NAME

Type::Tiny::Manual::Coercions – adding coercions to type constraints

DESCRIPTION

Stop! Don't do it!

OK, it's fairly common practice in Moose/Mouse code to define coercions for type constraints. For example, suppose we define a type constraint in a type library:

```
class_type PathTiny, { class => "Path::Tiny" };
```

We may wish to define a coercion (i.e. a conversion routine) to handle strings, and convert them into Path::Tiny objects:

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

However, there are good reasons to avoid this practice. It ties the coercion routine to the type constraint. Any people wishing to use your `PathTiny` type constraint need to buy in to your idea of how they should be coerced from `Str`. With Path::Tiny this is unlikely to be controversial, however consider:

```
coerce ArrayRef,
   from Str, via { [split /\n/] };
```

In one part of the application (dealing with parsing log files for instance), this could be legitimate. But another part (dealing with logins perhaps) might prefer to split on colons. Another (dealing with web services) might attempt to parse the string as a JSON array.

If all these coercions have attached themselves to the `ArrayRef` type constraint, coercing a string becomes a complicated proposition! In a large application where coercions are defined across many different files, the application can start to suffer from ''spooky action at a distance''.

In the interests of Moose-compatibility, Type::Tiny and Type::Coercion do allow you to define coercions this way, but they also provide an alternative that you should consider: `plus_coercions`.

plus_coercions

Type::Tiny offers a method `plus_coercions` which constructs a new anonymous type constraint, but with additional coercions.

In our earlier example, we'd define the `PathTiny` type constraint in our type library as before:

```
class_type PathTiny, { class => "Path::Tiny" };
```

But then not define any coercions for it. Later, when using the type constraint in a class, we can add coercions:

```
my $ConfigFileType = PathTiny->plus_coercions(
   Str,   sub { "Path::Tiny"->new($_) },
   Undef, sub { "Path::Tiny"->new("/etc/myapp/default.conf") },
);

has config_file => (
   is     => "ro",
   isa    => $ConfigFileType,
   coerce => 1,
);
```

Where the `PathTiny` constraint is used in another part of the code, it will not see these coercions, because they were added to the new anonymous type constraint, not to the `PathTiny` constraint itself!

Named Coercions

A type library may define a named set of coercions to a particular type. For example, let's define that coercion from `Str` to `ArrayRef`:

```
declare_coercion "LinesFromStr",
   to_type ArrayRef,
   from Str, q{ [split /\n/] };
```

Now we can import that coercion using a name, and it makes our code look a little cleaner:

```
use Types::Standard qw(ArrayRef);
use MyApp::Types qw(LinesFromStr);

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

Parameterized Coercions

Parameterized type constraints are familiar from Moose. For example, an arrayref of integers:

```
ArrayRef[Int]
```

Type::Coercion supports parameterized named coercions too. For example, the following type constraint has a coercion from strings that splits them into lines:

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

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

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

plus_fallback_coercions, minus_coercions and no_coercions

Getting back to the `plus_coercions` method, there are some other methods that perform coercion maths.

`plus_fallback_coercions` is the same as `plus_coercions` but the added coercions have a lower priority than any existing coercions.

`minus_coercions` can be given a list of type constraints that we wish to ignore coercions for. Imagine our `PathTiny` constraint already has a coercion from `Str`, then the following creates a new anonymous type constraint without that coercion:

```
PathTiny->minus_coercions(Str)
```

`no_coercions` gives us a new type anonymous constraint without any of its parents coercions. This is useful as a way to create a blank slate for a subsequent `plus_coercions`:

```
PathTiny->no_coercions->plus_coercions(...)
```

plus_constructors

The `plus_constructors` method defined in Type::Tiny::Class is sugar for `plus_coercions`. The following two are the same:

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

PathTiny->plus_constructors(Str, "new");
```

"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 declaring a type library:

```
declare Paths, as ArrayRef[PathTiny];
```

The `PathTiny` type (declared earlier in the tutorial) has a coercion from `Str`, so `Paths` should be able to coerce from an arrayref of strings, right?

Wrong! `ArrayRef[PathTiny]` can coerce from an arrayref of strings, but `Paths` is a separate type constraint which, although it inherits from `ArrayRef[PathTiny]` 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:

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

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

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

coerce PathTiny,
   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.

You should avoid falling into this trap by following the advice found under ‘‘The (Lack of) Zen of 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.

The (Lack of) Zen of Coercions

Coercions can lead to ugliness.

Let's say we define a type constraint `Path` which has a coercion from `Str`. Now we define a class which uses that type constraint.

Now in another class, we define a coercion from `ArrayRef` to `Path`. This kind of action at a distance is not really desirable. And in fact, things will probably subtly break – the first class may have already built a constructor inlining a bunch of code from the coercion.

However, you too can achieve coercion zen by following these three weird tricks <http://www.slate.com/articles/business/moneybox/2013/07/how_one_weird_trick_conquered_the_internet_what_happen

1. If you want to define coercions for a type, do it *within your type constraint library*, so the coercions are all defined before the type constraint is ever used.
2. At the end of your type constraint library, consider calling `$type->coercion->freeze` on each type constraint that has a coercion. This makes the type's coercions immutable. If anybody wants to define any additional coercions, they'll have to create a child type to do it with.

   A shortcut exists to do this on all types in your library:

   ```
   __PACKAGE__->meta->make_immutable;
   ```
3. Use `plus_coercions` and similar methods to easily create a child type constraint of any existing type, and add more coercions to it. Don't fiddle directly with the existing type constraint which may be being used elsewhere.

   Note that these methods all return type constraint objects with frozen (immutable) coercions.

That's it.

SEE ALSO

Moose::Manual::BestPractices,
<http://www.catalyzed.org/2009/06/keeping-your-coercions-to-yourself.html>.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES