NAME

Type::Tiny::Manual::Coercions – advanced information on coercions

MANUAL

This section of the manual assumes you've already read Type::Tiny::Manual::UsingWithMoo.

Type::Tiny takes a slightly different approach to type constraints from Moose. In Moose, there is a single flat namespace for type constraints. Moose defines a type constraint called **Str** for strings and a type constraint called **ArrayRef** for arrayrefs. If you want to define strings differently (maybe you think that the empty string doesn't really count as a string, or maybe you think objects overloading `q[""]` should count as strings) then you can't call it **Str**; you need to choose a different name.

With Type::Tiny, two type libraries can each offer a string type constraint with their own definitions for what counts as a string, and you can choose which one to import, or import them both with different names:

```
use Some::Types qw( Str );
use Other::Types "Str" => { -as => "Str2" };
```

This might seem to be a small advantage of Type::Tiny, but where this global-versus-local philosophy really makes a difference is coercions.

Let's imagine for a part of your application that deals with reading username and password data you need to have a "username:password" string. You may wish to accept a `[$username, $password]` arrayref and coerce it to a string using `join ":", @$arrayref`. But another part of your application deals with slurping log files, and wants to coerce a string from an arrayref using `join "\n", @$arrayref`. These are both perfectly sensible ways to coerce an arrayref. In Moose, a typical way to do this would be:

```
package My::UserManager {
  use Moose;
  use Moose::Util::TypeConstraints;

  coerce 'Str',
    from 'ArrayRef', via { join ":", @$_ };

  ...;
}

package My::LogReader {
  use Moose;
  use Moose::Util::TypeConstraints;

  coerce 'Str',
    from 'ArrayRef', via { join "\n", @$_ };

  ...;
}
```

However, because in Moose all types and coercions are global, if both these classes are loaded, only one of them will work. One class will overrule the other's coercion. Which one "wins" will depend on load order.

It is possible to solve this with Moose native types, but it requires extra work. (The solution is for My::UserManager and My::LogReader to each create a subtype of **Str** and define the coercion on that subtype instead of on **Str** directly.)

Type::Tiny solves this in two ways:

1. Type::Tiny makes it possible for type libraries to "protect" their type constraints to prevent external code from adding new coercions to them.

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

   You can freeze coercions for your entire type library using:

```
__PACKAGE__->make_immutable;
```

If you try to add coercions to a type constraint that has frozen coercions, it will throw an error.

```
use Types::Standard qw( Str ArrayRef );

Str->coercion->add_type_coercions(
  ArrayRef, sub { join "\n", @$_ },
);
```

2. Type::Tiny makes the above-mentioned pattern of adding coercions to a subtype much easier.

```
use Types::Standard ( Str ArrayRef );

my $subtype = Str->plus_coercions(
  ArrayRef, sub { join "\n", @$_ },
);
```

The `plus_coercions` method creates a new child type, adds new coercions to it, copies any existing coercions from the parent type, and then freezes coercions for the new child type.

The end result is you now have a "copy" of **Str** that can coerce from **ArrayRef** but other copies of **Str** won't be affected by your coercion.

Defining Coercions within Type Libraries

Some coercions like joining an arrayref to make a string are not going to be coercions that everybody will agree on. Join with a line break in between them as above? Or with a colon, a tab, a space, some other chanaracter? It depends a lot on your application.

Others, like coercing a Path::Tiny object from a string, are likely to be very obvious. It is this kind of coercion that it makes sense to define within the library itself so it's available to any packages that use the library.

```
my $pt = __PACKAGE__->add_type(
  Type::Tiny::Class->new(
    name    => 'Path',
    class   => 'Path::Tiny',
  ),
);

$pt->coercion->add_type_coercions(
  Str, q{ Path::Tiny::path($_) },
);

$pt->coercion->freeze;
```

Tweak Coercions Outside Type Libraries

The `plus_coercions` method creates a new type constraint with additional coercions. If the original type already had coercions, the new coercions have a higher priority.

