NAME

Type::Library – tiny, yet Moo(se)–compatible type libraries

SYNOPSIS

```
package Types::Mine {
   use Scalar::Util qw(looks_like_number);
   use Type::Library -base;
   use Type::Tiny;

   my $NUM = "Type::Tiny"->new(
      name       => "Number",
      constraint => sub { looks_like_number($_) },
      message    => sub { "$_ ain't a number" },
   );

   __PACKAGE__->meta->add_type($NUM);

   __PACKAGE__->meta->make_immutable;
}

package Ermintrude {
   use Moo;
   use Types::Mine qw(Number);
   has favourite_number => (is => "ro", isa => Number);
}

package Bullwinkle {
   use Moose;
   use Types::Mine qw(Number);
   has favourite_number => (is => "ro", isa => Number);
}

package Maisy {
   use Mouse;
   use Types::Mine qw(Number);
   has favourite_number => (is => "ro", isa => Number);
}
```

STATUS

This module is covered by the Type-Tiny stability policy.

DESCRIPTION

Type::Library is a tiny class for creating MooseX::Types–like type libraries which are compatible with Moo, Moose and Mouse.

If you're reading this because you want to create a type library, then you're probably better off reading Type::Tiny::Manual::Libraries.

Methods

A type library is a singleton class. Use the `meta` method to get a blessed object which other methods can get called on. For example:

```
Types::Mine->meta->add_type($foo);
```

`add_type($type)` or `add_type(%opts)`

Add a type to the library. If `%opts` is given, then this method calls `Type::Tiny->new(%opts)` first, and adds the resultant type.

Adding a type named "Foo" to the library will automatically define four functions in the library's

namespace:

`Foo`
: Returns the Type::Tiny object.

`is_Foo($value)`
: Returns true iff `$value` passes the type constraint.

`assert_Foo($value)`
: Returns `$value` iff `$value` passes the type constraint. Dies otherwise.

`to_Foo($value)`
: Coerces the value to the type.

`get_type($name)`
: Gets the `Type::Tiny` object corresponding to the name.

`has_type($name)`
: Boolean; returns true if the type exists in the library.

`type_names`
: List all types defined by the library.

`add_coercion($c)` or `add_coercion(%opts)`
: Add a standalone coercion to the library. If `%opts` is given, then this method calls `Type::Coercion->new(%opts)` first, and adds the resultant coercion.

  Adding a coercion named "FooFromBar" to the library will automatically define a function in the library's namespace:

  `FooFromBar`
  : Returns the Type::Coercion object.

`get_coercion($name)`
: Gets the `Type::Coercion` object corresponding to the name.

`has_coercion($name)`
: Boolean; returns true if the coercion exists in the library.

`coercion_names`
: List all standalone coercions defined by the library.

`import(@args)`
: Type::Library–based libraries are exporters.

`make_immutable`
: A shortcut for calling `$type->coercion->freeze` on every type constraint in the library.

Constants

`NICE_PROTOTYPES`
: If this is true, then Type::Library will give parameterizable type constraints slightly the nicer prototype of `(;$)` instead of the default `(;@)`. This allows constructs like:

```
ArrayRef[Int] | HashRef[Int]
```

  ... to "just work".

Export

Type libraries are exporters. For the purposes of the following examples, assume that the `Types::Mine` library defines types `Number` and `String`.

```
# Exports nothing.
#
use Types::Mine;

# Exports a function "String" which is a constant returning
```

```
    # the String type constraint.
    #
    use Types::Mine qw( String );

    # Exports both String and Number as above.
    #
    use Types::Mine qw( String Number );

    # Same.
    #
    use Types::Mine qw( :types );

    # Exports "coerce_String" and "coerce_Number", as well as any other
    # coercions
    #
    use Types::Mine qw( :coercions );

    # Exports a sub "is_String" so that "is_String($foo)" is equivalent
    # to "String->check($foo)".
    #
    use Types::Mine qw( is_String );

    # Exports "is_String" and "is_Number".
    #
    use Types::Mine qw( :is );

    # Exports a sub "assert_String" so that "assert_String($foo)" is
    # equivalent to "String->assert_return($foo)".
    #
    use Types::Mine qw( assert_String );

    # Exports "assert_String" and "assert_Number".
    #
    use Types::Mine qw( :assert );

    # Exports a sub "to_String" so that "to_String($foo)" is equivalent
    # to "String->coerce($foo)".
    #
    use Types::Mine qw( to_String );

    # Exports "to_String" and "to_Number".
    #
    use Types::Mine qw( :to );

    # Exports "String", "is_String", "assert_String" and "coerce_String".
    #
    use Types::Mine qw( +String );

    # Exports everything.
    #
    use Types::Mine qw( :all );
```

Type libraries automatically inherit from Exporter::Tiny; see the documentation of that module for tips and tricks importing from libraries.

BUGS

Please report any bugs to <http://rt.cpan.org/Dist/Display.html?Queue=Type−Tiny>.

SEE ALSO

Type::Tiny::Manual.

Type::Tiny, Type::Utils, Types::Standard, Type::Coercion.

Moose::Util::TypeConstraints, Mouse::Util::TypeConstraints.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES