, Filename}])`.

`trace(verbose, Filename) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- Reason = term()

Same as `trace([start, verbose, {file, Filename}])`.

`trace(OptionName, OptionValue) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- OptionValue = term()
- Reason = term()

Same as `trace([{OptionName, OptionValue}])`.

`trace(verbose) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- Reason = term()

Same as `trace([start, verbose])`.

`trace(OptionName) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- Reason = term()

Same as `trace([OptionName])`.

`trace({OptionName, OptionValue}) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- OptionValue = term()
- Reason = term()

Same as `trace([{OptionName, OptionValue}])`.

`trace([Option]) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- Option = start | stop | {procs, PidSpec} | {procs, [PidSpec]} | verbose | {verbose, bool()} | file | {file, Filename} | {tracer, Tracer}
- PidSpec = pid() | atom()
- Tracer = pid() | port()
- Reason = term()

Starts or stops tracing.

`PidSpec` and `Tracer` are used in calls to `erlang:trace(PidSpec, true, [{tracer, Tracer} | Flags])`, and `Filename` is used to call `dbg:trace_port(file, Filename)`. Please see the appropriate documentation.

Option description:

`stop` Stops a running `fprof` trace and clears all tracing from the node. Either option `stop` or `start` must be specified, but not both.

`start` Clears all tracing from the node and starts a new `fprof` trace. Either option `start` or `stop` must be specified, but not both.

`verbose | {verbose, bool()}` The options `verbose` or `{verbose, true}` adds some trace flags that `fprof` does not need, but that may be interesting for general debugging purposes. This option is only allowed with the `start` option.

`cpu_time | {cpu_time, bool()}` The options `cpu_time` or `{cpu_time, true>` makes the timestamps in the trace be in CPU time instead of wallclock time which is the default. This option is only allowed with the `start` option.

`{procs, PidSpec}` | `{procs, [PidSpec]}` Specifies which processes that shall be traced. If this option is not given, the calling process is traced. All processes spawned by the traced processes are also traced. This option is only allowed with the `start` option.

`file` | `{file, Filename}` Specifies the filename of the trace. If the option `file` is given, or none of these options are given, the file `"fprof.trace"` is used. This option is only allowed with the `start` option, but not with the `{tracer, Tracer}` option.

`{tracer, Tracer}` Specifies that trace to process or port shall be done instead of trace to file. This option is only allowed with the `start` option, but not with the `{file, Filename}` option.

`profile() -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- Reason = term()

Same as `profile([])`.

`profile(OptionName, OptionValue) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- OptionValue = term()
- Reason = term()

Same as `profile([{OptionName, OptionValue}])`.

`profile(OptionName) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- Reason = term()

Same as `profile([OptionName])`.

`profile({OptionName, OptionValue}) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- OptionValue = term()
- Reason = term()

Same as `profile([{OptionName, OptionValue}])`.

`profile([Option]) -> ok | {ok, Tracer} | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- Option = file | {file, Filename} | dump | {dump, Dump} | append | start | stop
- Dump = pid() | Dumpfile | []

- Tracer = pid()
- Reason = term()

Compiles a trace into raw profile data held by the `fprof` server.

`Dumpfile` is used to call `file:open/2`, and `Filename` is used to call `dbg:trace_port(file, Filename)`. Please see the appropriate documentation.

Option description:

`file | {file, Filename}` Reads the file `Filename` and creates raw profile data that is stored in RAM by the `fprof` server. If the option `file` is given, or none of these options are given, the file `"fprof.trace"` is read. The call will return when the whole trace has been read with the return value `ok` if successful. This option is not allowed with the `start` or `stop` options.

`dump | {dump, Dump}` Specifies the destination for the trace text dump. If this option is not given, no dump is generated, if it is `dump` the destination will be the caller's group leader, otherwise the destination `Dump` is either the pid of an I/O device or a filename. And, finally, if the filename is `[]` - `"fprof.dump"` is used instead. This option is not allowed with the `stop` option.

`append` Causes the trace text dump to be appended to the destination file. This option is only allowed with the `{dump, Dumpfile}` option.

`start` Starts a tracer process that profiles trace data in runtime. The call will return immediately with the return value `{ok, Tracer}` if successful. This option is not allowed with the `stop`, `file` or `{file, Filename}` options.

`stop` Stops the tracer process that profiles trace data in runtime. The return value will be value `ok` if successful. This option is not allowed with the `start`, `file` or `{file, Filename}` options.

`analyse() -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- Reason = term()

Same as `analyse([])`.

`analyse(OptionName, OptionValue) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- OptionValue = term()
- Reason = term()

Same as `analyse([{OptionName, OptionValue}])`.

`analyse(OptionName) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- Reason = term()

Same as `analyse([OptionName])`.

`analyse({OptionName, OptionValue}) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- OptionName = atom()
- OptionValue = term()
- Reason = term()

Same as `analyse([{OptionName, OptionValue}])`.

`analyse([Option]) -> ok | {error, Reason} | {'EXIT', ServerPid, Reason}`

Types:

- Option = dest | {dest, Dest} | append | {cols, Cols} | callers | {callers, bool()} | no_callers | {sort, SortSpec} | totals | {totals, bool()} | details | {details, bool()} | no_details
- Dest = pid() | Destfile
- Cols = integer() >= 80
- SortSpec = acc | own
- Reason = term()

Analyses raw profile data in the `fprof` server. If called while there is no raw profile data available, `{error, no_profile}` is returned.

`Destfile` is used to call `file:open/2`. Please see the appropriate documentation.

Option description:

`dest | {dest, Dest}` Specifies the destination for the analysis. If this option is not given or it is `dest`, the destination will be the caller's group leader, otherwise the destination `Dest` is either the `pid()` of an I/O device or a filename. And, finally, if the filename is `[]` - `"fprof.analysis"` is used instead.

`append` Causes the analysis to be appended to the destination file. This option is only allowed with the `{dest, Destfile}` option.

`{cols, Cols}` Specifies the number of columns in the analysis text. If this option is not given the number of columns is set to 80.

`callers | {callers, true}` Prints callers and called information in the analysis. This is the default.

`{callers, false} | no_callers` Suppresses the printing of callers and called information in the analysis.

`{sort, SortSpec}` Specifies if the analysis should be sorted according to the ACC column, which is the default, or the OWN column. See Analysis Format [page 45] below.

`totals | {totals, true}` Includes a section containing call statistics for all calls regardless of process, in the analysis.

`{totals, false}` Supresses the totals section in the analysis, which is the default.

`details | {details, true}` Prints call statistics for each process in the analysis. This is the default.

`{details, false} | no_details` Suppresses the call statistics for each process from the analysis.

Analysis format

This section describes the output format of the analyse command. See analyse/0 [page 43].

The format is parsable with the standard Erlang parsing tools `erl_scan` and `erl_parse`, `file:consult/1` or `io:read/2`. The parse format is not explained here - it should be easy for the interested to try it out. Note that some flags to `analyse/1` will affect the format.

The following example was run on OTP/R8 on Solaris 8, all OTP internals in this example are very version dependent.

As an example, we will use the following function, that you may recogise as a slightly modified benchmark function from the manpage file(3):

```
-module(foo).
-export([create_file_slow/2]).

create_file_slow(Name, N) when integer(N), N >= 0 ->
    {ok, FD} =
        file:open(Name, [raw, write, delayed_write, binary]),
    if N > 256 ->
            ok = file:write(FD,
                            lists:map(fun (X) -> <<X:32/unsigned>> end,
                            lists:seq(0, 255))),
            ok = create_file_slow(FD, 256, N);
       true ->
            ok = create_file_slow(FD, 0, N)
    end,
    ok = file:close(FD).

create_file_slow(FD, M, M) ->
    ok;
create_file_slow(FD, M, N) ->
    ok = file:write(FD, <<M:32/unsigned>>),
    create_file_slow(FD, M+1, N).
```

Let us have a look at the printout after running:

```
1> fprof:apply(foo, create_file_slow, [junk, 1024]).
2> fprof:profile().
3> fprof:analyse().
```

The printout starts with:

```
%% Analysis results:
{  analysis_options,
 [{callers, true},
  {sort, acc},
  {totals, false},
  {details, true}]}.

%                                        CNT       ACC       OWN
[{ totals,                              9627, 1691.119, 1659.074}].  %%%
```

The CNT column shows the total number of function calls that was found in the trace. In the ACC column is the total time of the trace from first timestamp to last. And in the OWN column is the sum of the execution time in functions found in the trace, not including called functions. In this case it is very close to the ACC time since the emulator had practically nothing else to do than to execute our test program.

All time values in the printout are in milliseconds.

The printout continues:

```
%                                        CNT        ACC       OWN
[{ "<0.28.0>",                          9627,undefined, 1659.074}].   %%
```

This is the printout header of one process. The printout contains only this one process since we did `fprof:apply/3` which traces only the current process. Therefore the CNT and OWN columns perfectly matches the totals above. The ACC column is undefined since summing the ACC times of all calls in the process makes no sense - you would get something like the ACC value from totals above multiplied by the average depth of the call stack, or something.

All paragraphs up to the next process header only concerns function calls within this process.

Now we come to something more interesting:

```
{[{undefined,                             0, 1691.076,    0.030}],
 { {fprof,apply_start_stop,4},           0, 1691.076,    0.030},     %
 [{{foo,create_file_slow,2},             1, 1691.046,    0.103},
  {suspend,                               1,    0.000,    0.000}]}.

{[{{fprof,apply_start_stop,4},           1, 1691.046,    0.103}],
 { {foo,create_file_slow,2},             1, 1691.046,    0.103},     %
 [{{file,close,1},                        1, 1398.873,    0.019},
  {{foo,create_file_slow,3},             1,  249.678,    0.029},
  {{file,open,2},                         1,   20.778,    0.055},
  {{lists,map,2},                         1,   16.590,    0.043},
  {{lists,seq,2},                         1,    4.708,    0.017},
  {{file,write,2},                        1,    0.316,    0.021}]}.
```

The printout consists of one paragraph per called function. The function *marked* with '%' is the one the paragraph concerns - `foo:create_file_slow/2`. Above the marked function are the *calling* functions - those that has called the marked, and below are those *called* by the marked function.

The paragraphs are per default sorted in decreasing order of the ACC column for the marked function. The calling list and called list within one paragraph are also per default sorted in decreasing order of their ACC column.

The columns are: CNT - the number of times the funcion has been called, ACC - the time spent in the function including called functions, and OWN - the time spent in the function not including called functions.

The rows for the *calling* functions contain statistics for the *marked* function with the constraint that only the occasions when a call was made from the *row's* function to the *marked* function are accounted for.

The row for the *marked* function simply contains the sum of all *calling* rows.

The rows for the *called* functions contains statistics for the *row's* function with the constraint that only the occasions when a call was made from the *marked* to the *row's* function are accounted for.

So, we see that `foo:create_file_slow/2` used very little time for its own execution. It spent most of its time in `file:close/1`. The function `foo:create_file_slow/3` that writes 3/4 of the file contents is the second biggest time thief.

We also see that the call to `file:write/2` that writes 1/4 of the file contents takes very little time in itself. What takes time is to build the data (`lists:seq/2` and `lists:map/2`).

The function 'undefined' that has called `fprof:apply_start_stop/4` is an unknown function because that call was not recorded in the trace. It was only recorded that the execution returned from `fprof:apply_start_stop/4` to some other function above in the call stack, or that the process exited from there.

Let us continue down the printout to find:

```
{[{{foo,create_file_slow,2},              1,  249.678,    0.029},
  {{foo,create_file_slow,3},            768,    0.000,   23.294}],
 { {foo,create_file_slow,3},            769,  249.678,   23.323},     %
 [{{file,write,2},                      768,  220.314,   14.539},
  {suspend,                              57,    6.041,    0.000},
  {{foo,create_file_slow,3},            768,    0.000,   23.294}]}.
```

If you compare with the code you will see there also that `foo:create_file_slow/3` was called only from `foo:create_file_slow/2` and itself, and called only `file:write/2`, note the number of calls to `file:write/2`. But here we see that `suspend` was called a few times. This is a pseudo function that indicates that the process was suspended while executing in `foo:create_file_slow/3`, and since there is no `receive` or `erlang:yield/0` in the code, it must be Erlang scheduling suspensions, or the trace file driver compensating for large file write operations (these are regarded as a shedule out followed by a shedule in to the same process).

Let us find the `suspend` entry:

```
{[{{file,write,2},                       53,    6.281,    0.000},
  {{foo,create_file_slow,3},            57,    6.041,    0.000},
  {{prim_file,drv_command,4},           50,    4.582,    0.000},
  {{prim_file,drv_get_response,1},      34,    2.986,    0.000},
  {{lists,map,2},                        10,    2.104,    0.000},
  {{prim_file,write,2},                 17,    1.852,    0.000},
  {{erlang,port_command,2},             15,    1.713,    0.000},
  {{prim_file,drv_command,2},           22,    1.482,    0.000},
  {{prim_file,translate_response,2},    11,    1.441,    0.000},
  {{prim_file,'-drv_command/2-fun-0-',1},  15,    1.340,    0.000},
  {{lists,seq,4},                         3,    0.880,    0.000},
  {{foo,'-create_file_slow/2-fun-0-',1},   5,    0.523,    0.000},
  {{erlang,bump_reductions,1},           4,    0.503,    0.000},
  {{prim_file,open_int_setopts,3},      1,    0.165,    0.000},
  {{prim_file,i32,4},                    1,    0.109,    0.000},
  {{fprof,apply_start_stop,4},          1,    0.000,    0.000}],
 { suspend,                             299,   32.002,    0.000},     %
 [ ]}.
```

We find no particulary long suspend times, so no function seems to have waited in a receive statement. Actually, `prim_file:drv_command/4` contains a receive statement, but in this test program, the message lies in the process receive buffer when the receive statement is entered. We also see that the total suspend time for the test run is small.

The `suspend` pseudo function has got an OWN time of zero. This is to prevent the process total OWN time from including time in suspension. Whether suspend time is really ACC or OWN time is more of a philosophical question.

Now we look at another interesting pesudo function, `garbage_collect`:

```
{[{{prim_file,drv_command,4},           25,    0.873,    0.873},
  {{prim_file,write,2},                 16,    0.692,    0.692},
  {{lists,map,2},                        2,    0.195,    0.195}],
 { garbage_collect,                     43,    1.760,    1.760},     %
 [ ]}.
```

Here we see that no function distinguishes itself considerably, which is very normal.

The `garbage_collect` pseudo function has not got an OWN time of zero like `suspend`, instead it is equal to the ACC time.

Garbage collect often occurs while a process is suspended, but `fprof` hides this fact by pretending that the suspended function was first unsuspended and then garbage collected. Otherwise the printout would show `garbage_collect` being called from `suspend` but not not which function that might have caused the garbage collection.

Let us now get back to the test code:

```
{[{{foo,create_file_slow,3},           768,  220.314,   14.539},
  {{foo,create_file_slow,2},             1,    0.316,    0.021}],
 { {file,write,2},                     769,  220.630,   14.560},     %
 [{{prim_file,write,2},                769,  199.789,   22.573},
  {suspend,                             53,    6.281,    0.000}]}.
```

Not unexpectedly, we see that `file:write/2` was called from `foo:create_file_slow/3` and `foo:create_file_slow/2`. The number of calls in each case as well as the used time are also just confirms the previous results.

We see that `file:write/2` only calls `prim_file:write/2`, but let us refrain from digging into the internals of the kernel application.

But, if we nevertheless *do* dig down we find the call to the linked in driver that does the file operations towards the host operating system:

```
{[{{prim_file,drv_command,4},          772, 1458.356, 1456.643}],
 { {erlang,port_command,2},            772, 1458.356, 1456.643},     %
 [{suspend,                             15,    1.713,    0.000}]}.
```

This is 86 % of the total run time, and as we saw before it is the close operation the absolutely biggest contributor. We find a comparision ratio a little bit up in the call stack:

```
{[{{prim_file,close,1},                   1, 1398.748,    0.024},
  {{prim_file,write,2},                 769,  174.672,   12.810},
  {{prim_file,open_int,4},                1,   19.755,    0.017},
  {{prim_file,open_int_setopts,3},        1,    0.147,    0.016}],
 { {prim_file,drv_command,2},           772, 1593.322,   12.867},     %
 [{{prim_file,drv_command,4},           772, 1578.973,   27.265},
  {suspend,                              22,    1.482,    0.000}]}.
```

The time for file operations in the linked in driver distributes itself as 1 % for open, 11 % for write and 87 % for close. All data is probably buffered in the operating system until the close.

The unsleeping reader may notice that the ACC times for `prim_file:drv_command/2` and `prim_file:drv_command/4` is not equal between the paragraphs above, even though it is easy to beleive that `prim_file:drv_command/2` is just a passthrough function.

The missing time can be found in the paragraph for `prim_file:drv_command/4` where it is evident that not only `prim_file:drv_command/2` is called but also a fun:

```
{[{{prim_file,drv_command,2},           772, 1578.973,   27.265}],
 { {prim_file,drv_command,4},           772, 1578.973,   27.265},     %
 [{{erlang,port_command,2},             772, 1458.356, 1456.643},
  {{prim_file,'-drv_command/2-fun-0-',1}, 772,   87.897,   12.736},
  {suspend,                              50,    4.582,    0.000},
  {garbage_collect,                      25,    0.873,    0.873}]}.
```

And some more missing time can be explained by the fact that `prim_file:open_int/4` both calls `prim_file:drv_command/2` directly as well as through `prim_file:open_int_setopts/3`, which complicates the picture.

```
{[{{prim_file,open,2},                    1,   20.309,    0.029},
  {{prim_file,open_int,4},                1,    0.000,    0.057}],
 { {prim_file,open_int,4},                2,   20.309,    0.086},     %
 [{{prim_file,drv_command,2},             1,   19.755,    0.017},
  {{prim_file,open_int_setopts,3},        1,    0.360,    0.032},
  {{prim_file,drv_open,2},                1,    0.071,    0.030},
  {{erlang,list_to_binary,1},             1,    0.020,    0.020},
  {{prim_file,i32,1},                     1,    0.017,    0.017},
  {{prim_file,open_int,4},                1,    0.000,    0.057}]}.
.
.
.
{[{{prim_file,open_int,4},                1,    0.360,    0.032},
  {{prim_file,open_int_setopts,3},        1,    0.000,    0.016}],
 { {prim_file,open_int_setopts,3},        2,    0.360,    0.048},     %
 [{suspend,                               1,    0.165,    0.000},
  {{prim_file,drv_command,2},             1,    0.147,    0.016},
  {{prim_file,open_int_setopts,3},        1,    0.000,    0.016}]}.
```

Notes

The actual supervision of execution times is in itself a CPU intensive activity. A message is written on the trace file for every function call that is made by the profiled code.

The ACC time calculation is sometimes difficult to make correct, since it is difficult to define. This happens especially when a function occurs in several instances in the call stack, for example by calling itself perhaps through other functions and perhaps even non-tail recursively.

To produce sensible results, `fprof` tries not to charge any function more than once for ACC time. The instance highest up (with longest duration) in the call stack is chosen.

Sometimes a function may unexpectedly waste a lot (some 10 ms or more depending on host machine OS) of OWN (and ACC) time, even functions that does practically nothing at all. The problem may be that the OS has chosen to schedule out the Erlang runtime system process for a while, and if the OS does not support high resolution cpu time measurements `fprof` will use wallclock time for its calculations, and it will appear as functions randomly burn virtual machine time.

See Also

dbg(3), eprof [page 36](3), erlang(3), io(3), Tools User's Guide [page 13]

instrument

Erlang Module

The module `instrument` contains support for studying the resource usage in an Erlang runtime system. Currently, only the allocation of memory can be studied.

Note:
Note that this whole module is experimental, and the representations used as well as the functionality is likely to change in the future.

Some of the functions in this module are only available in Erlang compiled with instrumentation; otherwise they exit with `badarg`. This is noted below for the individual functions. To start an Erlang runtime system with instrumentation, use the command-line option `-instr` to the `erl` command.

The basic object of study in the case of memory allocation is a memory allocation list, which contains one descriptor for each allocated memory block. Currently, a descriptor is a 4-tuple

```
{Type, Address, Size, Pid}
```

where `Type` indicates what the block is used for, `Address` is its place in memory, and `Size` is its size, in bytes. `Pid` is either `undefined` (if the block was allocated by the runtime system itself) or a tuple `{A,B,C}` representing the process which allocated the block, which corresponds to a pid with the user-visible representation `<A.B.C>` (the function `c:pid/3` can be used to transform the numbers to a real pid).

Various details about memory allocation:

On Unix (for example, Solaris), memory for a process is allocated linearly, usually from 0. The current size of the process cannot be obtained from within Erlang, but can be seen with one of the system statistics tools, e.g., `ps` or `top`. (There may be a hole above the highest used memory block; in that case the functions in the `instrument` module cannot tell you about it; you have to compare the `High` value from `mem_limits/1` with the value which the system reports for Erlang.)

In the memory allocation list, certain small objects do not show up individually, since they are allocated from blocks of 20 objects (called “fixalloc” blocks). The blocks themselves do show up, but the amount of internal fragmentation in them currently cannot be observed.

Overhead for instrumentation: instrumented memory allocation uses 28 bytes extra for each block. The time overhead for managing the list is negligible.

Exports

`holes(AllocList) -> ok`

Types:

- AllocList = [Desc]
- Desc = {int(), int(), int(), pid_tuple()}
- pid_tuple() = {int(), int(), int()}

Prints out the size of each hole (i.e., the space between allocated blocks) on the terminal. The list must be sorted (see `sort/1`).

`mem_limits(AllocList) -> {Low, High}`

Types:

- AllocList = [Desc]
- Desc = {int(), int(), int(), pid_tuple()}
- pid_tuple() = {int(), int(), int()}
- Low = High = int()

returns a tuple `{Low, High}` indicating the lowest and highest address used. The list must be sorted (see `sort/1`).

`memory_data() -> AllocList`

Types:

- AllocList = [Desc]
- Desc = {int(), int(), int(), pid_tuple()}
- pid_tuple() = {int(), int(), int()}

Returns the memory allocation list. Only available in an Erlang runtime system compiled for instrumentation. Blocks execution of other processes while the list is collected.

`read_memory_data(File) -> {ok, AllocList} | {error, Reason}`

Types:

- File = string()
- AllocList = [Desc]
- Desc = {int(), int(), int(), pid_tuple()}
- pid_tuple() = {int(), int(), int()}

Reads a memory allocation list from the file `File`. The file is assumed to have been created by `store_memory_data/1`. The error codes are the same as for `file:consult/1`.

`sort(AllocList) -> AllocList`

Types:

- AllocList = [Desc]
- Desc = {int(), int(), int(), pid_tuple()}
- pid_tuple() = {int(), int(), int()}

Sorts a memory allocation list so the addresses are in ascending order. The list arguments to many of the functions in this module must be sorted. No other function in this module returns a sorted list.

`store_memory_data(File) -> ok`

Types:

- File = string()

Stores the memory allocation list on the file `File`. The contents of the file can later be read using `read_memory_data/1`. Only available in an Erlang runtime system compiled for instrumentation. Blocks execution of other processes while the list is collected (the time to write the data is around 0.1 ms/line on a Sun Ultra 1).

Failure: `badarg` if the file could not be written.

`sum_blocks(AllocList) -> int()`

Types:

- AllocList = [Desc]
- Desc = {int(), int(), int(), pid_tuple()}
- pid_tuple() = {int(), int(), int()}

Returns the total size of the memory blocks in the list. The list must be sorted (see `sort/1`).

`type_string(Type) -> string()`

Types:

- Type = int()

Translates a memory block type number into a readable string, which is a short description of the block type.

Failure: `badarg` if the argument is not a valid block type number.

make

Erlang Module

The module `make` provides a set of functions similar to the UNIX type `Make` functions.

Exports

`all() -> up_to_date | error`
`all(Options) -> up_to_date | error`

Types:

- Options = [Option]
- Option = noexec | load | netload | par | <compiler option>

This function first looks in the current working directory for a file named `Emakefile` (see below) specifying the set of modules to compile. If no such file is found, the set of modules to compile defaults to all modules in the current working directory.

Traversing the set of modules, it then recompiles every module for which at least one of the following conditions apply:

- there is no object file, or
- the source file has been modified since it was last compiled, or,
- an include file has been modified since the source file was last compiled.

As a side effect, the function prints the name of each module it tries to compile. If compilation fails for a module, the make procedure stops and `error` is returned.

`Options` is a list of make- and compiler options. The following make options exist:

- `noexec`
  No execution mode. Just prints the name of each module that needs to be compiled.
- `load`
  Load mode. Loads all recompiled modules.
- `netload`
  Net load mode. Loads all recompiled modules an all known nodes.
- `par`
  Parallel mode. `make` is used in parallell on all known nodes.

All items in `Options` that are not make options are assumed to be compiler options and are passed as-is to `compile:file/2`. `Options` defaults to `[]`.

`files(ModFiles) -> up_to_date | error`

`files(ModFiles, Options) -> up_to_date | error`

Types:

- ModFiles = [Module | File]
- Module = atom()
- File = string()
- Options = [Option]
- Option = noexec | load | netload | par | <compiler option>

`files/1,2` does exactly the same thing as `all/0,1` but for the specified `ModFiles`, which is a list of module or file names. The file extension `.erl` may be omitted.

Files

`make:all/0,1` looks in the current working directory for a file named `Emakefile` for the set of modules to compile. If it exists, `Emakefile` should contain the module names (atoms) separated by periods. If the module is located in another directory, the path has to be specified. For example:

```
file1.
file2.
'../foo/file3'.
'File4'.
```

tags

Erlang Module

A `TAGS` file is used by Emacs to find function and variable definitions in any source file in large projects. This module can generate a `TAGS` file from Erlang source files. It recognises functions, records, and macro definitions.

Exports

`file(File [, Options])`

Create a `TAGS` file for the file `File`.

`files(FileList [, Options])`

Create a TAGS file for the files in the list `FileList`.

`dir(Dir [, Options])`

Create a TAGS file for all files in directory `Dir`.

`dirs(DirList [, Options])`

Create a TAGS file for all files in any directory in `DirList`.

`subdir(Dir [, Options])`

Descend recursively down the directory `Dir` and create a `TAGS` file based on all files found.

`subdirs(DirList [, Options])`

Descend recursively down all the directories in `DirList` and create a `TAGS` file based on all files found.

`root([Options])`

Create a `TAGS` file covering all files in the Erlang distribution.

OPTIONS

The functions above have an optional argument, `Options`. It is a list which can contain the following elements:

- `{outfile, NameOfTAGSFile}` Create a `TAGS` file named `NameOfTAGSFile`.
- `{outdir, NameOfDirectory}` Create a file named `TAGS` in the directory `NameOfDirectory`.

The default behaviour is to create a file named `TAGS` in the current directory.

Examples

- `tags:root([{outfile, "root.TAGS"}]).`
  This command will create a file named `root.TAGS` in the current directory. The file will contain references to all Erlang source files in the Erlang distribution.
- `tags:files(["foo.erl", "bar.erl", "baz.erl"], [{outdir, "../projectdir"}]).`
  Here we create file named `TAGS` placed it in the directory `../projectdir`. The file contains information about the functions, records, and macro definitions of the three files.

SEE ALSO

- Richard M. Stallman. GNU Emacs Manual, chapter “Editing Programs”, section “Tag Tables”. Free Software Foundation, 1995.
- Anders Lindgren. The Erlang editing mode for Emacs. Ericsson, 1998.

xref

Erlang Module

Xref is a cross reference tool that can be used for finding dependencies between functions, modules, applications and releases.

Calls between functions are either *local calls* like `f()`, or *external calls* like `m:f()`. *Module data*, which are extracted from BEAM files, include local functions, exported functions, local calls and external calls. By default, calls to built-in functions (*BIF*) are ignored, but if the option `builtins`, accepted by some of this module's functions, is set to `true`, calls to BIFs are included as well. It is the analyzing OTP version that decides what functions are BIFs. Functional objects are assumed to be called where they are created (and nowhere else). *Unresolved calls* are calls to `apply` or `spawn` with variable module, variable function, or variable arguments. Examples are `M:F(a)`, `apply(M, f, [a])`, and `spawn(m, f(), Args)`. Unresolved calls are represented by calls where variable modules have been replaced with the atom `'$M_EXPR'`, variable functions have been replaced with the atom `'$F_EXPR'`, and variable number of arguments have been replaced with the number `-1`. The above mentioned examples are represented by calls to `'$M_EXPR':'$F_EXPR'/1`, `'$M_EXPR':f/1`, and `m:'$F_EXPR'/-1`. The unresolved calls are a subset of the external calls.

Warning:
Unresolved calls make module data incomplete, which implies that the results of analyses may be invalid.

Applications are collections of modules. The modules' BEAM files are located in the `ebin` subdirectory of the application directory. The name of the application directory determines the name and version of the application. *Releases* are collections of applications located in the `lib` subdirectory of the release directory. There is more to read about applications and releases in the Design Principles book.

Xref servers are identified by names, supplied when creating new servers. Each Xref server holds a set of releases, a set of applications, and a set of modules with module data. Xref servers are independent of each other, and all analyses are evaluated in the context of one single Xref server (exceptions are the functions `m/1` and `d/1` which do not use servers at all). The *mode* of an Xref server determines what module data are extracted from BEAM files as modules are added to the server. Starting with R7, BEAM files compiled with the option `debug_info` contain so called debug information, which is an abstract representation of the code. In `functions` mode, which is the default mode, function calls and line numbers are extracted from debug information. In `modules` mode, debug information is ignored if present, but dependencies between modules are extracted from other parts of the BEAM files. The `modules` mode is significantly less time and space consuming than the `functions` mode, but the analyses that can be done are limited.

An *analyzed module* is a module that has been added to an Xref server together with its module data. A *library module* is a module located in some directory mentioned in the *library path*. A library module is said to be used if some of its exported functions are used by some analyzed module. An *unknown module* is a module that is neither an analyzed module nor a library module, but whose exported functions are used by some analyzed module. An *unknown function* is a used function that is neither local or exported by any analyzed module nor exported by any library module. An *undefined function* is an externally used function that is not exported by any analyzed module or library module. With this notion, a local function can be an undefined function, namely if it is used externally from some module. All unknown functions are also undefined functions; there is a figure [page 17] in the User's Guide that illustrates this relationship.

Before any analysis can take place, module data must be *set up*. For instance, the cross reference and the unknown functions are computed when all module data are known. The functions that need complete data (`analyze`, `q`, `variables`) take care of setting up data automatically. Module data need to be set up (again) after calls to any of the `add`, `replace`, `remove`, `set_library_path` or `update` functions.

The result of setting up module data is the *Call Graph*. A (directed) graph consists of a set of vertices and a set of (directed) edges. The edges represent *calls* (From, To) between functions, modules, applications or releases. From is said to call To, and To is said to be used by From. The vertices of the Call Graph are the functions of all module data: local and exported functions of analyzed modules; used BIFs; used exported functions of library modules; and unknown functions. The functions `module_info/0,1` added by the compiler are included among the exported functions, but only when called from some module. The edges are the function calls of all module data. A consequence of the edges being a set is that there is only one edge if a function is used locally or externally several times on one and the same line of code.

The Call Graph is represented by Erlang terms (the sets are lists), which is suitable for many analyses. But for analyses that look at chains of calls, a list representation is much too slow. Instead the representation offered by the `digraph` module is used. The translation of the list representation of the Call Graph - or a subgraph thereof - to the `digraph` representation does not come for free, so the language used for expressing queries to be described below has a special operator for this task and a possibility to save the `digraph` representation for subsequent analyses.

In addition to the Call Graph there is a graph called the *Inter Call Graph*. This is a graph of calls (From, To) such that there is a chain of calls from From to To in the Call Graph, and each of From and To is an exported function or an unused local function. The vertices are the same as for the Call Graph.

Calls between modules, applications and releases are also directed graphs. The *types* of the vertices and edges of these graphs are (ranging from the most special to the most general): `Fun` for functions; `Mod` for modules; `App` for applications; and `Rel` for releases. The following paragraphs will describe the different constructs of the language used for selecting and analyzing parts of the graphs, beginning with the *constants*:

- Expression ::= Constants
- Constants ::= Consts | Consts : Type | RegExpr
- Consts ::= Constant | [Constant, ...] | {Constant, ...}
- Constant ::= Call | Const
- Call ::= FunSpec -> FunSpec | {MFA, MFA} | AtomConst -> AtomConst | {AtomConst, AtomConst}
- Const ::= AtomConst | FunSpec | MFA

- AtomConst ::= Application | Module | Release
- FunSpec ::= Module : Function / Arity
- MFA ::= {Module, Function, Arity}
- RegExpr ::= RegString : Type | RegFunc | RegFunc : Type
- RegFunc ::= RegModule : RegFunction / RegArity
- RegModule ::= RegAtom
- RegFunction ::= RegAtom
- RegArity ::= RegString | Number | _
- RegAtom ::= RegString | Atom | _
- RegString ::= - a regular expression, as described in the `regexp` module, enclosed in double quotes -
- Type ::= `Fun` | `Mod` | `App` | `Rel`
- Function ::= Atom
- Application ::= Atom
- Module ::= Atom
- Release ::= Atom
- Arity ::= Number
- Atom ::= - same as Erlang atoms -
- Number ::= - same as non-negative Erlang integers -

Examples of constants are: `kernel`, `kernel->stdlib`, `[kernel, sasl]`, `[pg -> mnesia, {tv, mnesia}] : Mod`. It is an error if an instance of `Const` does not match any vertex of any graph. If there are more than one vertex matching an untyped instance of `AtomConst`, then the one of the most general type is chosen. A list of constants is interpreted as a set of constants, all of the same type. A tuple of constants constitute a chain of calls (which may, but does not have to, correspond to an actual chain of calls of some graph). Assigning a type to a list or tuple of `Constant` is equivalent to assigning the type to each `Constant`.

Regular expressions are used as a means to select some of the vertices of a graph. A `RegExpr` consisting of a `RegString` and a type - an example is `"xref_.*" : Mod` - is interpreted as those modules (or applications or releases, depending on the type) that match the expression. Similarly, a `RegFunc` is interpreted as those vertices of the Call Graph that match the expression. An example is `"xref_.*":"add_.*"/"(2|3)"`, which matches all `add` functions of arity two or three of any of the xref modules. Another example, one that matches all functions of arity 10 or more: `_:_/"[1-9].+"`. Here `_` is an abbreviation for `".*"`, that is, the regular expression that matches anything.

The syntax of *variables* is simple:

- Expression ::= Variable
- Variable ::= - same as Erlang variables -

There are two kinds of variables: predefined variables and user variables. *Predefined variables* hold set up module data, and cannot be assigned to but only used in queries. *User variables* on the other hand can be assigned to, and are typically used for temporary results while evaluating a query, and for keeping results of queries for use in subsequent queries. The predefined variables are (variables marked with (*) are available in `functions` mode only):

E Call Graph Edges (*).

V Call Graph Vertices (*).

M Modules. All modules: analyzed modules, used library modules, and unknown modules.

A Applications.

R Releases.

ME Module Edges. All module calls.

AE Application Edges. All application calls.

RE Release Edges. All release calls.

L Local Functions (*). All local functions of analyzed modules.

X Exported Functions. All exported functions of analyzed modules and all used exported functions of library modules.

F Functions (*).

B Used BIFs. B is empty if `builtins` is `false` for all analyzed modules.

U Unknown Functions.

UU Unused Functions (*). All local and exported functions of analyzed modules that have not been used.

XU Externally Used Functions. Functions of all modules - including local functions - that have been used in some external call.

LU Locally Used Functions (*). Functions of all modules that have been used in some local call.

LC Local Calls (*).

XC External Calls (*).

AM Analyzed Modules.

UM Unknown Modules.

LM Used Library Modules.

UC Unresolved Calls. Empty in `modules` mode.

EE Inter Call Graph Edges (*).

These are a few facts about the predefined variables (the set operators + (union) and − (difference) as well as the cast operator (Type) are described below):

- F is equal to `L + X`.
- V is equal to `X + L + B + U`, where X, L, B and U are pairwise disjoint (that is, have no elements in common).
- UU is equal to `V - (XU + LU)`, where LU and XU may have elements in common. Put in another way:
- V is equal to `UU + XU + LU`.
- E is equal to `LC + XC`. Note that LC and XC may have elements in common, namely if some function is used locally and externally from one and the same function.
- U is a subset of XU.
- B is a subset of XU.
- LU is equal to `range LC`.
- XU is equal to `range XC`.

- LU is a subset of F.
- UU is a subset of F.
- range UC is a subset of U.
- M is equal to AM + LM + UM, where AM, LM and UM are pairwise disjoint.
- ME is equal to (Mod) E.
- AE is equal to (App) E.
- RE is equal to (Rel) E.
- (Mod) V is a subset of M. Equality holds if all analyzed modules have some local, exported, or unknown function.
- (App) M is a subset of A. Equality holds if all applications have some module.
- (Rel) A is a subset of R. Equality holds if all releases have some application.

An important notion is that of *conversion* of expressions. The syntax of a cast expression is:

- Expression ::= (Type) Expression

The interpretation of the cast operator depends on the named type Type, the type of Expression, and the structure of the elements of the interpretation of Expression. If the named type is equal to the expression type, no conversion is done. Otherwise, the conversion is done one step at a time; (Fun) (App) RE, for instance, is equivalent to (Fun) (Mod) (App) RE. Now assume that the interpretation of Expression is a set of constants (functions, modules, applications or releases). If the named type is more general than the expression type, say Mod and Fun respectively, then the interpretation of the cast expression is the set of modules that have at least one of their functions mentioned in the interpretation of the expression. If the named type is more special than the expression type, say Fun and Mod, then the interpretation is the set of all the functions of the modules (in modules mode, the conversion is partial since the local functions are not known). The conversions to and from applications and releases work analogously. For instance, (App) "xref_.*" : Mod returns all applications containing at least one module such that xref_ is a prefix of the module name.

Now assume that the interpretation of Expression is a set of calls. If the named type is more general than the expression type, say Mod and Fun respectively, then the interpretation of the cast expression is the set of calls (M1, M2) such that the interpretation of the expression contains a call from some function of M1 to some function of M2. If the named type is more special than the expression type, say Fun and Mod, then the interpretation is the set of all function calls (F1, F2) such that the interpretation of the expression contains a call (M1, M2) and F1 is a function of M1 and F2 is a function of M2 (in modules mode, there are no functions calls, so a cast to Fun always yields an empty set). Again, the conversions to and from applications and releases work analogously.

The interpretation of constants and variables are sets, and those sets can be used as the basis for forming new sets by the application of *set operators*. The syntax:

- Expression ::= Expression BinarySetOp Expression
- BinarySetOp ::= + | * | -

`+`, `*` and `-` are interpreted as union, intersection and difference respectively: the union of two sets contains the elements of both sets; the intersection of two sets contains the elements common to both sets; and the difference of two sets contains the elements of the first set that are not members of the second set. The elements of the two sets must be of the same structure; for instance, a function call cannot be combined with a function. But if a cast operator can make the elements compatible, then the more general elements are converted to the less general element type. For instance, `M + F` is equivalent to `(Fun) M + F`, and `E - AE` is equivalent to `E - (Fun) AE`. One more example: `X * xref : Mod` is interpreted as the set of functions exported by the module `xref`; `xref : Mod` is converted to the more special type of `X` (`Fun`, that is) yielding all functions of `xref`, and the intersection with `X` (all functions exported by analyzed modules and library modules) is interpreted as those functions that are exported by some module *and* functions of `xref`.

There are also unary set operators:

- Expression ::= UnarySetOp Expression
- UnarySetOp ::= `domain` | `range` | `strict`

Recall that a call is a pair (From, To). `domain` applied to a set of calls is interpreted as the set of all vertices From, and `range` as the set of all vertices To. The interpretation of the `strict` operator is the operand with all calls on the form (A, A) removed.

The interpretation of the *restriction operators* is a subset of the first operand, a set of calls. The second operand, a set of vertices, is converted to the type of the first operand. The syntax of the restriction operators:

- Expression ::= Expression RestrOp Expression
- RestrOp ::= `|`
- RestrOp ::= `||`
- RestrOp ::= `|||`

The interpretation in some detail for the three operators:

`|` The subset of calls from any of the vertices.

`||` The subset of calls to any of the vertices.

`|||` The subset of calls to and from any of the vertices. For all sets of calls `CS` and all sets of vertices `VS`, `CS ||| VS` is equivalent to `CS | VS * CS || VS`.

Two functions (modules, applications, releases) belong to the same strongly connected component if they call each other (in)directly. The interpretation of the `components` operator is the set of strongly connected components of a set of calls. The `condensation` of a set of calls is a new set of calls between the strongly connected components such that there is an edge between two components if there is some constant of the first component that calls some constant of the second component.

The interpretation of the `of` operator is a chain of calls of the second operand (a set of calls) that passes throw all of the vertices of the first operand (a tuple of constants), in the given order. The second operand is converted to the type of the first operand. For instance, the `of` operator can be used for finding out whether a function calls another function indirectly, and the chain of calls demonstrates how. The syntax of the graph analyzing operators:

- Expression ::= Expression GraphOp Expression

- GraphOp ::= `components` | `condensation` | `of`

As was mentioned before, the graph analyses operate on the `digraph` representation of graphs. By default, the `digraph` representation is created when needed (and deleted when no longer used), but it can also be created explicitly by use of the `closure` operator:

- Expression ::= ClosureOp Expression
- ClosureOp ::= `closure`

The interpretation of the `closure` operator is the transitive closure of the operand.

The restriction operators are defined for closures as well; `closure E | xref : Mod` is interpreted as the direct or indirect function calls from the `xref` module, while the interpretation of `E | xref : Mod` is the set of direct calls from `xref`. If some graph is to be used in several graph analyses, it saves time to assign the `digraph` representation of the graph to a user variable, and then make sure that each graph analysis operates on that variable instead of the list representation of the graph.

The lines where functions are defined (more precisely: where the first clause begins) and the lines where functions are used are available in `functions` mode. The line numbers refer to the files where the functions are defined. This holds also for files included with the `-include` and `-include_lib` directives, which may result in functions defined apparently in the same line. The *line operators* are used for assigning line numbers to functions and for assigning sets of line numbers to function calls. The syntax is similar to the one of the cast operator:

- Expression ::= (LineOp) Expression
- Expression ::= (XLineOp) Expression
- LineOp ::= `Lin` | `ELin` | `LLin` | `XLin`
- XLineOp ::= `XXL`

The interpretation of the `Lin` operator applied to a set of functions assigns to each function the line number where the function is defined. Unknown functions and functions of library modules are assigned the number 0.

The interpretation of some LineOp operator applied to a set of function calls assigns to each call the set of line numbers where the first function calls the second function. Not all calls are assigned line numbers by all operators:

- the `Lin` operator is defined for Call Graph Edges;
- the `LLin` operator is defined for Local Calls.
- the `XLin` operator is defined for External Calls.
- the `ELin` operator is defined for Inter Call Graph Edges.

The `Lin` (`LLin`, `XLin`) operator assigns the lines where calls (local calls, external calls) are made. The `ELin` operator assigns to each call (From, To), for which it is defined, each line L such that there is a chain of calls from From to To beginning with a call on line L.

The `XXL` operator is defined for the interpretation of any of the LineOp operators applied to a set of function calls. The result is that of replacing the function call with a line numbered function call, that is, each of the two functions of the call is replaced by a pair of the function and the line where the function is defined. The effect of the `XXL` operator can be undone by the LineOp operators. For instance, `(Lin) (XXL) (Lin) E` is equivalent to `(Lin) E`.

The `+`, `-`, `*` and `#` operators are defined for line number expressions, provided the operands are compatible. The LineOp operators are also defined for modules, applications, and releases; the operand is implicitly converted to functions. Similarly, the cast operator is defined for the interpretation of the LineOp operators.

The interpretation of the *counting operator* is the number of elements of a set. The operator is undefined for closures. The `+`, `-` and `*` operators are interpreted as the obvious arithmetical operators when applied to numbers. The syntax of the counting operator:

- Expression ::= CountOp Expression
- CountOp ::= `#`

All binary operators are left associative; for instance, `A | B || C` is equivalent to `(A | B) || C`. The following is a list of all operators, in increasing order of *precedence*:

- `+`, `-`
- `*`
- `#`
- `|`, `||`, `|||`
- `of`
- `(`Type`)`
- `closure`, `components`, `condensation`, `domain`, `range`, `strict`

Parentheses are used for grouping, either to make an expression more readable or to override the default precedence of operators:

- Expression ::= `(` Expression `)`

A *query* is a non-empty sequence of statements. A statement is either an assignment of a user variable or an expression. The value of an assignment is the value of the right hand side expression. It makes no sense to put a plain expression anywhere else but last in queries. The syntax of queries is summarized by these productions:

- Query ::= Statement`,` ...
- Statement ::= Assignment `|` Expression
- Assignment ::= Variable `:=` Expression `|` Variable `=` Expression

A variable cannot be assigned a new value unless first removed. Variables assigned to by the `=` operator are removed at the end of the query, while variables assigned to by the `:=` operator can only be removed by calls to `forget`. There are no user variables when module data need to be set up again; if any of the functions that make it necessary to set up module data again is called, all user variables are forgotten.

Types

```
application() = atom()
arity() = int()
bool() = true | false
call() = {atom(), atom()} | funcall()
constant() = mfa() | module() | application() | release()
directory() = string()
file() = string()
funcall() = {mfa(), mfa()}
function() = atom()
int() = integer() >= 0
library() = atom()
library_path() = path() | code_path
mfa() = {module(), function(), arity()}
mode() = functions | modules
module() = atom()
release() = atom()
string_position() = int() | at_end
variable() = atom()
xref() = atom()
```

Exports

`add_application(Xref, Directory [, Options]) -> {ok, application()} | Error`

Types:

- Directory = directory()
- Error = {error, module(), Reason}
- Options = [Option] | Option
- Option = {builtins, bool()} | {name, application()} | {verbose, bool()} | {warnings, bool()}
- Reason = {application_clash, {application(), directory(), directory()}} | {file_error, file(), error()} | {invalid_filename, term()} | {invalid_options, term()} |
  - see also add_directory -
- Xref = xref()

Adds an application, the modules of the application and module data [page 58] of the modules to an Xref server [page 58]. The modules will be members of the application. The default is to use the base name of the directory with the version removed as application name, but this can be overridden by the `name` option. Returns the name of the application.

If the given directory has a subdirectory named `ebin`, modules (BEAM files) are searched for in that directory, otherwise modules are searched for in the given directory.

If the mode [page 58] of the Xref server is `functions`, BEAM files that contain no debug information [page 58] are ignored.

`add_directory(Xref, Directory [, Options]) -> {ok, Modules} | Error`

Types:

- Directory = directory()
- Error = {error, module(), Reason}

- Modules = [module()]
- Options = [Option] | Option
- Option = {builtins, bool()} | {recurse, bool()} | {verbose, bool()} | {warnings, bool()}
- Reason = {file_error, file(), error()} | {invalid_filename, term()} | {invalid_options, term()} | {unrecognized_file, file()} | - error from beam_lib:chunks/2 -
- Xref = xref()

Adds the modules found in the given directory and the modules' data [page 58] to an Xref server [page 58]. The default is not to examine subdirectories, but if the option `recurse` has the value `true`, modules are searched for in subdirectories on all levels as well as in the given directory. Returns a sorted list of the names of the added modules.

The modules added will not be members of any applications.

If the mode [page 58] of the Xref server is `functions`, BEAM files that contain no debug information [page 58] are ignored.

`add_module(Xref, File [, Options]) -> {ok, module()} | Error`

Types:

- Error = {error, module(), Reason}
- File = file()
- Options = [Option] | Option
- Option = {builtins, bool()} | {verbose, bool()} | {warnings, bool()}
- Reason = {file_error, file(), error()} | {invalid_filename, term()} | {invalid_options, term()} | {module_clash, {module(), file(), file()}} | {no_debug_info, file()} | - error from beam_lib:chunks/2 -
- Xref = xref()

Adds a module and its module data [page 58] to an Xref server [page 58]. The module will not be member of any application. Returns the name of the module.

If the mode [page 58] of the Xref server is `functions`, and the BEAM file contains no debug information [page 58], the error message `no_debug_info` is returned.

`add_release(Xref, Directory [, Options]) -> {ok, release()} | Error`

Types:

- Directory = directory()
- Error = {error, module(), Reason}
- Options = [Option] | Option
- Option = {builtins, bool()} | {name, release()} | {verbose, bool()} | {warnings, bool()}
- Reason = {application_clash, {application(), directory(), directory()}} | {file_error, file(), error()} | {invalid_filename, term()} | {invalid_options, term()} | {release_clash, {release(), directory(), directory()}} | - see also add_directory -
- Xref = xref()

Adds a release, the applications of the release, the modules of the applications, and module data [page 58] of the modules to an Xref server [page 58]. The applications will be members of the release, and the modules will be members of the applications. The default is to use the base name of the directory as release name, but this can be overridden by the `name` option. Returns the name of the release.

If the given directory has a subdirectory named `lib`, the directories in that directory are assumed to be application directories, otherwise all subdirectories of the given directory are assumed to be application directories. If there are several versions of some application, the one with the highest version is chosen.

If the mode [page 58] of the Xref server is `functions`, BEAM files that contain no debug information [page 58] are ignored.

`analyze(Xref, Analysis [, Options]) -> {ok, Answer} | Error`

Types:

- Analysis = undefined_function_calls | undefined_functions | locals_not_used | exports_not_used | {call, FuncSpec} | {use, FuncSpec} | {module_call, ModSpec} | {module_use, ModSpec} | {application_call, AppSpec} | {application_use, AppSpec} | {release_call, RelSpec} | {release_use, RelSpec}
- Answer = [term()]
- AppSpec = application() | [application()]
- Error = {error, module(), Reason}
- FuncSpec = mfa() | [mfa()]
- ModSpec = module() | [module()]
- Options = [Option] | Option
- Option = {verbose, bool()}
- RelSpec = release() | [release()]
- Reason = {invalid_options, term()} | {parse_error, string_position(), term()} | {unavailable_analysis, term()} | {unknown_analysis, term()} | {unknown_constant, string()} | {unknown_variable, variable()}
- Xref = xref()

Evaluates a predefined analysis. Returns a sorted list without duplicates of `call()` or `constant()`, depending on the chosen analysis. The predefined analyses, which operate on all analyzed modules [page 59], are (analyses marked with (*) are available in `functions` mode [page 58] only):

`undefined_function_calls` **(*)** Returns a list of calls to undefined functions [page 59].

`undefined_functions` Returns a list of undefined functions [page 59].

`locals_not_used` **(*)** Returns a list of local functions that have not been used locally.

`exports_not_used` Returns a list of exported functions that have not been used externally.

`{call, FuncSpec}` **(*)** Returns a list of functions called by some of the given functions.

`{use, FuncSpec}` **(*)** Returns a list of functions that use some of the given functions.

`{module_call, ModSpec}` Returns a list of modules called by some of the given modules.

`{module_use, ModSpec}` Returns a list of modules that use some of the given modules.

{`application_call, AppSpec`} Returns a list of applications called by some of the given applications.

{`application_use, AppSpec`} Returns a list of applications that use some of the given applications.

{`release_call, RelSpec`} Returns a list of releases called by some of the given releases.

{`release_use, RelSpec`} Returns a list of releases that use some of the given releases.

`d(Directory) -> [DebugInfoResult] | [NoDebugInfoResult] | Error`

Types:

- Directory = directory()
- DebugInfoResult = {undefined, [funcall()]} | {unused, [mfa()]}
- Error = {error, module(), Reason}
- NoDebugInfoResult = {undefined, [mfa()]}
- Reason = {file_error, file(), error()} | {invalid_filename, term()} | {unrecognized_file, file()} | - error from beam_lib:chunks/2 -

The modules found in the given directory are checked for calls to undefined functions [page 59] and for unused local functions. The code path is used as library path [page 59].

If some of the found BEAM files contain debug information [page 58], then those modules are checked and a list of tuples is returned. The first element of each tuple is one of:

- `undefined`, the second element is a sorted list of calls to undefined functions;
- `unused`, the second element is a sorted list of unused local functions.

If no BEAM file contains debug information, then a list of one tuple is returned. The first element of the tuple is `undefined`, and the second element is a sorted list of undefined functions.

`forget(Xref) -> ok`
`forget(Xref, Variables) -> ok | Error`

Types:

- Error = {error, module(), Reason}
- Reason = {not_user_variable, term()}
- Variables = [variable()] | variable()
- Xref = xref()

`forget/1` and `forget/2` remove all or some of the user variables [page 60] of an xref server [page 58].

`format_error(Error) -> character_list()`

Types:

- Error = {error, module(), term()}

Given the error returned by any function of this module, the function `format_error` returns a descriptive string of the error in English. For file errors, the function `format_error/1` in the `file` module is called.

`get_default(Xref) -> [{Option, Value}]`
`get_default(Xref, Option) -> {ok, Value} | Error`

Types:

- Error = {error, module(), Reason}
- Option = builtins | recurse | verbose | warnings
- Reason = {invalid_options, term()}
- Value = bool()
- Xref = xref()

Returns the default values of one or more options.

`get_library_path(Xref) -> {ok, LibraryPath}`

Types:

- LibraryPath = library_path()
- Xref = xref()

Returns the library path [page 59].

`info(Xref) -> [Info]`
`info(Xref, Category) -> [{Item, [Info]}]`
`info(Xref, Category, Items) -> [{Item, [Info]}]`

Types:

- Application = [] | [application()]
- Category = modules | applications | releases | libraries
- Info = {application, Application} | {builtins, bool()} | {directory, directory()} | {library_path, library_path()} | {mode, mode()} | {no_analyzed_modules, int()} | {no_applications, int()} | {no_calls, {NoResolved, NoUnresolved}} | {no_function_calls, {NoLocal, NoResolvedExternal, NoUnresolved}} | {no_functions, {NoLocal, NoExternal}} | {no_inter_function_calls, int()} | {no_releases, int()} | {release, Release} | {version, Version}
- Item = module() | application() | release() | library()
- Items = Item | [Item]
- NoLocal = NoExternal = NoResolvedExternal, NoResolved = NoUnresolved = int()
- Release = [] | [release()]
- Version = [int()]
- Xref = xref()

The `info` functions return information as a list of pairs {Tag, term()} in some order about the state and the module data [page 58] of an Xref server [page 58].

`info/1` returns information with the following tags (tags marked with (*) are available in `functions` mode only):

- `library_path`, the library path [page 59];
- `mode`, the mode [page 58];

- no_releases, number of releases;
- no_applications, total number of applications (of all releases);
- no_analyzed_modules, total number of analyzed modules [page 59];
- no_calls (*), total number of calls (in all modules), regarding instances of one function call in different lines as separate calls;
- no_function_calls (*), total number of local calls [page 58], resolved external calls [page 58] and unresolved calls [page 58];
- no_functions (*), total number of local and exported functions;
- no_inter_function_calls (*), total number of calls of the Inter Call Graph [page 59].

info/2 and info/3 return information about all or some of the analyzed modules, applications, releases or library modules of an Xref server. The following information is returned for each analyzed module:

- application, an empty list if the module does not belong to any application, otherwise a list of the application name;
- builtins, whether calls to BIFs are included in the module's data;
- directory, the directory where the module's BEAM file is located;
- no_calls (*), number of calls, regarding instances of one function call in different lines as separate calls;
- no_function_calls (*), number of local calls, resolved external calls and unresolved calls;
- no_functions (*), number of local and exported functions;
- no_inter_function_calls (*), number of calls of the Inter Call Graph;

The following information is returned for each application:

- directory, the directory where the modules' BEAM files are located;
- no_analyzed_modules, number of analyzed modules;
- no_calls (*), number of calls of the application's modules, regarding instances of one function call in different lines as separate calls;
- no_function_calls (*), number of local calls, resolved external calls and unresolved calls of the application's modules;
- no_functions (*), number of local and exported functions of the application's modules;
- no_inter_function_calls (*), number of calls of the Inter Call Graph of the application's modules;
- release, an empty list if the application does not belong to any release, otherwise a list of the release name;
- version, the application's version as a list of numbers. For instance, the directory "kernel-2.6" results in the application name kernel and the application version [2,6]; "kernel" yields the name kernel and the version [].

The following information is returned for each release:

- directory, the release directory;
- no_analyzed_modules, number of analyzed modules;

- `no_applications`, number of applications;
- `no_calls` (*), number of calls of the release's modules, regarding instances of one function call in different lines as separate calls;
- `no_function_calls` (*), number of local calls, resolved external calls and unresolved calls of the release's modules;
- `no_functions` (*), number of local and exported functions of the release's modules;
- `no_inter_function_calls` (*), number of calls of the Inter Call Graph of the release's modules.

The following information is returned for each library module:

- `directory`, the directory where the library module's [page 59] BEAM file is located.

For each number of calls, functions etc. returned by the `no_` tags, there is a query returning the same number. Listed below are examples of such queries. Some of the queries return the sum of a two or more of the `no_` tags numbers. `mod` (`app`, `rel`) refers to any module (application, release).

- `no_analyzed_modules`
  - `"# AM"` (info/1)
  - `"# (Mod) app:App"` (application)
  - `"# (Mod) rel:Rel"` (release)
- `no_applications`
  - `"# A"` (info/1)
- `no_calls`. The sum of the number of resolved and unresolved calls:
  - `"# (XLin) E + # (LLin) E"` (info/1)
  - `"T = E | mod:Mod, # (LLin) T + # (XLin) T"` (module)
  - `"T = E | app:App, # (LLin) T + # (XLin) T"` (application)
  - `"T = E | rel:Rel, # (LLin) T + # (XLin) T"` (release)
- `no_functions`. Functions in library modules and the functions `module_info/0,1` are not counted by `info`. Assuming that `"Extra := _:module_info/\"(0|1)\" + LM"` has been evaluated, the sum of the number of local and exported functions are:
  - `"# (F - Extra)"` (info/1)
  - `"# (F * mod:Mod - Extra)"` (module)
  - `"# (F * app:App - Extra)"` (application)
  - `"# (F * rel:Rel - Extra)"` (release)
- `no_function_calls`. The sum of the number of local calls, resolved external calls and unresolved calls:
  - `"# LC + # XC"` (info/1)
  - `"# LC | mod:Mod + # XC | mod:Mod"` (module)
  - `"# LC | app:App + # XC | app:App"` (application)
  - `"# LC | rel:Rel + # XC | mod:Rel"` (release)
- `no_inter_function_calls`
  - `"# EE"` (info/1)
  - `"# EE | mod:Mod"` (module)

  - "# EE | app:App" (application)
  - "# EE | rel:Rel" (release)
- no_releases
  - "# R" (info/1)

`m(Module) -> [DebugInfoResult] | [NoDebugInfoResult] | Error`
`m(File) -> [DebugInfoResult] | [NoDebugInfoResult] | Error`

Types:

- DebugInfoResult = {undefined, [funcall()]} | {unused, [mfa()]}
- Error = {error, module(), Reason}
- File = file()
- Module = module()
- NoDebugInfoResult = {undefined, [mfa()]}
- Reason = {file_error, file(), error()} | {interpreted, module()} | {invalid_filename, term()} | {cover_compiled, module()} | {no_such_module, module()} | - error from beam_lib:chunks/2 -

The given BEAM file (with or without the `.beam` extension) or the file found by calling `code:which(Module)` is checked for calls to undefined functions [page 59] and for unused local functions. The code path is used as library path [page 59]. If the BEAM file contains debug information [page 58], then a list of tuples is returned. The first element of each tuple is one of:

- `undefined`, the second element is a sorted list of calls to undefined functions;
- `unused`, the second element is a sorted list of unused local functions.