There's also a `plus_fallback_coercions` method which does the same as `plus_coercions` but adds the new coercions with a lower priority than any existing ones.

Type::Tiny::Class provides a `plus_constructors` method as a shortcut for coercing via a constructor method. The following two are the same:

```
Path->plus_constructors(Str, "new")

Path->plus_coercions(Str, q{ Path::Tiny->new($_) })
```

To create a type constraint without particular existing coercions, you can use `minus_coercions`. The following uses the **Datetime** type defined in Type::Tiny::Manual::Libraries, removing the coercion from

Int but keeping the coercions from **Undef** and **Dict**.

```
use Types::Standard qw( Int );
use Example::Types qw( Datetime );

has start_date => (
  is      => 'ro',
  isa     => Datetime->minus_coercions(Int),
  coerce  => 1,
);
```

There's also a `no_coercions` method that creates a subtype with no coercions at all. This is most useful either to create a "blank slate" for `plus_coercions`:

```
my $Path = Path->no_coercions->plus_coercions(Str, sub { ... });
```

Or to disable coercions for Type::Params. Type::Params will always automatically coerce a parameter if there is a coercion for that type.

```
use Types::Standard qw( Object );
use Types::Common::String qw( UpperCaseStr );
use Type::Params;

sub set_account_name {
  state $check = compile( Object, UpperCaseStr->no_coercions );
  my ($self, $name) = $check->(@_);
  $self->_account_name($name);
  $self->db->update($self);
  return $self;
}

# This will die instead of coercing from lowercase
$robert->_set_account_name('bob');
```

Named Coercions

A compromise between defining a coercion in the type library or defining them in the package that uses the type library is for a type library to define a named collection of coercions which can be optionally added to a type constraint.

```
{
  package MyApp::Types;
  use Type::Library -base;
  use Type::Utils qw( extends );

  BEGIN { extends 'Types::Standard' };

  __PACKAGE__->add_coercion(
    name              => "FromLines",
    type_constraint   => ArrayRef,
    type_coercion_map => [
      Str,     q{ [split /\n/] },
      Undef,   q{ [] },
    ],
  );
}
```

This set of coercions has a name and can be imported and used:

```
use MyApp::Types qw( ArrayRef FromLines );
```

```
has lines => (
  is      => 'ro',
  isa     => ArrayRef->plus_coercions( FromLines ),
  coerce  => 1,
);
```

Types::Standard defines a named coercion **MkOpt** designed to be used for **OptList**.

```
use Types::Standard qw( OptList MkOpt );
my $OptList = OptList->plus_coercions(MkOpt);
```

Parameterized Coercions

Named coercions can also be parameterizable.

```
my $ArrayOfLines = ArrayRef->plus_coercions( Split[ qr{\n} ] );
```

Types::Standard defines **Split** and **Join** parameterizable coercions.

Viewing the source code for Types::Standard should give you hints as to how they are implemented.

"Deep" Coercions

Certain parameterized type constraints can automatically acquire coercions if their parameters have coercions. For example:

```
ArrayRef[ Int->plus_coercions(Num, q{int($_)}) ]
```

... does what you mean!

The parameterized type constraints that do this magic include the following ones from Types::Standard:

- **ScalarRef**
- **ArrayRef**
- **HashRef**
- **Map**
- **Tuple**
- **CycleTuple**
- **Dict**
- **Optional**
- **Maybe**

Imagine we're defining a type **Paths** in a type library:

```
__PACKAGE__->add_type(
  name      => 'Paths',
  parent    => ArrayRef[Path],
);
```

The **Path** type has a coercion from **Str**, so **Paths** should be able to coerce from an arrayref of strings, right?

Wrong! Although **ArrayRef[Path]** could coerce from an arrayref of strings, **Paths** is a separate type constraint which, although it inherits from **ArrayRef[Path]** has its own (currently empty) set of coercions.

