

Dialyzer

Copyright © 2006-2026 Ericsson AB. All Rights Reserved.
Dialyzer 5.1.3.1
October 31, 2026

Copyright © 2006-2026 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..

October 31, 2026

1 Dialyzer User's Guide

1.1 Dialyzer

1.1.1 Introduction

Scope

Dialyzer is a static analysis tool that identifies software discrepancies, such as definite type errors, code that has become dead or unreachable because of programming error, and unnecessary tests, in single Erlang modules or entire (sets of) applications.

Dialyzer can be called from the command line and from Erlang.

Prerequisites

It is assumed that the reader is familiar with the Erlang programming language.

1.1.2 The Persistent Lookup Table

Dialyzer stores the result of an analysis in a Persistent Lookup Table (PLT). The PLT can then be used as a starting point for later analyses. It is recommended to build a PLT with the Erlang/OTP applications that you are using, but also to include your own applications that you are using frequently.

The PLT is built using option `--build_plt` to Dialyzer. The following command builds the recommended minimal PLT for Erlang/OTP:

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

Dialyzer looks if there is an environment variable called `DIALYZER_PLT` and places the PLT at this location. If no such variable is set, Dialyzer places the PLT in a file called .dialyzer_plt in the `filename:basedir(user_cache, "erlang")` folder. The placement can also be specified using the options `--plt` or `--output_plt`.

Information can be added to an existing PLT using option `--add_to_plt`. If you also want to include the Erlang compiler in the PLT and place it in a new PLT, then use the following command:

```
dialyzer --add_to_plt --apps compiler --output_plt my.plt
```

Then you can 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 the Erlang compiler in this one:

```
dialyzer --remove_from_plt --plt my.plt --apps compiler
```

Later, when you have fixed a bug in your application my_app, you want to update the PLT so that it becomes fresh the next time you run Dialyzer. In this case, run the following command:

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

Dialyzer then reanalyzes the changed files and the files that depend on these files. Notice that this consistency check is performed automatically the next time you run Dialyzer with this PLT. Option `--check_plt` is only for doing so without doing any other analysis.

To get information about a PLT, use the following option:

```
dialyzer --plt_info
```

To specify which PLT, use option `--plt`.

To get the output printed to a file, use option `--output_file`.

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

1.1.3 Using Dialyzer from the Command Line

Dialyzer has a command-line version for automated use. See `dialyzer(3)`.

1.1.4 Using Dialyzer from Erlang

Dialyzer can also be used directly from Erlang. See `dialyzer(3)`.

1.1.5 Dialyzer's Model of Analysis

Dialyzer operates somewhere between a classical type checker and a more general static-analysis tool: It checks and consumes function specs, yet doesn't require them, and it can find bugs across modules which consider the dataflow of the programs under analysis. This means Dialyzer can find genuine bugs in complex code, and is pragmatic in the face of missing specs or limited information about the codebase, only reporting issues which it can prove have the potential to cause a genuine issue at runtime. This means Dialyzer will sometimes not report every bug, since it cannot always find this proof.

How Dialyzer Utilises Function Specifications

Dialyzer infers types for all top-level functions in a module. If the module also has a spec given in the source-code, Dialyzer will compare the inferred type to the spec. The comparison checks, for each argument and the return, that the inferred and specified types overlap - which is to say, the types have at least one possible runtime value in common. Notice that Dialyzer does not check that one type contains a subset of values of the other, or that they're precisely equal: This allows Dialyzer to make simplifying assumptions to preserve performance and avoid reporting program flows which could potentially succeed at runtime.

If the inferred and specified types do not overlap, Dialyzer will warn that the spec is invalid with respect to the implementation. If they do overlap, however, Dialyzer will proceed under the assumption that the correct type for the given function is the intersection of the inferred type and the specified type (the rationale being that the user may know something that Dialyzer itself cannot deduce). One implication of this is that if the user gives a spec for a function which overlaps with Dialyzer's inferred type, but is more restrictive, Dialyzer will trust those restrictions. This may then generate an error elsewhere which follows from the erroneously restricted spec.

Examples:

Non-overlapping argument:

```
-spec foo(boolean()) -> string().
%% Dialyzer will infer: foo(integer()) -> string().
foo(N) ->
    integer_to_list(N).
```

Since the type of the argument in the spec is different from the type that Dialyzer inferred, Dialyzer will generate the following warning:

```
some_module.erl:7:2: Invalid type specification for function some_module:foo/1.
 The success typing is t:foo
         (integer()) -> string()
 But the spec is t:foo
         (boolean()) -> string()
 They do not overlap in the 1st argument
```

Non-overlapping return:

```
-spec bar(a | b) -> atom().
%% Dialyzer will infer: bar(a | b) -> binary().
bar(a) -> <<"a">>;
bar(b) -> <<"b">>.
```

Since the return value in the spec and the return value inferred by Dialyzer are different, Dialyzer will generate the following warning:

```
some_module.erl:11:2: Invalid type specification for function some_module:bar/1.
 The success typing is t:bar
         ('a' | 'b') -> <<_:8>>
 But the spec is t:bar
         ('a' | 'b') -> atom()
 The return types do not overlap
```

Overlapping spec and inferred type:

```
-spec baz(a | b) -> non_neg_integer().
%% Dialyzer will infer: baz(b | c | d) -> -1 | 0 | 1.
baz(b) -> -1;
baz(c) -> 0;
baz(d) -> 1.
```

Dialyzer will "trust" the spec and using the intersection of the spec and inferred type:

```
baz(b) -> 0 | 1.
```

Notice how the `c` and `d` from the argument to `baz/1` and the `-1` in the return from the inferred type were dropped once the spec and inferred type were intersected. This could result in warnings being emitted for later functions.

For example, if `baz/1` is called like this:

```
call_baz1(A) ->
    case baz(A) of
        -1 -> negative;
        0 -> zero;
        1 -> positive
    end.
```

Dialyzer will generate the following warning:

```
some_module.erl:25:9: The pattern
         -1 can never match the type
         0 | 1
```

If `baz/1` is called like this:

```
call_baz2() ->
    baz(a).
```

Dialyzer will generate the following warnings:

```
some_module.erl:30:1: Function call_baz2/0 has no local return
some_module.erl:31:9: The call t:baz
         ('a') will never return since it differs in the 1st argument
               from the success typing arguments:
         ('b' | 'c' | 'd')
```

1.1.6 Feedback and Bug Reports

We very much welcome user feedback - even wishlists! If you notice anything weird, especially if Dialyzer reports any discrepancy that is a false positive, please send an error report describing the symptoms and how to reproduce them.

2 Reference Manual

dialyzer

Erlang module

Dialyzer is a static analysis tool that identifies software discrepancies, such as definite type errors, code that has become dead or unreachable because of programming errors, and unnecessary tests, 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).

Using Dialyzer from the Command Line

Dialyzer has a command-line version for automated use. This section provides a brief description of the options. The same information can be obtained by writing the following in a shell:

```
dialyzer --help
```

Exit status of the command-line version:

`0`

No problems were found during the analysis and no warnings were emitted.

`1`

Problems were found during the analysis.

`2`

No problems were found during the analysis, but warnings were emitted.

Usage:

```
dialyzer [--add_to_plt] [--apps applications] [--build_plt]
         [--check_plt] [-Ddefine]* [-Dname]* [--dump_callgraph file]
         [--error_location flag] [files_or_dirs] [--fullpath]
         [--get_warnings] [--help] [-I include_dir]*
         [--incremental] [--metrics_file] [--no_check_plt] [--no_indentation]
         [--no_spec] [-o outfile] [--output_plt file] [-pa dir]* [--plt plt]
         [--plt_info] [--plts plt*] [--quiet] [-r dirs] [--raw]
         [--remove_from_plt] [--shell] [--src] [--statistics] [--verbose]
         [--version] [--warning_apps applications] [-Wwarn]*
```

Note:

* denotes that multiple occurrences of the option are possible.

Options of the command-line version:

`--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. Notice that the analysis possibly can include files from the PLT if they depend on the new files. This option only works for BEAM files.

`--apps applications`

By default, warnings will be reported to all applications given by `--apps`. However, if `--warning_apps` is used, only those applications given to `--warning_apps` will have warnings reported. All applications given

by `--apps`, but not `--warning_apps`, will be analysed to provide context to the analysis, but warnings will not be reported for them. For example, you may want to include libraries you depend on in the analysis with `--apps` so discrepancies in their usage can be found, but only include your own code with `--warning_apps` so that discrepancies are only reported in code that you own.

`--warning_apps applications`

This option is typically used when building or modifying a PLT as in:

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

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

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

`--build_plt`

The analysis starts from an empty PLT and creates a new one from the files specified with `-c` and `-r`. This option only works for BEAM files. To override the default PLT location, use `--plt` or `--output_plt`.

`--check_plt`

Check the PLT for consistency and rebuild it if it is not up-to-date.

`-Dname` (or `-Dname=value`)

When analyzing from source, pass the define to Dialyzer. (**)

`--dump_callgraph file`

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

`--error_location column | line`

Use a pair `{Line, Column}` or an integer `Line` to pinpoint the location of warnings. The default is to use a pair `{Line, Column}`. When formatted, the line and the column are separated by a colon.

`files_or_dirs` (for backward 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.

`--fullpath`

Display the full path names of files for which warnings are emitted.

`--get_warnings`

Make Dialyzer emit warnings even when manipulating the PLT. Warnings are only emitted for files that are analyzed.

`--help` (or `-h`)

Print this message and exit.

`-I include_dir`

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

`--input_list_file file`

Analyze the file names that are listed in the specified file (one file name per line).

`--no_check_plt`

Skip the PLT check when running Dialyzer. This is useful when working with installed PLTs that never change.

`--incremental`

The analysis starts from an existing incremental PLT, or builds one from scratch if one does not exist, and runs the minimal amount of additional analysis to report all issues in the given set of apps. Notably, incremental PLT files are not compatible with "classic" PLT files, and vice versa. The initial incremental PLT will be updated unless an alternative output incremental PLT is given.

`--no_indentation`

Do not insert line breaks in types, contracts, and Erlang Code when formatting warnings.

`--no_spec`

Ignore functions specs. This is useful for debugging when one suspects that some specs are incorrect.

`-o outfile` (or `--output outfile`)

When using Dialyzer from the command line, send the analysis results to the specified outfile rather than to `stdout`.

`--metrics_file file`

Write metrics about Dialyzer's incrementality (for example, total number of modules considered, how many modules were changed since the PLT was last updated, how many modules needed to be analyzed) to a file. This can be useful for tracking and debugging Dialyzer's incrementality.

`--output_plt file`

Store the PLT at the specified file after building it.

`-pa dir`

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

`--plt plt`

Use the specified PLT as the initial PLT. If the PLT was built during setup, the files are checked for consistency.

`--plt_info`

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

`--plts plt*`

Merge the specified PLTs to create the initial PLT. This requires that the PLTs are disjoint (that is, 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
```

They can then 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
```

Notice the `--` delimiter in the second case.

`--quiet` (or `-q`)

Make Dialyzer a bit more quiet.

`-r dirs`

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

`--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 example, to filter warnings or to output HTML pages).

`--remove_from_plt`

The information from the files specified with `-c` and `-r` is removed from the PLT. Notice that this can cause a reanalysis of the remaining dependent files.

`--src`

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

`--statistics`

Print information about the progress of execution (analysis phases, time spent in each, and size of the relative input).

`--verbose`

Make Dialyzer a bit more verbose.

`--version` (or `-v`)

Print the Dialyzer version and some more information and exit.

`-Wwarn`

A family of options that selectively turn on/off warnings. (For help on the names of warnings, use `dialyzer -Whelp`.) Notice that the options can also be specified in the file with a `-dialyzer()` attribute. For details, see section Requesting or Suppressing Warnings in Source Files.

Note:

** the syntax of defines and includes is the same as that used by erlc(1).

Warning options:

`-Werror_handling` (***)

Include warnings for functions that only return by an exception.

`-Wextra_return` (***)

Warn about functions whose specification includes types that the function cannot return.

`-Wmissing_return` (***)

Warn about functions that return values that are not part of the specification.

`-Wno_behaviours`

Suppress warnings about behavior callbacks that drift from the published recommended interfaces.

`-Wno_contracts`

Suppress warnings about invalid contracts.

`-Wno_fail_call`

Suppress warnings for failing calls.

`-Wno_fun_app`

Suppress warnings for fun applications that will fail.

`-Wno_improper_lists`

Suppress warnings for construction of improper lists.

`-Wno_match`

Suppress warnings for patterns that are unused or cannot match.

`-Wno_missing_calls`

Suppress warnings about calls to missing functions.

`-Wno_opaque`

Suppress warnings for violations of opacity of data types.

`-Wno_return`

Suppress warnings for functions that will never return a value.

`-Wno_undefined_callbacks`

Suppress warnings about behaviors that have no `-callback` attributes for their callbacks.

`-Wno_unused`

Suppress warnings for unused functions.

`-Wno_unknown`

Suppress warnings about unknown functions and types. The default is to warn about unknown functions and types when setting the exit status. When using Dialyzer from Erlang, warnings about unknown functions and types are returned.

`-Wunderspecs` (***)

Warn about underspecified functions (the specification is strictly more allowing than the success typing).

`-Wunmatched_returns` (***)

Include warnings for function calls that ignore a structured return value or do not match against one of many possible return values. However, no warnings are included if the possible return values are a union of atoms or a union of numbers.

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 specification is strictly less allowing than the success typing).

`-Wspecdiffs` (***)

Warn when the specification is different than the success typing.

Note:

*** denotes options that turn on warnings rather than turning them off.

The following option is not strictly needed as it specifies the default. It is primarily intended to be used with the `-dialyzer` attribute. For an example see section Requesting or Suppressing Warnings in Source Files.

`-Wno_underspecs`

Suppress warnings about underspecified functions (the specification is strictly more allowing than the success typing).

`-Wno_extra_return`

Suppress warnings about functions whose specification includes types that the function cannot return.

`-Wno_missing_return`

Suppress warnings about functions that return values that are not part of the specification.

Using Dialyzer from Erlang

Dialyzer can be used directly from Erlang. The options are similar to the ones given from the command line, see section Using Dialyzer from the Command Line.

Default Dialyzer Options

The (host operating system) environment variable `ERL_COMPILER_OPTIONS` can be used to give default Dialyzer options. Its value must be a valid Erlang term. If the value is a list, it is used as is. If it is not a list, it is put into a list.

The list is appended to any options given to run/1 or on the command line.

The list can be retrieved with compile:env_compiler_options/0.

Currently the only option used is the `error_location` option.

Dialyzer configuration file:

Dialyzer's configuration file may also be used to augment the default options and those given directly to the Dialyzer command. It is commonly used to avoid repeating options which would otherwise need to be given explicitly to Dialyzer on every invocation.

The location of the configuration file can be set via the `DIALYZER_CONFIG` environment variable, and defaults to within the `user_config` from `filename:basedir/3`.

An example configuration file's contents might be:

```
      {incremental,
        {default_apps,[stdlib,kernel,erts]},
        {default_warning_apps,[stdlib]}
      }.
      {warnings, [no_improper_lists]}.
      {add_pathsa,["/users/samwise/potatoes/ebin"]}.
      {add_pathsz,["/users/smeagol/fish/ebin"]}.
```

Requesting or Suppressing Warnings in Source Files

Attribute `-dialyzer()` 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).
```

Attribute `-dialyzer()` 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]}).
```

The warning option for underspecified functions, `-Wunderspecs`, can result in useful warnings, but often functions with specifications that are strictly more allowing than the success typing cannot easily be modified to be less allowing. To turn off the warning for underspecified function `f/0`, include the following line:

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

For help on the warning options, use `dialyzer -Whelp`. The options are also enumerated, see type `warn_option()`.

Attribute `-dialyzer()` can also be used for turning on warnings. For example, if a module has been fixed regarding unmatched returns, adding the following line can help in assuring that no new unmatched return warnings are introduced:

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

Data Types

```
dial_option() =
    {files, [FileName :: file:filename()]} |
    {files_rec, [DirName :: file:filename()]} |
    {defines, [{Macro :: atom(), Value :: term()}]} |
    {from, src_code | byte_code} |
    {init_plt, FileName :: file:filename()} |
    {plts, [FileName :: file:filename()]} |
    {include_dirs, [DirName :: file:filename()]} |
    {output_file, FileName :: file:filename()} |
    {metrics_file, FileName :: file:filename()} |
    {module_lookup_file, FileName :: file:filename()} |
    {output_plt, FileName :: file:filename()} |
    {check_plt, boolean()} |
    {analysis_type,
     succ_typings | plt_add | plt_build | plt_check | plt_remove |
     incremental} |
    {warnings, [warn_option()]} |
    {get_warnings, boolean()} |
    {use_spec, boolean()} |
    {filename_opt, filename_opt()} |
    {callgraph_file, file:filename()} |
    {mod_deps_file, file:filename()} |
    {warning_files_rec, [DirName :: file:filename()]} |
    {error_location, error_location()}
```

Option `from` defaults to `byte_code`. Options `init_plt` and `plts` change the default.

```
dial_warn_tag() =
    warn_behaviour | warn_bin_construction | warn_callgraph |
    warn_contract_extra_return | warn_contract_missing_return |
    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_map_construction |
    warn_matching | warn_non_proper_list | warn_not_called |
    warn_opaque | warn_overlapping_contract |
    warn_return_no_exit | warn_return_only_exit |
    warn_undefined_callbacks | warn_unknown | warn_umatched_return
dial_warning() =
    {Tag :: dial_warn_tag(),
     Id :: file_location(),
     Msg :: {atom(), [term()]}}
error_location() = column | line
```

If the value of this option is `line`, an integer `Line` is used as `Location` in messages. If the value is `column`, a pair `{Line, Column}` is used as `Location`. The default is `column`.

```
file_location() =
    {File :: file:filename(), Location :: erl_anno:location()}
filename_opt() = basename | fullpath
format_option() =
    {indent_opt, boolean()} |
    {filename_opt, filename_opt()} |
    {error_location, error_location()}
warn_option() =
    error_handling | no_behaviours | no_contracts | no_fail_call |
    no_fun_app | no_improper_lists | no_match | no_missing_calls |
    no_opaque | no_return | no_undefined_callbacks |
    no_underspecs | no_unknown | no_unused | underspecs |
    unknown | unmatched_returns | overspecs | specdiffs |
    extra_return | no_extra_return | missing_return |
    no_missing_return
```

See section Warning options for a description of the warning options.

Exports

```
format_warning(Warnings) -> string()
```

Types:

```
   Warnings = dial_warning()
```

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

```
format_warning(Warnings, Options) -> string()
```

Types:

```
   Warnings = dial_warning()
   Options = filename_opt() | [format_option()]
   format_option() =
       {indent_opt, boolean()} |
       {filename_opt, filename_opt()} |
       {error_location, error_location()}
   filename_opt() = basename | fullpath
```

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

If `indent_opt` is set to `true` (default), line breaks are inserted in types, contracts, and Erlang code to improve readability.

If `error_location` is set to `column` (default), locations are formatted as `Line:Column` if the column number is available, otherwise locations are formatted as `Line` even if the column number is available.

```
plt_info(Plt) ->
            {ok, ClassicResult | IncrementalResult} |
            {error, Reason}
```

Types:

```
   Plt = file:filename()
   ClassicResult = [{files, [file:filename()]}]
   IncrementalResult = {incremental, [{modules, [module()]}]}
   Reason = not_valid | no_such_file | read_error
```

Returns information about the specified PLT.

```
run(Options) -> Warnings
```

Types:

```
   Options = [dial_option()]
   Warnings = [dial_warning()]
```

Dialyzer command-line version.

typer

Command

TypEr shows type information for Erlang modules to the user. Additionally, it can annotate the code of files with such type information.

Using TypEr from the Command Line

TypEr is used from the command-line. This section provides a brief description of the options. The same information can be obtained by writing the following in a shell:

```
typer --help
```

Usage:

```
typer [--help] [--version] [--plt PLT] [--edoc]
      [--show | --show-exported | --annotate | --annotate-inc-files | --annotate-in-place]
      [-Ddefine]* [-I include_dir]* [-pa dir]* [-pz dir]*
      [-T application]* file* [-r directory*]
```

Note:

* denotes that multiple occurrences of the option are possible.

Options:

`-r`

Search directories recursively for .erl files below them. If a list of files is given, this must be after them.

`--show`

Print type specifications for all functions on stdout. (This is the default behaviour; this option is not really needed.)

`--show-exported` (or `show_exported`)

Same as `--show`, but print specifications for exported functions only. Specs are displayed sorted alphabetically on the function's name.

`--annotate`

Annotate the specified files with type specifications.

`--annotate-inc-files`

Same as `--annotate` but annotates all `-include()` files as well as all .erl files. (Use this option with caution - it has not been tested much).

`--annotate-in-place`

Annotate directly on the source code files, instead of dumping the annotated files in a different directory (use this option with caution - has not been tested much)

`--edoc`

Print type information as Edoc `@spec` comments, not as type specs.

`--plt`

Use the specified dialyzer PLT file rather than the default one.

`-T file*`

The specified file(s) already contain type specifications and these are to be trusted in order to print specs for the rest of the files. (Multiple files or dirs, separated by spaces, can be specified.)

`-Dname` (or `-Dname=value`)

Pass the defined name(s) to TypEr. (**)

`-I`

Pass the include_dir to TypEr. (**)

`-pa dir`

Include `dir` in the path for Erlang. This is useful when analyzing files that have `-include_lib()` directives or use parse transforms.

`-pz dir`

Include `dir` in the path for Erlang. This is useful when analyzing files that have `-include_lib()` directives or use parse transforms.

`--version` (or `-v`)

Print the TypEr version and some more information and exit.

Note:

** options `-D` and `-I` work both from the command line and in the TypEr GUI; the syntax of defines and includes is the same as that used by erlc(1).