



TypeCheck

TypeCheck: Fast and flexible runtime type-checking for your Elixir projects.

hex v0.13.5 hexdocs latest coverage 85% coverage 85%

Core ideas

- Type- and function specifications are constructed using (essentially) the **same syntax** as built-in Elixir Typespecs.
- When a value does not match a type check, the user is shown **human-friendly error messages**.
- Types and type-checks are generated at compiletime.
 - This means **type-checking code is optimized** rigorously by the compiler.
- **Property-checking generators** can be extracted from type specifications without extra work.
 - Automatically create a **spectest** which checks for each function if it adheres to its spec.
- Flexibility to add **custom checks**: Subparts of a type can be named, and ‘type guards’ can be specified to restrict what values are allowed to match that refer to these types.

Prefer to watch a presentation instead of reading? See “TypeCheck: Effortless Runtime Type Checking” - Marten Wijnja - *ElixirConf EU 2022*.

Usage Example

We add `use TypeCheck` to a module and wherever we want to add runtime type-checks we replace the normal calls to `@type` and `@spec` with `@type!` and `@spec!` respectively.

```
1 defmodule User do
2   use TypeCheck
3   defstruct [:name, :age]
4
```

```
5  @type! t :: %User{name: binary, age: integer}
6  end
7
8  defmodule AgeCheck do
9    use TypeCheck
10
11    @spec! user_older_than?(User.t, integer) :: boolean
12    def user_older_than?(user, age) do
13      user.age >= age
14    end
15  end
```

Now we can try the following:

```
1  iex> AgeCheck.user_older_than?(%User{name: "Qqwy", age: 11}, 10)
2  true
3  iex> AgeCheck.user_older_than?(%User{name: "Qqwy", age: 9}, 10)
4  false
```

So far so good. Now let's see what happens when we pass values that are incorrect:

```
1  iex> AgeCheck.user_older_than?("foobar", 42)
2  ** (TypeCheck.TypeError) At lib/type_check_example.ex:28:
3  The call to `user_older_than?/2` failed,
4  because parameter no. 1 does not adhere to the spec `%User{age: integer
5  (), name: binary()}`.
6  Rather, its value is: `"foobar"`.
7  Details:
8  The call `user_older_than?("foobar", 42)`
9  does not adhere to spec `user_older_than?(%User{age: integer(), name:
10  binary()}, integer()) :: boolean()`. Reason:
11  parameter no. 1:
12  `"foobar"` does not check against `%User{age: integer(), name:
13  binary()}`. Reason:
14  `"foobar"` is not a map.
15  (type_check_example 0.1.0) lib/type_check_example.ex:28: AgeCheck.
16  user_older_than?/2
```

```
1  iex> AgeCheck.user_older_than?(%User{name: nil, age: 11}, 10)
2  ** (TypeCheck.TypeError) At lib/type_check_example.ex:28:
3  The call to `user_older_than?/2` failed,
4  because parameter no. 1 does not adhere to the spec `%User{age: integer
5  (), name: binary()}`.
6  Rather, its value is: `%User{age: 11, name: nil}`.
7  Details:
8  The call `user_older_than?(%User{age: 11, name: nil}, 10)`
9  does not adhere to spec `user_older_than?(%User{age: integer(), name:
10  binary()}, integer()) :: boolean()`. Reason:
11  parameter no. 1:
12  `%User{age: 11, name: nil}` does not check against `%User{age:
13  integer(), name: binary()}`. Reason:
```

```
11         under key `:name`:
12         `nil` is not a binary.
13 (type_check_example 0.1.0) lib/type_check_example.ex:28: AgeCheck.
    user_older_than?/2
```

```
1 iex> AgeCheck.user_older_than?(%User{name: "Aaron", age: nil}, 10)
2 ** (TypeCheck.TypeError) At lib/type_check_example.ex:28:
3 The call to `user_older_than?/2` failed,
4 because parameter no. 1 does not adhere to the spec `%User{age: integer
5   (), name: binary()}`.
6 Rather, its value is: `%User{age: nil, name: "Aaron"}`.
7 Details:
8 The call `user_older_than?(%User{age: nil, name: "Aaron"}, 10)`
9 does not adhere to spec `user_older_than?(%User{age: integer(), name:
10   binary()}, integer()) :: boolean()`. Reason:
11 parameter no. 1:
12 `%User{age: nil, name: "Aaron"}` does not check against `%User{
13   age: integer(), name: binary()}`. Reason:
14   under key `:age`:
15   `nil` is not an integer.
16 (type_check_example 0.1.0) lib/type_check_example.ex:28: AgeCheck.
17   user_older_than?/2
```

```
1
2 iex> AgeCheck.user_older_than?(%User{name: "José", age: 11}, 10.0)
3 ** (TypeCheck.TypeError) At lib/type_check_example.ex:28:
4 The call to `user_older_than?/2` failed,
5 because parameter no. 2 does not adhere to the spec `integer()`.
6 Rather, its value is: `10.0`.
7 Details:
8 The call `user_older_than?(%User{age: 11, name: "José"}, 10.0)`
9 does not adhere to spec `user_older_than?(%User{age: integer(), name:
10   binary()}, integer()) :: boolean()`. Reason:
11 parameter no. 2:
12 `10.0` is not an integer.
13 (type_check_example 0.1.0) lib/type_check_example.ex:28: AgeCheck.
14   user_older_than?/2
```

And if we were to introduce an error in the function definition:

```
1 defmodule AgeCheck do
2   use TypeCheck
3
4   @spec! user_older_than?(User.t, integer) :: boolean
5   def user_older_than?(user, age) do
6     user.age
7   end
8 end
```

Then we get a nice error message explaining that problem as well:

```

1 ** (TypeCheck.TypeError) The call to `user_older_than?/2` failed,
2 because the returned result does not adhere to the spec `boolean()`.
3 Rather, its value is: `26`.
4 Details:
5   The result of calling `user_older_than?(%User{age: 26, name: "Marten"
6     }, 10)`
7   does not adhere to spec `user_older_than?(%User{age: integer(), name:
8     binary()}, integer()) :: boolean()`. Reason:
9   Returned result:
10    `26` is not a boolean.
11    (type_check_example 0.1.0) lib/type_check_example.ex:28: AgeCheck.
12    user_older_than?/2

```

Features & Roadmap

Implemented

- ☑ Proof and implementation of the basic concept
- ☑ Custom type definitions (type, typep, opaque)
 - ☑ Basic
 - ☑ Parameterized
- ☑ Hide implementation of **opaque** from documentation
- ☑ Spec argument types checking
- ☑ Spec return type checking
- ☑ Spec possibly named arguments
- ☑ Implementation of Elixir's builtin types
 - ☑ Primitive types
 - ☑ More primitive types
 - ☑ Compound types
 - ☑ special forms like `|`, `a . b` etc.
 - ☑ Literal lists
 - ☑ Maps with keys => types
 - ☑ Structs with keys => types
 - ☑ More map/list-based structures.
 - ☑ Bitstring type syntax `<<>>`, `<<_ :: size>>`, `<<_ :: _ * unit>>`, `<<_ :: size, _ :: _ * unit>>`
- ☑ A **when** to add guards to typedefs for more power.
- ☑ Make errors raised when types do not match humanly readable

-
- ☒ Improve readability of spec-errors by repeating spec and which parameter did not match.
 - ☒ Creating generators from types
 - ☒ Don't warn on zero-arity types used without parentheses.
 - ☒ Hide structure of `opaque` and `typep` from documentation
 - ☒ Make sure to handle recursive (and mutually recursive) types without hanging.
 - ☒ A compile-error is raised when a type is expanded more than a million times
 - ☒ A macro called `lazy` is introduced to allow to defer type expansion to runtime (to *within* the check).
 - ☒ the Elixir formatter likes the way types+specs are constructed
 - ☒ A type `impl(ProtocolName)` to work with 'any type implementing protocol `ProtocolName`':
 - ☒ Type checks.
 - ☒ `StreamData` generator.
 - ☒ High code-coverage to ensure stability of implementation.
 - ☒ Make sure we handle most (if not all) of Typespec's primitive types and syntax. (With the exception of functions and binary pattern matching)
 - ☒ Option to turn `@type/@opaque/@typep`-injection off for the cases in which it generates improper results.
 - ☒ Manually overriding generators for user-specified types if so desired.
 - ☒ Creating generators from specs
 - ☒ Wrap spec-generators so you have a single statement to call in the test suite which will prop-test your function against all allowed inputs/outputs.
 - ☒ Option to turn the generation of runtime checks off for a given module in a particular environment (`enable_runtime_checks`).
 - ☒ Support for function-types (for typechecks as well as property-testing generators):
 - `(-> result_type)`
 - `(...-> result_type)`
 - `(param_type, param2_type -> result_type)`
 - ☒ Basic support for maps with a single `required(type)` or `optional(type)`.
 - ☒ Overrides for builtin remote types (`String.t`, `Enum.t`, `Range.t`, `MapSet.t` etc.) **(75% done)** Details
 - ☒ Overrides for more builtin remote types
 - ☒ Support for maps with mixed `required(type)` and `optional(type)` syntaxes.
-

-
- ☒ Configurable setting to turn checks on/off at compile-time, on a per-OTP-app basis (so you have control over your dependencies) as well as your individual modules.
 - ☒ Hide named types from opaque types.
 - ☒ A way to define structs and their field types at the same time.
 - ☒ Finalize formatter specification and make a generator for this so that people can easily test their own formatters.

Pre-stable

Longer-term future ideas

- ☐ Per-module or even per-spec settings to turn on/off, configure formatter, etc.

Installation

TypeCheck is available in Hex. The package can be installed by adding `type_check` to your list of dependencies in `mix.exs`:

```
1 def deps do
2   [
3     {:type_check, "~> 0.13.3"},
4     # To allow spectesting and property-testing data generators (
5       optional):
6     {:stream_data, "~> 0.5.0", only: :test},
7   ]
8 end
```

The documentation can be found at https://hexdocs.pm/type_check.

Formatter

TypeCheck exports a couple of macros that you might want to use without parentheses. To make `mix format` respect this setting, add `import_deps: [:type_check]` to your `.formatter.exs` file.

Changelog

The full changelog can be found [here](#)

TypeCheck compared to other tools

TypeCheck is by no means the other solution out there to reduce the number of bugs in your code.

Elixir's builtin typespecs and Dialyzer

Elixir's builtin type-specifications use the same syntax as TypeCheck. They are however not used by the compiler or the runtime, and therefore mainly exist to improve your documentation.

Besides documentation, extra external tools like Dialyzer can be used to perform static analysis of the types used in your application.

Dialyzer is an opt-in static analysis tool. This means that it can point out some inconsistencies or bugs, but because of its opt-in nature, there are also many problems it cannot detect, and it requires your dependencies to have written all of their typespecs correctly.

Dialyzer is also (unfortunately) infamous for its at times difficult-to-understand error messages.

An advantage that Dialyzer has over TypeCheck is that its checking is done without having to execute your program code (thus not having any effect on the runtime behaviour or efficiency of your projects).

Because TypeCheck adds `@type`, `@typep`, `@opaque` and `@spec`-attributes based on the types that are defined, it is possible to use Dialyzer together with TypeCheck.

Norm

Norm is an Elixir library for specifying the structure of data that can be used for both validation and data-generation.

On a superficial level, Norm and TypeCheck seem similar. However, there are important differences in their design considerations.

Is it any good?

yes