NAME

Type::Coercion – a set of coercions to a particular target type constraint

STATUS

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

DESCRIPTION

Constructors

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

`add($c1, $c2)`
: Create a Type::Coercion from two existing Type::Coercion objects.

Attributes

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

```
my $c = Type::Coercion->new( type_constraint => Int );
my $t = $c->type_constraint;  # Int
```

Important attributes

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

`type_constraint`
: Weak reference to the target type constraint (i.e. the type constraint which the output of coercion coderefs is expected to conform to).

`type_coercion_map`
: Arrayref of source–type/code pairs.

`frozen`
: Boolean; default false. A frozen coercion cannot have `add_type_coercions` called upon it.

`name`
: A name for the coercion. 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 coercion.

`display_name`
: A name to display for the coercion 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 coercion is associated with. Optional. Informational only: setting this attribute does not install the coercion into the package.

Attributes related to parameterizable and parameterized coercions

The following attributes are used for parameterized coercions, but are not fully documented because they may change in the near future:

`coercion_generator`
`parameters`
`parameterized_from`

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_coercion`
Coderef to coerce a value (`$_[0]`).

The general point of this attribute is that you should not set it, but rely on the lazily-built default. Type::Coerce will usually generate a pretty fast coderef, inlining all type constraint checks, etc.

`moose_coercion`
A Moose::Meta::TypeCoercion object equivalent to this one. Don't set this manually; rely on the default built one.

Methods

Predicate methods

These methods return booleans indicating information about the coercion. They are each tightly associated with a particular attribute. (See "Attributes".)

`has_type_constraint`, `has_library`
Simple Moose-style predicate methods indicating the presence or absence of an attribute.

`is_anon`
Returns true iff the coercion does not have a `name`.

The following predicates are used for parameterized coercions, but are not fully documented because they may change in the near future:

`has_coercion_generator`
`has_parameters`
`is_parameterizable`
`is_parameterized`

Coercion

The following methods are used for coercing values to a type constraint:

`coerce($value)`
Coerce the value to the target type.

Returns the coerced value, or the original value if no coercion was possible.

`assert_coerce($value)`
Coerce the value to the target type, and throw an exception if the result does not validate against the target type constraint.

Returns the coerced value.

Coercion code definition methods

These methods all return `$self` so are suitable for chaining.

`add_type_coercions($type1, $code1, ...)`
Takes one or more pairs of Type::Tiny constraints and coercion code, creating an ordered list of source types and coercion codes.

Coercion codes can be expressed as either a string of Perl code (this includes objects which overload stringification), or a coderef (or object that overloads coderefification). In either case, the value to be coerced is `$_`.

`add_type_coercions($coercion_object)` also works, and can be used to copy coercions from another type constraint:

```
$type->coercion->add_type_coercions($othertype->coercion)->freeze;
```

`freeze`
Sets the `frozen` attribute to true. Called automatically by Type::Tiny sometimes.

`i_really_want_to_unfreeze`
: If you really want to unfreeze a coercion, call this method.

  Don't call this method. It will potentially lead to subtle bugs.

  This method is considered unstable; future versions of Type::Tiny may alter its behaviour (e.g. to throw an exception if it has been detected that unfreezing this particular coercion will cause bugs).

Parameterization

The following method is used for parameterized coercions, but is not fully documented because it may change in the near future:

`parameterize(@params)`

Type coercion introspection methods

These methods allow you to determine a coercion's relationship to type constraints:

`has_coercion_for_type($source_type)`
: Returns true iff this coercion has a coercion from the source type.

  Returns the special string `"0 but true"` if no coercion should actually be necessary for this type. (For example, if a coercion coerces to a theoretical "Number" type, there is probably no coercion necessary for values that already conform to the "Integer" type.)

`has_coercion_for_value($value)`
: Returns true iff the value could be coerced by this coercion.

  Returns the special string `"0 but true"` if no coercion would be actually be necessary for this value (due to it already meeting the target type constraint).

The `type_constraint` attribute provides a type constraint object for the target type constraint of the coercion. See "Attributes".

Inlining methods

The following methods are used to generate strings of Perl code which may be pasted into stringy `eval`uated subs to perform type coercions:

`can_be_inlined`
: Returns true iff the coercion can be inlined.

`inline_coercion($varname)`
: Much like `inline_coerce` from Type::Tiny.

Other methods

`qualified_name`
: For non-anonymous coercions that have a library, returns a qualified `"MyLib::MyCoercion"` 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::TypeCoercion.

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

`compile_type_coercion`
`meta`

Overloading

- Boolification is overloaded to always return true.
- Coderefification is overloaded to call `coerce`.
- On Perl 5.10.1 and above, smart match is overloaded to call `has_coercion_for_value`.

Previous versions of Type::Coercion would overload the `+` operator to call `add`. Support for this was dropped after 0.040.

DIAGNOSTICS

Attempt to add coercion code to a Type::Coercion which has been frozen

Type::Tiny type constraints are designed as immutable objects. Once you've created a constraint, rather than modifying it you generally create child constraints to do what you need.

Type::Coercion objects, on the other hand, are mutable. Coercion routines can be added at any time during the object's lifetime.

Sometimes Type::Tiny needs to freeze a Type::Coercion object to prevent this. In Moose and Mouse code this is likely to happen as soon as you use a type constraint in an attribute.

Workarounds:

- Define as many of your coercions as possible within type libraries, not within the code that uses the type libraries. The type library will be evaluated relatively early, likely before there is any reason to freeze a coercion.
- If you do need to add coercions to a type within application code outside the type library, instead create a subtype and add coercions to that. The `plus_coercions` method provided by Type::Tiny should make this simple.

BUGS

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

SEE ALSO

Type::Tiny::Manual.

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

Type::Coercion::Union.

Moose::Meta::TypeCoercion.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES