Dialyzer Application (DIALYZER)

version 1.7

Typeset in LaTeX from SGML source using the DocBuilder-0.9 Document System.

Contents

1 Dialyzer User's Guide 1
1.1 Dialyzer 1
1.1.1 Introduction 1
1.1.2 Using the Dialyzer from the GUI 1
1.1.3 Using the Dialyzer from the command line 3
1.1.4 Using the Dialyzer from Erlang 3
1.1.5 More on the Persistent Lookup Table (PLT) 3
1.1.6 Feedback and bug reports 4

2 Dialyzer Reference Manual 5
2.1 dialyzer 6

Chapter 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 several modes of analysis. These are controlled by the buttons in the top-middle part of the main window, under "Analysis Options".

The parameters are:

Analyze: Byte code: The analysis starts from `.beam` bytecode files. Whenever the `.beam` file has been generated with the `+debug_info` compiler option on, analysis automatically starts from abstract code (via Core Erlang) and the results are identical to those obtained starting from source code.

Source code: The analysis starts from `.erl` files.

Granularity: You can choose to Analyze each module locally, or to make the analysis global over all modules. The default (and recommended mode) is a global analysis, but the module-local variant can be handy when the interfaces between modules are not yet fully implemented and you are simply interested in some quick-and-dirty feedback.

Iteration: Here you can specify if the analysis should perform a fixpoint iteration over the chosen granularity, or make just one pass.

One Pass: One pass iteration analyzes all modules in an unspecified order. Each module is still analyzed to fixpoint, but no fixpoint iteration is performed over module boundaries.

Fixpoint: All specified modules are analyzed until fixpoint. To speed up the iteration, a static call graph for inter-modular calls is constructed after the first pass of the analysis. This call graph is then used to order the modules for the second (and final) run of the analysis.

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.

> **Note:**
> Currently, the information which is displayed is NOT the type signatures of the functions. The return values are the least upper bound of the returned type from the function and the argument types are the least upper bound of the types that the function is called with. In other words, the argument types is not what the function can accept, but rather a description of how the function is used.

We are working on finding the type signatures of the function, and this will (hopefully) be included in a future version of Dialyzer.

1.1.3 Using the Dialyzer from the command line

See dialyzer(3) [page 6].

1.1.4 Using the Dialyzer from Erlang

See dialyzer(3) [page 6].

1.1.5 More on the Persistent Lookup Table (PLT)

During setup, a Persistent Lookup Table will automatically be created for the Erlang/OTP libraries specified in `dialyzer/src/Makefile`. This table will be the starting point for later analyses. At each startup of Dialyzer the validity of the PLT will be checked, and if something has changed in the included libraries a new PLT will be constructed.

To build a PLT of your own favorite files use the `--output_plt` option to specify where the file containing the PLT should be stored. At later analyses this PLT can be used as the starting point by using the `--plt` option. Note that the new PLT file also includes the information from the PLT that was used as the starting point

> **Warning:**
> You should not analyze files that are already included in the PLT. This can lead to unexpected results if some file has changed since the PLT was built. Such dependencies are currently only checked when the PLT was built using the default libraries in `dialyzer/src/Makefile`.

In an Erlang/OTP system which has been installed, the user typically does not have write permission to the file of the PLT. If Dialyzer later finds that the PLT is not up-to-date, the analysis is aborted with a warning. Until the installed PLT has been updated a temporary PLT can be created and used in the manner described above.

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
```

Dialyzer Reference Manual

Short Summaries

- Erlang Module dialyzer [page 6] – The Dialyzer, a DIscrepancy AnalYZer for ERlang programs

dialyzer

The following functions are exported:

- `gui() -> ok | {error, Msg}`
  [page 8] Dialyzer GUI version
- `gui(OptList) -> ok | {error, Msg}`
  [page 8] Dialyzer GUI version
- `run(OptList) -> {ok, Warnings, Errors} | {ok, Warnings} | {error, Message}`
  [page 8] Dialyzer command line version

dialyzer

Erlang Module

The 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. Currently, Dialyzer starts its analysis from either BEAM bytecode or from Erlang source code and reports to its user the functions where the discrepancies occur and an indication of what the discrepancy is about. Dialyzer currently supports various modes of operation and its analysis is precise (in particular, there are no false positives) and quite fast.

Read more about Dialyzer and about how to use it from the GUI in Dialyzer User's Guide [page 1].

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] [-Ddefine]* [-I include_dir]*
         [--output_plt file] [-Wwarn]*  [--src]
         [-c applications] [-r applications] [-o outfile]
```

Options:

`-c applications` (or `--command-line applications`) use Dialyzer from the command line (no GUI) to detect defects in the specified applications (directories or `.erl` or `.beam` files)

`-r applications` same as `-c` only that directories are searched recursively for subdirectories containing `.erl` or `.beam` files (depending on the type of analysis)

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

`--src` overwrite the default, which is to analyze BEAM bytecode, 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 location after building it.

`--no_warn_on_inline` Suppress warnings when analyzing an inline compiled bytecode file.

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

`-Wwarn` a family of option which selectively turn on/off warnings. (for help on the names of warnings use `dialyzer -Whelp`)

`--check_init_plt` Only checks if the init plt is up to date. For installed systems this also forces the rebuilding of the plt if this is not the case.

`--shell` do not disable the Erlang shell while running the GUI

`--version (or -v)` prints the Dialyzer version and some more information and exits

`--help (or -h)` prints this message and exits

`--quiet (or -q)` makes Dialyzer a bit more quiet

`--verbose` makes Dialyzer a bit more verbose

`--dataflow` Makes Dialyzer use dataflow analysis to find discrepancies. (Default)

`--succ_typings` Makes Dialyzer use success typings to find discrepancies.

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 of no return.

`-Wno_unused` Suppress warnings for unused functions.

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

`-Wno_tuple_as_fun` Suppress warnings for using tuples instead of funs.

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

`-Wno_match` Suppress warnings for pattern matching operations that will never succeed.

`-Wno_comp` Suppress warnings for term comparisons that will always return `false`.

`-Wno_guards` Suppress warnings for guards that will always fail.

`-Wno_unsafe_beam` Suppress warnings for unsafe BEAM code produced by an old BEAM compiler.

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

> Note:
> *** This is the only option that turns 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 version is 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.

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
         | {include_dirs,   [DirName : string()]}
         | {output_file,    FileName : string()}
         | {supress_inline, bool()}               %% Defaults to true
         | {warnings,       [WarnOpts]}

WarnOpts : no_return
         | no_unused
         | no_improper_lists
         | no_tuple_as_fun
         | no_fun_app
         | no_match
         | no_comp
         | no_guards
         | no_unsafe_beam
         | no_fail_call
         | error_handling
```

```
run(OptList) -> {ok, Warnings, Errors} | {ok, Warnings} | {error, Message}
```

Types:

- OptList – see gui/0,1
- Warnings = [{MFA, string()}]
- MFA = {Module, Function, Arity}

- Module = Function = atom()
- Arity = int()
- Errors = string()
- Message = string()

Dialyzer command line version.

Index of Modules and Functions

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

dialyzer
`gui/0`, 8
`gui/1`, 8
`run/1`, 8

`gui/0`
dialyzer , 8

`gui/1`
dialyzer , 8

`run/1`
dialyzer , 8