NAME

Type::Tiny – tiny, yet Moo(se)–compatible type constraint

SYNOPSIS

```
use Scalar::Util qw(looks_like_number);
use Type::Tiny;

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

package Ermintrude {
    use Moo;
    has favourite_number => (is => "ro", isa => $NUM);
}

package Bullwinkle {
    use Moose;
    has favourite_number => (is => "ro", isa => $NUM);
}

package Maisy {
    use Mouse;
    has favourite_number => (is => "ro", isa => $NUM);
}
```

STATUS

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

DESCRIPTION

Type::Tiny is a tiny class for creating Moose-like type constraint objects which are compatible with Moo, Moose and Mouse.

Maybe now we won't need to have separate MooseX, MouseX and MooX versions of everything? We can but hope...

This documents the internals of Type::Tiny. Type::Tiny::Manual is a better starting place if you're new.

Constructor

```
new(%attributes)
```
    Moose-style constructor function.

Attributes

Attributes are named values that may be passed to the constructor. For each attribute, there is a corresponding reader method. For example:

```
my $type = Type::Tiny->new( name => "Foo" );
print $type->name, "\n";   # says "Foo"
```

Important attributes

These are the attributes you are likely to be most interested in providing when creating your own type constraints, and most interested in reading when dealing with type constraint objects.

```
constraint
```
    Coderef to validate a value (`$_`) against the type constraint. The coderef will not be called unless the value is known to pass any parent type constraint (see `parent` below).

    Alternatively, a string of Perl code checking `$_` can be passed as a parameter to the constructor, and will be converted to a coderef.

Defaults to `sub { 1 }` − i.e. a coderef that passes all values.

parent
: Optional attribute; parent type constraint. For example, an "Integer" type constraint might have a parent "Number".

    If provided, must be a Type::Tiny object.

inlined
: A coderef which returns a string of Perl code suitable for inlining this type. Optional.

    If `constraint` (above) is a coderef generated via Sub::Quote, then Type::Tiny *may* be able to automatically generate `inlined` for you. If `constraint` (above) is a string, it will be able to.

name
: The name of the type constraint. These need to conform to certain naming rules (they must begin with an uppercase letter and continue using only letters, digits 0−9 and underscores).

    Optional; if not supplied will be an anonymous type constraint.

display_name
: A name to display for the type constraint when stringified. These don't have to conform to any naming rules. Optional; a default name will be calculated from the `name`.

library
: The package name of the type library this type is associated with. Optional. Informational only: setting this attribute does not install the type into the package.

deprecated
: Optional boolean indicating whether a type constraint is deprecated. Type::Library will issue a warning if you attempt to import a deprecated type constraint, but otherwise the type will continue to function as normal. There will not be deprecation warnings every time you validate a value, for instance. If omitted, defaults to the parent's deprecation status (or false if there's no parent).

message
: Coderef that returns an error message when `$_` does not validate against the type constraint. Optional (there's a vaguely sensible default.)

coercion
: A Type::Coercion object associated with this type.

    Generally speaking this attribute should not be passed to the constructor; you should rely on the default lazily-built coercion object.

    You may pass `coercion => 1` to the constructor to inherit coercions from the constraint's parent. (This requires the parent constraint to have a coercion.)

my_methods
: Experimenal hashref of additional methods that can be called on the type constraint object.

Attributes related to parameterizable and parameterized types

The following additional attributes are used for parameterizable (e.g. `ArrayRef`) and parameterized (e.g. `ArrayRef[Int]`) type constraints. Unlike Moose, these aren't handled by separate subclasses.

constraint_generator
: Coderef that is called when a type constraint is parameterized. When called, it is passed the list of parameters, though any parameter which looks like a foreign type constraint (Moose type constraints, Mouse type constraints, etc, *and coderefs(!!!)*) is first coerced to a native Type::Tiny object.

    Note that for compatibility with the Moose API, the base type is *not* passed to the constraint generator, but can be found in the package variable `$Type::Tiny::parameterize_type`. The first parameter is also available as `$_`.

    Types *can* be parameterized with an empty parameter list. For example, in Types::Standard, `Tuple` is

just an alias for `ArrayRef` but `Tuple[]` will only allow zero-length arrayrefs to pass the constraint. If you wish `YourType` and `YourType[]` to mean the same thing, then do:

```
return $Type::Tiny::parameterize_type unless @_;
```

The constraint generator should generate and return a new constraint coderef based on the parameters. Alternatively, the constraint generator can return a fully-formed Type::Tiny object, in which case the `name_generator`, `inline_generator`, and `coercion_generator` attributes documented below are ignored.

Optional; providing a generator makes this type into a parameterizable type constraint. If there is no generator, attempting to parameterize the type constraint will throw an exception.

name_generator
: A coderef which generates a new display_name based on parameters. Called with the same parameters and package variables as the `constraint_generator`. Expected to return a string.

Optional; the default is reasonable.

inline_generator
: A coderef which generates a new inlining coderef based on parameters. Called with the same parameters and package variables as the `constraint_generator`. Expected to return a coderef.

Optional.

coercion_generator
: A coderef which generates a new Type::Coercion object based on parameters. Called with the same parameters and package variables as the `constraint_generator`. Expected to return a blessed object.

Optional.

deep_explanation
: This API is not finalized. Coderef used by Error::TypeTiny::Assertion to peek inside parameterized types and figure out why a value doesn't pass the constraint.

parameters
: In parameterized types, returns an arrayref of the parameters.

Lazy generated attributes

The following attributes should not be usually passed to the constructor; unless you're doing something especially unusual, you should rely on the default lazily-built return values.

compiled_check
: Coderef to validate a value (`$_[0]`) against the type constraint. This coderef is expected to also handle all validation for the parent type constraints.

complementary_type
: A complementary type for this type. For example, the complementary type for an integer type would be all things that are not integers, including floating point numbers, but also alphabetic strings, arrayrefs, filehandles, etc.

moose_type, mouse_type
: Objects equivalent to this type constraint, but as a Moose::Meta::TypeConstraint or Mouse::Meta::TypeConstraint.

It should rarely be necessary to obtain a Moose::Meta::TypeConstraint object from Type::Tiny because the Type::Tiny object itself should be usable pretty much anywhere a Moose::Meta::TypeConstraint is expected.

Methods

Predicate methods

These methods return booleans indicating information about the type constraint. They are each tightly

associated with a particular attribute. (See "Attributes".)

`has_parent`,   `has_library`,   `has_inlined`,   `has_constraint_generator`, `has_inline_generator`, `has_coercion_generator`, `has_parameters`, `has_message`, `has_deep_explanation`
> Simple Moose-style predicate methods indicating the presence or absence of an attribute.

`has_coercion`
> Predicate method with a little extra DWIM. Returns false if the coercion is a no-op.

`is_anon`
> Returns true iff the type constraint does not have a `name`.

`is_parameterized`, `is_parameterizable`
> Indicates whether a type has been parameterized (e.g. `ArrayRef[Int]`) or could potentially be (e.g. `ArrayRef`).

Validation and coercion

The following methods are used for coercing and validating values against a type constraint:

`check($value)`
> Returns true iff the value passes the type constraint.

`validate($value)`
> Returns the error message for the value; returns an explicit undef if the value passes the type constraint.

`assert_valid($value)`
> Like `check($value)` but dies if the value does not pass the type constraint.
>
> Yes, that's three very similar methods. Blame Moose::Meta::TypeConstraint whose API I'm attempting to emulate. :−)

`assert_return($value)`
> Like `assert_valid($value)` but returns the value if it passes the type constraint.
>
> This seems a more useful behaviour than `assert_valid($value)`. I would have just changed `assert_valid($value)` to do this, except that there are edge cases where it could break Moose compatibility.

`get_message($value)`
> Returns the error message for the value; even if the value passes the type constraint.

`validate_explain($value, $varname)`
> Like `validate` but instead of a string error message, returns an arrayref of strings explaining the reasoning why the value does not meet the type constraint, examining parent types, etc.
>
> The `$varname` is an optional string like `'$foo'` indicating the name of the variable being checked.

`coerce($value)`
> Attempt to coerce `$value` to this type.

`assert_coerce($value)`
> Attempt to coerce `$value` to this type. Throws an exception if this is not possible.

Child type constraint creation and parameterization

These methods generate new type constraint objects that inherit from the constraint they are called upon:

`create_child_type(%attributes)`
> Construct a new Type::Tiny object with this object as its parent.

`where($coderef)`
> Shortcut for creating an anonymous child type constraint. Use it like `HashRef->where(sub { exists($_->{name}) })`. That said, you can get a similar result using overloaded `&`:

```
        HashRef & sub { exists($_->{name}) }
```

Like the `constraint` attribute, this will accept a string of Perl code:

```
        HashRef->where('exists($_->{name})')
```

`child_type_class`
    The class that create_child_type will construct by default.

`parameterize(@parameters)`
    Creates a new parameterized type; throws an exception if called on a non-parameterizable type.

`of(@parameters)`
    A cute alias for `parameterize`. Use it like `ArrayRef->of(Int)`.

`plus_coercions($type1, $code1, ...)`
    Shorthand for creating a new child type constraint with the same coercions as this one, but then adding some extra coercions (at a higher priority than the existing ones).

`plus_fallback_coercions($type1, $code1, ...)`
    Like `plus_coercions`, but added at a lower priority.

`minus_coercions($type1, ...)`
    Shorthand for creating a new child type constraint with fewer type coercions.

`no_coercions`
    Shorthand for creating a new child type constraint with no coercions at all.

Type relationship introspection methods

These methods allow you to determine a type constraint's relationship to other type constraints in an organised hierarchy:

`equals($other),      is_subtype_of($other),      is_supertype_of($other), is_a_type_of($other)`
    Compare two types. See Moose::Meta::TypeConstraint for what these all mean. (OK, Moose doesn't define `is_supertype_of`, but you get the idea, right?)

    Note that these have a slightly DWIM side to them. If you create two Type::Tiny::Class objects which test the same class, they're considered equal. And:

```
        my $subtype_of_Num = Types::Standard::Num->create_child_type;
        my $subtype_of_Int = Types::Standard::Int->create_child_type;
        $subtype_of_Int->is_subtype_of( $subtype_of_Num );  # true
```

`strictly_equals($other),                    is_strictly_subtype_of($other), is_strictly_supertype_of($other),is_strictly_a_type_of($other)`
    Stricter versions of the type comparison functions. These only care about explicit inheritance via `parent`.

```
        my $subtype_of_Num = Types::Standard::Num->create_child_type;
        my $subtype_of_Int = Types::Standard::Int->create_child_type;
        $subtype_of_Int->is_strictly_subtype_of( $subtype_of_Num );  # false
```

`parents`
    Returns a list of all this type constraint's ancestor constraints. For example, if called on the `Str` type constraint would return the list (`Value`, `Defined`, `Item`, `Any`).

    Due to a historical misunderstanding, this differs from the Moose implementation of the `parents` method. In Moose, `parents` only returns the immediate parent type constraints, and because type constraints only have one immediate parent, this is effectively an alias for `parent`. The extension module MooseX::Meta::TypeConstraint::Intersection is the only place where multiple type constraints are returned; and they are returned as an arrayref in violation of the base class' documentation. I'm keeping my behaviour as it seems more useful.

find_parent($coderef)

>　Loops through the parent type constraints *including the invocant itself* and returns the nearest ancestor type constraint where the coderef evaluates to true. Within the coderef the ancestor currently being checked is $_. Returns undef if there is no match.

>　In list context also returns the number of type constraints which had been looped through before the matching constraint was found.

coercibles

>　Return a type constraint which is the union of type constraints that can be coerced to this one (including this one). If this type constraint has no coercions, returns itself.

type_parameter

>　In parameterized type constraints, returns the first item on the list of parameters; otherwise returns undef. For example:

```
( ArrayRef[Int] )->type_parameter;    # returns Int
( ArrayRef[Int] )->parent;            # returns ArrayRef
```

>　Note that parameterizable type constraints can perfectly legitimately take multiple parameters (several off the parameterizable type constraints in Types::Standard do). This method only returns the first such parameter. "Attributes related to parameterizable and parameterized types" documents the parameters attribute, which returns an arrayref of all the parameters.

Inlining methods

The following methods are used to generate strings of Perl code which may be pasted into stringy evaluated subs to perform type checks:

can_be_inlined

>　Returns boolean indicating if this type can be inlined.

inline_check($varname)

>　Creates a type constraint check for a particular variable as a string of Perl code. For example:

```
print ( Types::Standard::Num->inline_check('$foo') );
```

>　prints the following output:

```
(!ref($foo) && Scalar::Util::looks_like_number($foo))
```

>　For Moose-compat, there is an alias _inline_check for this method.

inline_assert($varname)

>　Much like inline_check but outputs a statement of the form:

```
die ... unless ...;
```

>　Note that if this type has a custom error message, the inlined code will *ignore* this custom message!!

Other methods

qualified_name

>　For non-anonymous type constraints that have a library, returns a qualified "MyLib::MyType" sort of name. Otherwise, returns the same as name.

isa($class), can($method), AUTOLOAD(@args)

>　If Moose is loaded, then the combination of these methods is used to mock a Moose::Meta::TypeConstraint.

>　If Mouse is loaded, then isa mocks Mouse::Meta::TypeConstraint.

DOES($role)

>　Overridden to advertise support for various roles.

>　See also Type::API::Constraint, etc.

TIESCALAR, TIEARRAY, TIEHASH
> These are provided as hooks that wrap Type::Tie. (Type::Tie is distributed separately, and can be used with non−Type::Tiny type constraints too.) They allow the following to work:

```
use Types::Standard qw(Int);
tie my @list, Int;
push @list, 123, 456;    # ok
push @list, "Hello";     # dies
```

The following methods exist for Moose/Mouse compatibility, but do not do anything useful.

```
compile_type_constraint
hand_optimized_type_constraint
has_hand_optimized_type_constraint
inline_environment
meta
```

Overloading
- Stringification is overloaded to return the qualified name.

- Boolification is overloaded to always return true.

- Coderefification is overloaded to call `assert_return`.

- On Perl 5.10.1 and above, smart match is overloaded to call `check`.

- The == operator is overloaded to call `equals`.

- The < and > operators are overloaded to call `is_subtype_of` and `is_supertype_of`.

- The ~ operator is overloaded to call `complementary_type`.

- The | operator is overloaded to build a union of two type constraints. See Type::Tiny::Union.

- The `&` operator is overloaded to build the intersection of two type constraints. See Type::Tiny::Intersection.

Previous versions of Type::Tiny would overload the + operator to call `plus_coercions` or `plus_fallback_coercions` as appropriate. Support for this was dropped after 0.040.

Constants
Type::Tiny::SUPPORT_SMARTMATCH
> Indicates whether the smart match overload is supported on your version of Perl.

Package Variables
$Type::Tiny::DD
> This undef by default but may be set to a coderef that Type::Tiny and related modules will use to dump data structures in things like error messages.

> Otherwise Type::Tiny uses it's own routine to dump data structures. $DD may then be set to a number to limit the lengths of the dumps. (Default limit is 72.)

> This is a package variable (rather than get/set class methods) to allow for easy localization.

Environment
PERL_TYPE_TINY_XS
> Currently this has more effect on Types::Standard than Type::Tiny. In future it may be used to trigger or suppress the loading XS implementations of parts of Type::Tiny.

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

SUPPORT
IRC: support is available through in the *#moops* channel on irc.perl.org <http://www.irc.perl.org/channels.html>.

SEE ALSO

    Type::Tiny::Manual, Type::API.

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

    Type::Tiny::Class, Type::Tiny::Role, Type::Tiny::Duck, Type::Tiny::Enum, Type::Tiny::Union, Type::Tiny::Intersection.

    Moose::Meta::TypeConstraint, Mouse::Meta::TypeConstraint.

    Type::Params.

AUTHOR

    Toby Inkster <tobyink@cpan.org>.

THANKS

    Thanks to Matt S Trout for advice on Moo integration.

COPYRIGHT AND LICENCE

    This software is copyright (c) 2013−2014, 2017−2019 by Toby Inkster.

    This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

DISCLAIMER OF WARRANTIES

    THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.