

Dialyzer

Copyright © 2006-2017 Ericsson AB. All Rights Reserved.
Dialyzer 2.9
December 1, 2017

Copyright © 2006-2017 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 1, 2017

1 Dialyzer User's Guide

Dialyzer is a static analysis tool that identifies software discrepancies such as type errors, unreachable code, unnecessary tests, etc in single Erlang modules or entire (sets of) applications.

1.1 Dialyzer

1.1.1 Introduction

Dialyzer is a static analysis tool that identifies software discrepancies such as type errors, unreachable code, unnecessary tests, etc in single Erlang modules or entire (sets of) applications.

1.1.2 Using the Dialyzer from the GUI

Choosing the applications or modules

In the "File" window you will find a listing of the current directory. Click your way to the directories/modules you want to add or type the correct path in the entry.

Mark the directories/modules you want to analyze for discrepancies and click "Add". You can either add the `.beam` and `.erl`-files directly, or you can add directories that contain these kinds of files. Note that you are only allowed to add the type of files that can be analyzed in the current mode of operation (see below), and that you cannot mix `.beam` and `.erl`-files.

The analysis modes

Dialyzer has two modes of analysis, "Byte Code" or "Source Code". These are controlled by the buttons in the top-middle part of the main window, under "Analysis Options".

Controlling the discrepancies reported by the Dialyzer

Under the "Warnings" pull-down menu, there are buttons that control which discrepancies are reported to the user in the "Warnings" window. By clicking on these buttons, one can enable/disable a whole class of warnings. Information about the classes of warnings can be found on the "Warnings" item under the "Help" menu (at the rightmost top corner).

If modules are compiled with inlining, spurious warnings may be emitted. In the "Options" menu you can choose to ignore inline-compiled modules when analyzing byte code. When starting from source code this is not a problem since the inlining is explicitly turned off by Dialyzer. The option causes Dialyzer to suppress all warnings from inline-compiled modules, since there is currently no way for Dialyzer to find what parts of the code have been produced by inlining.

Running the analysis

Once you have chosen the modules or directories you want to analyze, click the "Run" button to start the analysis. If for some reason you want to stop the analysis while it is running, push the "Stop" button.

The information from the analysis will be displayed in the Log and the Warnings windows.

Include directories and macro definitions

When analyzing from source you might have to supply Dialyzer with a list of include directories and macro definitions (as you can do with the `erlc` flags `-I` and `-D`). This can be done either by starting Dialyzer with these flags from the command line as in:

```
dialyzer -I my_includes -DDEBUG -Dvsn=42 -I one_more_dir
```

or by adding these explicitly using the "Manage Macro Definitions" or "Manage Include Directories" sub-menus in the "Options" menu.

Saving the information on the Log and Warnings windows

In the "File" menu there are options to save the contents of the Log and the Warnings window. Just choose the options and enter the file to save the contents in.

There are also buttons to clear the contents of each window.

Inspecting the inferred types of the analyzed functions

Dialyzer stores the information of the analyzed functions in a Persistent Lookup Table (PLT). After an analysis you can inspect this information. In the PLT menu you can choose to either search the PLT or inspect the contents of the whole PLT. The information is presented in edoc format.

1.1.3 Using the Dialyzer from the command line

See *dialyzer(3)*.

1.1.4 Using the Dialyzer from Erlang

See *dialyzer(3)*.

1.1.5 More on the Persistent Lookup Table (PLT)

The persistent lookup table, or PLT, is used to store the result of an analysis. The PLT can then be used as a starting point for later analyses. It is recommended to build a PLT with the otp applications that you are using, but also to include your own applications that you are using frequently.

The PLT is built using the --build_plt option to dialyzer. The following command builds the recommended minimal PLT for OTP.

```
dialyzer --build_plt -r $ERL_TOP/lib/stdlib/ebin\
                        $ERL_TOP/lib/kernel/ebin \
                        $ERL_TOP/lib/mnesia/ebin
```

Dialyzer will look if there is an environment variable called $DIALYZER_PLT and place the PLT at this location. If no such variable is set, Dialyzer will place the PLT at $HOME/.dialyzer_plt. The placement can also be specified using the --plt, or --output_plt options.

You can also add information to an existing plt using the --add_to_plt option. Suppose you want to also include the compiler in the PLT and place it in a new PLT, then give the command

```
dialyzer --add_to_plt -r $ERL_TOP/lib/compiler/ebin --output_plt my.plt
```

Then you would like to add your favorite application my_app to the new plt.

```
dialyzer --add_to_plt --plt my.plt -r my_app/ebin
```

But you realize that it is unnecessary to have compiler in this one.

```
dialyzer --remove_from_plt --plt my.plt -r $ERL_TOP/lib/compiler/ebin
```

Later, when you have fixed a bug in your application my_app, you want to update the plt so that it will be fresh the next time you run Dialyzer, run the command

```
dialyzer --check_plt --plt my.plt
```

Dialyzer will then reanalyze the files that have been changed, and the files that depend on these files. Note that this consistency check will be performed automatically the next time you run Dialyzer with this plt. The --check_plt option is merely for doing so without doing any other analysis.

To get some information about a plt use the option

```
dialyzer --plt_info
```

You can also specify which plt with the --plt option, and get the output printed to a file with --output_file

Note that when manipulating the plt, no warnings are emitted. To turn on warnings during (re)analysis of the plt, use the option --get_warnings.

1.1.6 Feedback and bug reports

At this point, we very much welcome user feedback (even wish-lists!). If you notice something weird, especially if the Dialyzer reports any discrepancy that is a false positive, please send an error report describing the symptoms and how to reproduce them to:

```
tobias.lindahl@it.uu.se, kostis@it.uu.se
```

2 Reference Manual

Dialyzer is a static analysis tool that identifies software discrepancies such as type errors, unreachable code, unnecessary tests, etc in single Erlang modules or entire (sets of) applications.

dialyzer

Erlang module

The Dialyzer is a static analysis tool that identifies software discrepancies such as definite type errors, code which has become dead or unreachable due to some programming error, unnecessary tests, etc. in single Erlang modules or entire (sets of) applications. Dialyzer starts its analysis from either debug-compiled BEAM bytecode or from Erlang source code. The file and line number of a discrepancy is reported along with an indication of what the discrepancy is about. Dialyzer bases its analysis on the concept of success typings which allows for sound warnings (no false positives).

Read more about Dialyzer and about how to use it from the GUI in *Dialyzer User's Guide*.

Using the Dialyzer from the command line

Dialyzer also has a command line version for automated use. Below is a brief description of the list of its options. The same information can be obtained by writing

```
dialyzer --help
```

in a shell. Please refer to the GUI description for more details on the operation of Dialyzer.

The exit status of the command line version is:

```
0 - No problems were encountered during the analysis and no
    warnings were emitted.
1 - Problems were encountered during the analysis.
2 - No problems were encountered, but warnings were emitted.
```

Usage:

```
 dialyzer [--help] [--version] [--shell] [--quiet] [--verbose]
[-pa dir]* [--plt plt] [--plts plt*] [-Ddefine]*
          [-I include_dir]* [--output_plt file] [-Wwarn]* [--raw]
          [--src] [--gui] [files_or_dirs] [-r dirs]
          [--apps applications] [-o outfile]
[--build_plt] [--add_to_plt] [--remove_from_plt]
[--check_plt] [--no_check_plt] [--plt_info] [--get_warnings]
          [--dump_callgraph file] [--no_native] [--fullpath]
          [--statistics] [--no_native_cache]
```

Options:

`files_or_dirs` (for backwards compatibility also as: `-c files_or_dirs`)
: Use Dialyzer from the command line to detect defects in the specified files or directories containing `.erl` or `.beam` files, depending on the type of the analysis.

`-r dirs`
: Same as the previous but the specified directories are searched recursively for subdirectories containing `.erl` or `.beam` files in them, depending on the type of analysis.

`--apps applications`
: Option typically used when building or modifying a plt as in:

```
dialyzer --build_plt --apps erts kernel stdlib mnesia ...
```

to conveniently refer to library applications corresponding to the Erlang/OTP installation. However, the option is general and can also be used during analysis in order to refer to Erlang/OTP applications. In addition, file or directory names can also be included, as in:

```
dialyzer --apps inets ssl ./ebin ../other_lib/ebin/my_module.beam
```

`-o outfile` (or `--output outfile`)
When using Dialyzer from the command line, send the analysis results to the specified outfile rather than to stdout.

`--raw`
When using Dialyzer from the command line, output the raw analysis results (Erlang terms) instead of the formatted result. The raw format is easier to post-process (for instance, to filter warnings or to output HTML pages).

`--src`
Override the default, which is to analyze BEAM files, and analyze starting from Erlang source code instead.

`-Dname` (or `-Dname=value`)
When analyzing from source, pass the define to Dialyzer. (**)

`-I include_dir`
When analyzing from source, pass the `include_dir` to Dialyzer. (**)

`-pa dir`
Include `dir` in the path for Erlang (useful when analyzing files that have `'-include_lib()'` directives).

`--output_plt file`
Store the plt at the specified file after building it.

`--plt plt`
Use the specified plt as the initial plt (if the plt was built during setup the files will be checked for consistency).

`--plts plt*`
Merge the specified plts to create the initial plt -- requires that the plts are disjoint (i.e., do not have any module appearing in more than one plt). The plts are created in the usual way:

```
dialyzer --build_plt --output_plt plt_1 files_to_include
...
dialyzer --build_plt --output_plt plt_n files_to_include
```

and then can be used in either of the following ways:

```
dialyzer files_to_analyze --plts plt_1 ... plt_n
```

or:

```
dialyzer --plts plt_1 ... plt_n -- files_to_analyze
```

(Note the -- delimiter in the second case)

`-Wwarn`
: A family of options which selectively turn on/off warnings (for help on the names of warnings use `dialyzer -Whelp`). Note that the options can also be given in the file with a `-dialyzer()` attribute. See *Requesting or Suppressing Warnings in Source Files* below for details.

`--shell`
: Do not disable the Erlang shell while running the GUI.

`--version` (or `-v`)
: Print the Dialyzer version and some more information and exit.

`--help` (or `-h`)
: Print this message and exit.

`--quiet` (or `-q`)
: Make Dialyzer a bit more quiet.

`--verbose`
: Make Dialyzer a bit more verbose.

`--statistics`
: Prints information about the progress of execution (analysis phases, time spent in each and size of the relative input).

`--build_plt`
: The analysis starts from an empty plt and creates a new one from the files specified with `-c` and `-r`. Only works for beam files. Use `--plt` or `--output_plt` to override the default plt location.

`--add_to_plt`
: The plt is extended to also include the files specified with `-c` and `-r`. Use `--plt` to specify which plt to start from, and `--output_plt` to specify where to put the plt. Note that the analysis might include files from the plt if they depend on the new files. This option only works with beam files.

`--remove_from_plt`
: The information from the files specified with `-c` and `-r` is removed from the plt. Note that this may cause a re-analysis of the remaining dependent files.

`--check_plt`
: Check the plt for consistency and rebuild it if it is not up-to-date.

`--no_check_plt`
: Skip the plt check when running Dialyzer. Useful when working with installed plts that never change.

`--plt_info`
: Make Dialyzer print information about the plt and then quit. The plt can be specified with `--plt(s)`.

`--get_warnings`
: Make Dialyzer emit warnings even when manipulating the plt. Warnings are only emitted for files that are actually analyzed.

`--dump_callgraph file`
: Dump the call graph into the specified file whose format is determined by the file name extension. Supported extensions are: raw, dot, and ps. If something else is used as file name extension, default format '.raw' will be used.

`--no_native` (or `-nn`)
: Bypass the native code compilation of some key files that Dialyzer heuristically performs when dialyzing many files; this avoids the compilation time but it may result in (much) longer analysis time.

`--no_native_cache`
: By default, Dialyzer caches the results of native compilation in the `$XDG_CACHE_HOME/erlang/dialyzer_hipe_cache` directory. `XDG_CACHE_HOME` defaults to `$HOME/.cache`. Use this option to disable caching.

`--fullpath`
: Display the full path names of files for which warnings are emitted.

`--gui`
: Use the GUI.

> Note:
>
> * denotes that multiple occurrences of these options are possible.
>
> ** options `-D` and `-I` work both from command-line and in the Dialyzer GUI; the syntax of defines and includes is the same as that used by `erlc`.

Warning options:

`-Wno_return`
: Suppress warnings for functions that will never return a value.

`-Wno_unused`
: Suppress warnings for unused functions.

`-Wno_improper_lists`
: Suppress warnings for construction of improper lists.

`-Wno_fun_app`
: Suppress warnings for fun applications that will fail.

`-Wno_match`
: Suppress warnings for patterns that are unused or cannot match.

`-Wno_opaque`
: Suppress warnings for violations of opaqueness of data types.

`-Wno_fail_call`
: Suppress warnings for failing calls.

`-Wno_contracts`
: Suppress warnings about invalid contracts.

`-Wno_behaviours`
: Suppress warnings about behaviour callbacks which drift from the published recommended interfaces.

`-Wno_missing_calls`
: Suppress warnings about calls to missing functions.

`-Wno_undefined_callbacks`
: Suppress warnings about behaviours that have no `-callback` attributes for their callbacks.

`-Wunmatched_returns`***
: Include warnings for function calls which ignore a structured return value or do not match against one of many possible return value(s).

`-Werror_handling`***
: Include warnings for functions that only return by means of an exception.

`-Wrace_conditions`***
: Include warnings for possible race conditions. Note that the analysis that finds data races performs intra-procedural data flow analysis and can sometimes explode in time. Enable it at your own risk.

i

`-Wunderspecs`***
: Warn about underspecified functions (the -spec is strictly more allowing than the success typing).

`-Wunknown`***
: Let warnings about unknown functions and types affect the exit status of the command line version. The default is to ignore warnings about unknown functions and types when setting the exit status. When using the Dialyzer from Erlang, warnings about unknown functions and types are returned; the default is not to return these warnings.

The following options are also available but their use is not recommended: (they are mostly for Dialyzer developers and internal debugging)

`-Woverspecs`***
: Warn about overspecified functions (the -spec is strictly less allowing than the success typing).

`-Wspecdiffs***`
Warn when the -spec is different than the success typing.

> **Note:**
> *** Identifies options that turn on warnings rather than turning them off.

Using the Dialyzer from Erlang

You can also use Dialyzer directly from Erlang. Both the GUI and the command line versions are available. The options are similar to the ones given from the command line, so please refer to the sections above for a description of these.

Requesting or Suppressing Warnings in Source Files

The `-dialyzer()` attribute can be used for turning off warnings in a module by specifying functions or warning options. For example, to turn off all warnings for the function `f/0`, include the following line:

```
-dialyzer({nowarn_function, f/0}).
```

To turn off warnings for improper lists, add the following line to the source file:

```
-dialyzer(no_improper_lists).
```

The `-dialyzer()` attribute is allowed after function declarations. Lists of warning options or functions are allowed:

```
-dialyzer([{nowarn_function, [f/0]}, no_improper_lists]).
```

Warning options can be restricted to functions:

```
-dialyzer({no_improper_lists, g/0}).
```

```
-dialyzer({[no_return, no_match], [g/0, h/0]}).
```

For help on the warning options use `dialyzer -Whelp`. The options are also enumerated *below* (`WarnOpts`).

> **Note:**
> The `-dialyzer()` attribute is not checked by the Erlang Compiler, but by the Dialyzer itself.

> Note:
>
> The warning option `-Wrace_conditions` has no effect when set in source files.

The `-dialyzer()` attribute can also be used for turning on warnings. For instance, if a module has been fixed regarding unmatched returns, adding the line

```
-dialyzer(unmatched_returns).
```

can help in assuring that no new unmatched return warnings are introduced.

Exports

`gui() -> ok | {error, Msg}`
`gui(OptList) -> ok | {error, Msg}`

Types:

`OptList -- see below`

Dialyzer GUI version.

```
OptList  :: [Option]
Option   :: {files,          [Filename :: string()]}
          | {files_rec,      [DirName :: string()]}
          | {defines,        [{Macro :: atom(), Value :: term()}]}
          | {from,           src_code | byte_code}   %% Defaults to byte_code
          | {init_plt,       FileName :: string()}   %% If changed from default
          | {plts,           [FileName :: string()]} %% If changed from default
          | {include_dirs,   [DirName :: string()]}
          | {output_file,    FileName :: string()}
          | {output_plt,     FileName :: string()}
          | {check_plt,      boolean()},
          | {analysis_type,  'succ_typings' |
                             'plt_add' |
                             'plt_build' |
                             'plt_check' |
                             'plt_remove'}
          | {warnings,       [WarnOpts]}
          | {get_warnings,   bool()}

WarnOpts :: no_return
          | no_unused
          | no_improper_lists
          | no_fun_app
          | no_match
          | no_opaque
          | no_fail_call
          | no_contracts
          | no_behaviours
          | no_undefined_callbacks
          | unmatched_returns
          | error_handling
          | race_conditions
          | overspecs
          | underspecs
          | specdiffs
```

```
          | unknown
```

run(OptList) -> Warnings

Types:

OptList -- see gui/0,1

Warnings -- see below

Dialyzer command line version.

```
Warnings :: [{Tag, Id, Msg}]
Tag :: 'warn_behaviour'
     | 'warn_bin_construction'
     | 'warn_callgraph'
     | 'warn_contract_not_equal'
     | 'warn_contract_range'
     | 'warn_contract_subtype'
     | 'warn_contract_supertype'
     | 'warn_contract_syntax'
     | 'warn_contract_types'
     | 'warn_failing_call'
     | 'warn_fun_app'
     | 'warn_matching'
     | 'warn_non_proper_list'
     | 'warn_not_called'
     | 'warn_opaque'
     | 'warn_race_condition'
     | 'warn_return_no_exit'
     | 'warn_return_only_exit'
     | 'warn_umatched_return'
     | 'warn_undefined_callbacks'
     | 'warn_unknown'
Id = {File :: string(), Line :: integer()}
Msg = msg() -- Undefined
```

format_warning(Msg) -> string()

Types:

Msg = {Tag, Id, msg()} -- See run/1

Get a string from warnings as returned by dialyzer:run/1.

plt_info(string()) -> {'ok', [{atom(), any()}]} | {'error', atom()}

Returns information about the specified plt.