Because that is often not what you want, Type::Tiny provides a shortcut when declaring a subtype to copy the parent type constraint's coercions:

```
__PACKAGE__->add_type(
  name      => 'Paths',
  parent    => ArrayRef[Path],
  coercion  => 1,   # inherit
);
```

Now **Paths** can coerce from an arrayref of strings.

Deep Caveat

Currently there exists ill-defined behaviour resulting from mixing deep coercions and mutable (non-frozen) coercions. Consider the following:

```
class_type Path, { class => "Path::Tiny" };
coerce Path,
   from Str, via { "Path::Tiny"->new($_) };

declare Paths, as ArrayRef[Path], coercion => 1;

coerce Path,
   from InstanceOf["My::File"], via { $_->get_path };
```

An arrayref of strings can now be coerced to an arrayref of Path::Tiny objects, but is it also now possible to coerce an arrayref of My::File objects to an arrayref of Path::Tiny objects?

Currently the answer is ‘‘no’’, but this is mostly down to implementation details. It’s not clear what the best way to behave in this situation is, and it could start working at some point in the future.

This is why you should freeze coercions.

Chained Coercions

Consider the following type library:

```
package Types::Geometric {
   use Type::Library -base, -declare => qw(
      VectorArray
      VectorArray3D
      Point
      Point3D
   );
   use Type::Utils;
   use Types::Standard qw( Num Tuple InstanceOf );

   declare VectorArray,
      as Tuple[Num, Num];

   declare VectorArray3D,
      as Tuple[Num, Num, Num];

   coerce VectorArray3D,
      from VectorArray, via {
         [ @$_, 0 ];
      };

   class_type Point, { class => "Point" };

   coerce Point,
      from VectorArray, via {
         Point->new(x => $_->[0], y => $_->[1]);
      };

   class_type Point3D, { class => "Point3D" };

   coerce Point3D,
      from VectorArray3D, via {
         Point3D->new(x => $_->[0], y => $_->[1], z => $_->[2]);
      },
```

```
        from Point, via {
           Point3D->new(x => $_->x, y => $_->y, z => 0);
        };
}
```

Given an arrayref `[1, 1]` you might reasonably expect it to be coercible to a **Point3D** object; it matches the type constraint **VectorArray** so can be coerced to **VectorArray3D** and thus to **Point3D**.

However, Type::Coercion does not automatically chain coercions like this. Firstly, it would be incompatible with Moose's type coercion system which does not chain coercions. Secondly, it's ambiguous; in our example, the arrayref could be coerced along two different paths (via **VectorArray3D** or via **Point**); in this case the end result would be the same, but in other cases it might not. Thirdly, it runs the risk of accidentally creating loops.

Doing the chaining manually though is pretty simple. Firstly, we'll take note of the `coercibles` method in Type::Tiny. This method called as `VectorArray3D->coercibles` returns a type constraint meaning "anything that can be coerced to a **VectorArray3D**".

So we can define the coercions for **Point3D** as:

```
coerce Point3D,
   from VectorArray3D->coercibles, via {
      my $tmp = to_VectorArray3D($_);
      Point3D->new(x => $tmp->[0], y => $tmp->[1], z => $tmp->[2]);
   },
   from Point, via {
      Point3D->new(x => $_->x, y => $_->y, z => 0);
   };
```

... and now coercing from `[1, 1]` will work.

SEE ALSO

Moose::Manual::BestPractices,
<https://web.archive.org/web/20090624164256/http://www.catalyzed.org/2009/06/keeping-your-coercions-to-yourself.html
MooseX::Types::MoreUtils.

NEXT STEPS

After that last example, probably have a little lie down. Once you're recovered, here's your next step:

- Type::Tiny::Manual::AllTypes

  An alphabetical list of all type constraints bundled with Type::Tiny.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES