
TypedStruct



TypedStruct is a library for defining structs with a type without writing boilerplate code.

Rationale

To define a struct in Elixir, you probably want to define three things:

- the struct itself, with default values,
- the list of enforced keys,
- its associated type.

It ends up in something like this:

```
1 defmodule Person do
2   @moduledoc """
3   A struct representing a person.
4   """
5
6   @enforce_keys [:name]
7   defstruct name: nil,
8             age: nil,
9             happy?: true,
10            phone: nil
11
12   @typedoc "A person"
13   @type t() :: %__MODULE__{
14     name: String.t(),
15     age: non_neg_integer() | nil,
16     happy?: boolean(),
17     phone: String.t() | nil
18   }
19 end
```

In the example above you can notice several points:

- the keys are present in both the `defstruct` and type definition,
- enforced keys must also be written in `@enforce_keys`,
- if a key has no default value and is not enforced, its type should be nullable.

If you want to add a field in the struct, you must therefore:

- add the key with its default value in the `defstruct` list,
- add the key with its type in the type definition.

If the field is not optional, you should even add it to `@enforce_keys`. This is way too much work for lazy people like me, and moreover it can be error-prone.

It would be way better if we could write something like this:

```
1 defmodule Person do
2   @moduledoc """
3   A struct representing a person.
4   """
5
6   use TypedStruct
7
8   typedstruct do
9     @typedoc "A person"
10
11     field :name, String.t(), enforce: true
12     field :age, non_neg_integer()
13     field :happy?, boolean(), default: true
14     field :phone, String.t()
15   end
16 end
```

Thanks to TypedStruct, this is now possible :)

Usage

Setup

To use TypedStruct in your project, add this to your Mix dependencies:

```
1 {:typed_struct, "~> 0.3.0"}
```

If you do not plan to compile modules using TypedStruct at runtime, you can add `runtime: false` to the dependency tuple as TypedStruct is only used at build time.

If you want to avoid `mix format` putting parentheses on field definitions, you can add to your `.formatter.exs`:

```
1 [
2   ...,
3   import_deps: [:typed_struct]
4 ]
```

General usage

To define a typed struct, use `TypedStruct`, then define your struct within a `typedstruct` block:

```
1 defmodule MyStruct do
2   # Use TypedStruct to import the typedstruct macro.
3   use TypedStruct
4
5   # Define your struct.
6   typedstruct do
7     # Define each field with the field macro.
8     field :a_string, String.t()
9
10    # You can set a default value.
11    field :string_with_default, String.t(), default: "default"
12
13    # You can enforce a field.
14    field :enforced_field, integer(), enforce: true
15  end
16 end
```

Each field is defined through the `field/2` macro.

Options

If you want to enforce all the keys by default, you can do:

```
1 defmodule MyStruct do
2   use TypedStruct
3
4   # Enforce keys by default.
5   typedstruct enforce: true do
6     # This key is enforced.
7     field :enforced_by_default, term()
8
9     # You can override the default behaviour.
10    field :not_enforced, term(), enforce: false
11
12    # A key with a default value is not enforced.
13    field :not_enforced_either, integer(), default: 1
14  end
15 end
```

You can also generate an opaque type for the struct:

```
1 defmodule MyOpaqueStruct do
2   use TypedStruct
3
4   # Generate an opaque type for the struct.
5   typedstruct opaque: true do
6     field :name, String.t()
7   end
8 end
```

If you often define submodules containing only a struct, you can avoid boilerplate code:

```
1 defmodule MyModule do
2   use TypedStruct
3
4   # You now have %MyModule.Struct{}.
5   typedstruct module: Struct do
6     field :field, term()
7   end
8 end
```

Documentation

To add a `@typedoc` to the struct type, just add the attribute in the `typedstruct` block:

```
1 typedstruct do
2   @typedoc "A typed struct"
3
4   field :a_string, String.t()
5   field :an_int, integer()
6 end
```

You can also document submodules this way:

```
1 typedstruct module: MyStruct do
2   @moduledoc "A submodule with a typed struct."
3   @typedoc "A typed struct in a submodule"
4
5   field :a_string, String.t()
6   field :an_int, integer()
7 end
```

Plugins

It is possible to extend the scope of TypedStruct by using its plugin interface, as described in `TypedStruct.Plugin`. For instance, to automatically generate lenses with the Lens library, you can use `TypedStructLens` and do:

```
1 defmodule MyStruct do
2   use TypedStruct
3
4   typedstruct do
5     plugin TypedStructLens
6   end
7 end
```

```

7   field :a_field, String.t()
8   field :other_field, atom()
9   end
10
11  @spec change(t()) :: t()
12  def change(data) do
13    # a_field/0 is generated by TypedStructLens.
14    lens = a_field()
15    put_in(data, [lens], "Changed")
16  end
17 end

```

Some available plugins

- `typed_struct_lens` – Integration with the Lens library.
- `typed_struct_legacy_reflection` – Re-enables the legacy reflection functions from TypedStruct 0.1.x.

This list is not meant to be exhaustive, please search for “typed_struct” on hex.pm for other results. If you want your plugin to appear here, please open an issue.

What do I get?

When defining an empty `typedstruct` block:

```

1  defmodule Example do
2    use TypedStruct
3
4    typedstruct do
5      end
6  end

```

you get an empty struct with its module type `t()`:

```

1  defmodule Example do
2    @enforce_keys []
3    defstruct []
4
5    @type t() :: %__MODULE__{}
6  end

```

Each `field` call adds information to the struct, `@enforce_keys` and the type `t()`.

A field with no options adds the name to the `defstruct` list, with `nil` as default. The type itself is made nullable:

```
1 defmodule Example do
2   use TypedStruct
3
4   typedstruct do
5     field :name, String.t()
6   end
7 end
```

becomes:

```
1 defmodule Example do
2   @enforce_keys []
3   defstruct name: nil
4
5   @type t() :: %__MODULE__{
6     name: String.t() | nil
7   }
8 end
```

The **default** option adds the default value to the **defstruct**:

```
1 field :name, String.t(), default: "John Smith"
2
3 # Becomes
4 defstruct name: "John Smith"
```

When set to **true**, the **enforce** option enforces the key by adding it to the **@enforce_keys** attribute.

```
1 field :name, String.t(), enforce: true
2
3 # Becomes
4 @enforce_keys [:name]
5 defstruct name: nil
```

In both cases, the type has no reason to be nullable anymore by default. In one case the field is filled with its default value and not **nil**, and in the other case it is enforced. Both options would generate the following type:

```
1 @type t() :: %__MODULE__{
2   name: String.t() # Not nullable
3 }
```

Passing **opaque: true** replaces **@type** with **@opaque** in the struct type specification:

```
1 typedstruct opaque: true do
2   field :name, String.t()
3 end
```

generates the following type:

```
1 @opaque t() :: %__MODULE__{
2     name: String.t()
3 }
```

When passing `module: ModuleName`, the whole `typedstruct` block is wrapped in a module definition. This way, the following definition:

```
1 defmodule MyModule do
2   use TypedStruct
3
4   typedstruct module: Struct do
5     field :field, term()
6   end
7 end
```

becomes:

```
1 defmodule MyModule do
2   defmodule Struct do
3     @enforce_keys []
4     defstruct field: nil
5
6     @type t() :: %__MODULE__{
7       field: term() | nil
8     }
9   end
10 end
```

Initial roadmap

- ☒ Struct definition
- ☒ Type definition (with nullable types)
- ☒ Default values
- ☒ Enforced keys (non-nullable types)
- ☒ Plugin API

Plugin ideas

- ☐ Default value type-checking (is it possible?)
- ☐ Guard generation
- ☒ Integration with Lens
- ☐ Integration with Ecto

Related libraries

- Domo: a library to validate structs that define a `t()` type, like the one generated by `TypedStruct`.
- TypedEctoSchema: a library that provides a DSL on top of `Ecto.Schema` to achieve the same result as `TypedStruct`, with `Ecto`.

Contributing

Before contributing to this project, please read the CONTRIBUTING.md.

License

Copyright © 2018-2022 Jean-Philippe Cugnet and Contributors

This project is licensed under the MIT license.