Dialyzer Application (DIALYZER)

version 1.4

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

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 supress 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.

More on the Persistent Lookup Table (PLT)

During setup, a Persistent Lookup Table will automatically be created for the Erlang/OTP standard library (stdlib). 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 stdlib a new PLT will be constructed.

If you want to start from a completely fresh table at each analysis choose “Init with empty plt” from the same menu.

Dialyzer Reference Manual

Short Summaries

- Command dialyzer [page 6] – Erlang Dialyzer

dialyzer

No functions are exported.

dialyzer

Command

This is the 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 in the User's Guide, 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 [--otp OTP_DIR] [--help] [--version] [--shell]
                [-pa dir]* [-plt plt] [-Ddefine]* [-I include_dir]*
                [--output_plt file] [-Wwarn]*  [--src]
                [-c applications] [-r applications] [-o outfile] [-q]

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 (**)
   --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.
  -pa dir
      Include dir in the path for Erlang. Useful when analyzing files
      that have '-include_lib()' directives.
  -Wwarn
      a family of option which selectively turn on/off warnings.
       (for help on the names of warnings use dialyzer -Whelp)
  --otp OTP_DIR
      overrides the default location of the Erlang/OTP system to use
  --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
  -q
      makes Dialyzer a bit more quiet
```

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.
```

Identified discrepancies

The discrepancies currently identified by Dialyzer can be classified in the following categories:

TYPE ERRORS

- Match failure
  - Warnings:
    - “The clause matching on X will never match; argument is of type T”
    - “The clause matching on tuple with arity N will never match; ”
    - “ argument is of type T!”
  - Description:
    - The function or case clause will never match since the calling argument has a different type than the expected one. Note that due to pattern-matching compilation the X above may be an argument enclosed in some structured term (tuple or list).
- Function call with wrong arguments
  - Warning:
    - “Call to function X with signature S will fail since the arguments are of type T!”
  - Description:
    - The arguments which the function is called with are not what the function implicitly expects.
- Closure of wrong type
  - Warnings:
    - “Fun application using type T instead of a fun!”
    - “Trying to use fun with type T with arguments AT”
  - Description:
    - The variable that is used in the fun application is either not a closure (fun entry) or a closure with the wrong domain.
- Improper list construction
  - Warnings:
    - “Cons will produce a non-proper list since its 2nd arg is of type T!”
    - “Call to '++'/2 will produce a non-proper list since its 2nd arg is of type T!”
  - Description:
    - This is a place where an improper list (i.e., a list whose last element is not the empty list []) is constructed. Strictly, these are not discrepancies in Erlang, but we strongly recommend that you fix these; there is ABSOLUTELY NO reason to create improper lists.
- Function of no return
  - Warning:
    - “Function will never return a proper value!”

  - Description:
    * This is a function that never returns. Strictly speaking, this is not a function and the code is OK only if this is used as a point where an exception is thrown when handling an error.

REDUNDANT OR UNREACHABLE CODE

- Unreachable case clause
  - Warning:
    * "Type guard X will always fail since variable is of type T!"
  - Description:
    * The case clause is redundant since the input argument is of a different type.
- Unreachable function clause
  - Warning:
    * "The guard X will always fail since the arguments are of type T!"
  - Description:
    * The clause is made redundant due to one of its guards always failing.
- Term comparison failure
  - Warnings:
    * "=:= between T1 and T2 will always fail!"
    * "=/= between T1 and T2 will always fail!"
  - Description:
    * The term comparison will always fail making the test always return 'false' or, in a guard context, making the clause redundant.
- Unused function
  - Warning:
    * "Function will never be called!"
  - Description:
    * The function is unused; no need to have it uncommented in the code.

CODE RELICS

- Tuple used as fun
  - Warnings:
    * "Unsafe use of tuple as a fun in call to X"
    * "Tuple used as fun will fail in native compiled code"
  - Description:
    * A 2-tuple is used as a function closure. The modern way of calling higher-order code in Erlang is by using proper funs. The code should be rewritten using a proper 'fun' rather than a 2-tuple since among other things makes the code cleaner and is safer for execution in native code.
- Unsafe BEAM code
  - Warning:
    * "Unsafe BEAM code! Please recompile with a newer BEAM compiler."
  - Description:
    * The analysis has encountered BEAM bytecode which will fail in a really bad way (even with a seg-fault) if used in an impoper way. Such code was produced by the BEAM compiler of R9C-0 (and prior) for some record expressions. The recommended action is to generate a new .beam file using a newer version of the BEAM compiler.

Feedback & 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