If the BEAM file does not contain debug information, then a list of one tuple is returned. The first element of the tuple is `undefined`, and the second element is a sorted list of undefined functions.

`q(Xref, Query [, Options]) -> {ok, Answer} | Error`

Types:

- Answer = false | [constant()] | [Call] | [Component] | int() | [DefineAt] | [CallAt] | [AllLines]
- Call = call() | ComponentCall
- ComponentCall = {Component, Component}
- Component = [constant()]
- DefineAt = {mfa(), LineNumber}
- CallAt = {funcall(), LineNumbers}
- AllLines = {{DefineAt, DefineAt}, LineNumbers}
- Error = {error, module(), Reason}
- LineNumbers = [LineNumber]
- LineNumber = int()
- Options = [Option] | Option
- Option = {verbose, bool()}
- Query = string() | atom()

- Reason = {invalid_options, term()} | {parse_error, string_position(), term()} | {type_error, string()} | {type_mismatch, string(), string()} | {unknown_analysis, term()} | {unknown_constant, string()} | {unknown_variable, variable()} | {variable_reassigned, string()}
- Xref = xref()

Evaluates a query [page 65] in the context of an Xref server [page 58], and returns the value of the last statement. The syntax of the value depends on the expression:

- A set of calls is represented by a sorted list without duplicates of `call()`.
- A set of constants is represented by a sorted list without duplicates of `constant()`.
- A set of strongly connected components is a sorted list without duplicates of `Component`.
- A set of calls between strongly connected components is a sorted list without duplicates of `ComponentCall`.
- A chain of calls is represented by a list of `constant()`. The list contains the From vertex of each call and the To vertex of the last call.
- The `of` operator returns `false` if no chain of calls between the given constants can be found.
- The value of the `closure` operator (the `digraph` representation) is represented by the atom `'closure()'`.
- A set of line numbered functions is represented by a sorted list without duplicates of `DefineAt`.
- A set of line numbered function calls is represented by a sorted list without duplicates of `CallAt`.
- A set of line numbered functions and function calls is represented by a sorted list without duplicates of `AllLines`.

For both `CallAt` and `AllLines` it holds that for no list element is `LineNumbers` an empty list; such elements have been removed. The constants of `component` and the integers of `LineNumbers` are sorted and without duplicates.

`remove_application(Xref, Applications) -> ok | Error`

Types:

- Applications = application() | [application()]
- Error = {error, module(), Reason}
- Reason = {no_such_application, application()}
- Xref = xref()

Removes applications and their modules and module data [page 58] from an Xref server [page 58].

`remove_module(Xref, Modules) -> ok | Error`

Types:

- Error = {error, module(), Reason}
- Modules = module() | [module()]
- Reason = {no_such_module, module()}
- Xref = xref()

Removes analyzed modules [page 59] and module data [page 58] from an Xref server [page 58].

`remove_release(Xref, Releases) -> ok | Error`

Types:

- Error = {error, module(), Reason}
- Reason = {no_such_release, release()}
- Releases = release() | [release()]
- Xref = xref()

Removes releases and their applications, modules and module data [page 58] from an Xref server [page 58].

`replace_application(Xref, Application, Directory [, Options]) -> {ok, application()} | Error`

Types:

- Application = application()
- Directory = directory()
- Error = {error, module(), Reason}
- Options = [Option] | Option
- Option = {builtins, bool()} | {verbose, bool()} | {warnings, bool()}
- Reason = {no_such_application, application()} | - see also add_application -
- Xref = xref()

Replaces the modules of an application with other modules read from an application directory. Release membership of the application is retained. Note that the name of the application is kept; the name of the given directory is not used.

`replace_module(Xref, Module, File [, Options]) -> {ok, module()} | Error`

Types:

- Error = {error, module(), Reason}
- File = file()
- Module = module()
- Options = [Option] | Option
- Option = {verbose, bool()} | {warnings, bool()}
- ReadModule = module()
- Reason = {module_mismatch, module(), ReadModule} | {no_such_module, module()} | - see also add_module -
- Xref = xref()

Replaces module data [page 58] of an analyzed module [page 59] with data read from a BEAM file. Application membership of the module is retained, and so is the value of the `builtins` option of the module. An error is returned if the name of the read module differs from the given module.

The `update` function is an alternative for updating module data of recompiled modules.

`set_default(Xref, Option, Value) -> {ok, OldValue} | Error`
`set_default(Xref, OptionValues) -> ok | Error`

Types:

- Error = {error, module(), Reason}
- OptionValues = [OptionValue] | OptionValue
- OptionValue = {Option, Value}
- Option = builtins | recurse | verbose | warnings
- Reason = {invalid_options, term()}
- Value = bool()
- Xref = xref()

Sets the default value of one or more options. The options that can be set this way are:

- `builtins`, with initial default value `false`;
- `recurse`, with initial default value `false`;
- `verbose`, with initial default value `false`;
- `warnings`, with initial default value `true`.

The initial default values are set when creating an Xref server [page 58].

`set_library_path(Xref, LibraryPath [, Options]) -> ok | Error`

Types:

- Error = {error, module(), Reason}
- LibraryPath = library_path()
- Options = [Option] | Option
- Option = {verbose, bool()}
- Reason = {invalid_options, term()} | {invalid_path, term()}
- Xref = xref()

Sets the library path [page 59]. If the given path is a list of directories, the set of library modules [page 59] is determined by choosing the first module encountered while traversing the directories in the given order, for those modules that occur in more than one directory. By default, the library path is an empty list.

The library path `code_path` is used by the functions `m/1` and `d/1`, but can also be set explicitly. Note however that the code path will be traversed once for each used library module [page 59] while setting up module data. On the other hand, if there are only a few modules that are used by not analyzed, using `code_path` may be faster than setting the library path to `code:get_path()`.

If the library path is set to `code_path`, the set of library modules is not determined, and the `info` functions will return empty lists of library modules.

`start(Xref [, Options]) -> Return`

Types:

- Options = [Option] | Option
- Option = {xref_mode, mode()} | term()
- Return = {ok, pid()} | {error, {already_started, pid()}}
- Xref = xref()

Creates an Xref server [page 58]. The default mode [page 58] is `functions`. Options that are not recognized by Xref are passed on to `gen_server:start/4`.

`stop(Xref)`

Types:

- Xref = xref()

Stops an Xref server [page 58].

`update(Xref [, Options]) -> {ok, Modules} | Error`

Types:

- Error = {error, module(), Reason}
- Modules = [module()]
- Options = [Option] | Option
- Option = {verbose, bool()} | {warnings, bool()}
- Reason = {invalid_options, term()} | {module_mismatch, module(), ReadModule} | - see also add_module -
- Xref = xref()

Replaces the module data [page 58] of all analyzed modules [page 59] the BEAM files of which have been modified since last read by an `add` function or `update`. Application membership of the modules is retained, and so is the value of the `builtins` option. Returns a sorted list of the names of the replaced modules.

`variables(Xref [, Options]) -> {ok, [VariableInfo]}`

Types:

- Options = [Option] | Option
- Option = predefined | user | {verbose, bool()}
- Reason = {invalid_options, term()}
- VariableInfo = {predefined, [variable()]} | {user, [variable()]}
- Xref = xref()

Returns a sorted lists of the names of the variables of an Xref server [page 58]. The default is to return the user variables [page 60] only.

See Also

beam_lib(3), digraph(3), digraph_utils(3), regexp(3), TOOLS User's Guide [page 14]

List of Figures

Glossary

BIF

Built-In Functions which perform operations that are impossible or inefficient to program in Erlang itself. Are defined inthe module Erlang in the application kernel

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.