NAME

MooseX::Types − Organise your Moose types in libraries

VERSION

version 0.50

SYNOPSIS

Library Definition

```
package MyLibrary;

# predeclare our own types
use MooseX::Types -declare => [
    qw(
        PositiveInt
        NegativeInt
        ArrayRefOfPositiveInt
        ArrayRefOfAtLeastThreeNegativeInts
        LotsOfInnerConstraints
        StrOrArrayRef
        MyDateTime
        )
];

# import builtin types
use MooseX::Types::Moose qw/Int HashRef/;

# type definition.
subtype PositiveInt,
    as Int,
    where { $_ > 0 },
    message { "Int is not larger than 0" };

subtype NegativeInt,
    as Int,
    where { $_ < 0 },
    message { "Int is not smaller than 0" };

# type coercion
coerce PositiveInt,
    from Int,
        via { 1 };

# with parameterized constraints.

subtype ArrayRefOfPositiveInt,
  as ArrayRef[PositiveInt];

subtype ArrayRefOfAtLeastThreeNegativeInts,
  as ArrayRef[NegativeInt],
  where { scalar(@$_) > 2 };

subtype LotsOfInnerConstraints,
  as ArrayRef[ArrayRef[HashRef[Int]]];

# with TypeConstraint Unions

subtype StrOrArrayRef,
  as Str|ArrayRef;

# class types
```

```
    class_type 'DateTime';

    # or better

    class_type MyDateTime, { class => 'DateTime' };

    coerce MyDateTime,
      from HashRef,
      via { DateTime->new(%$_) };

    1;
```

Usage

```
    package Foo;
    use Moose;
    use MyLibrary qw( PositiveInt NegativeInt );

    # use the exported constants as type names
    has 'bar',
        isa    => PositiveInt,
        is     => 'rw';
    has 'baz',
        isa    => NegativeInt,
        is     => 'rw';

    sub quux {
        my ($self, $value);

        # test the value
        print "positive\n" if is_PositiveInt($value);
        print "negative\n" if is_NegativeInt($value);

        # coerce the value, NegativeInt doesn't have a coercion
        # helper, since it didn't define any coercions.
        $value = to_PositiveInt($value) or die "Cannot coerce";
    }

    1;
```

DESCRIPTION

The type system provided by Moose effectively makes all of its builtin type global, as are any types you declare with Moose. This means that every module that declares a type named `PositiveInt` is sharing the same type object. This can be a problem when different parts of the code base want to use the same name for different things.

This package lets you declare types using short names, but behind the scenes it namespaces all your type declarations, effectively prevent name clashes between packages.

This is done by creating a type library module like `MyApp::Types` and then importing types from that module into other modules.

As a side effect, the declaration mechanism allows you to write type names as barewords (really function calls), which catches typos in names at compile time rather than run time.

This module also provides some helper functions for using Moose types outside of attribute declarations.

If you mix string-based names with types created by this module, it will warn, with a few exceptions. If you are declaring a `class_type()` or `role_type()` within your type library, or if you use a fully qualified name like `"MyApp::Foo"`.

LIBRARY DEFINITION

A MooseX::Types is just a normal Perl module. Unlike Moose itself, it does not install `use strict` and `use warnings` in your class by default, so this is up to you.

The only thing a library is required to do is

```
use MooseX::Types -declare => \@types;
```

with `@types` being a list of types you wish to define in this library. This line will install a proper base class in your package as well as the full set of handlers for your declared types. It will then hand control over to Moose::Util::TypeConstraints' `import` method to export the functions you will need to declare your types.

If you want to use Moose' built-in types (e.g. for subtyping) you will want to

```
use MooseX::Types::Moose @types;
```

to import the helpers from the shipped MooseX::Types::Moose library which can export all types that come with Moose.

You will have to define coercions for your types or your library won't export a ''to_$type'' coercion helper for it.

Note that you currently cannot define types containing `::`, since exporting would be a problem.

You also don't need to use `warnings` and `strict`, since the definition of a library automatically exports those.

LIBRARY USAGE

You can import the ''type helpers'' of a library by `use`ing it with a list of types to import as arguments. If you want all of them, use the `:all` tag. For example:

```
use MyLibrary      ':all';
use MyOtherLibrary qw( TypeA TypeB );
```

MooseX::Types comes with a library of Moose' built-in types called MooseX::Types::Moose.

The exporting mechanism is, since version 0.5, implemented via a wrapper around Sub::Exporter. This means you can do something like this:

```
use MyLibrary TypeA => { -as => 'MyTypeA' },
              TypeB => { -as => 'MyTypeB' };
```

TYPE HANDLER FUNCTIONS

`$type`

A constant with the name of your type. It contains the type's fully qualified name. Takes no value, as all constants.

is_$type

This handler takes a value and tests if it is a valid value for this `$type`. It will return true or false.

to_$type

A handler that will take a value and coerce it into the `$type`. It will return a false value if the type could not be coerced.

Important Note: This handler will only be exported for types that can do type coercion. This has the advantage that a coercion to a type that has not defined any coercions will lead to a compile-time error.

WRAPPING A LIBRARY

You can define your own wrapper subclasses to manipulate the behaviour of a set of library exports. Here is an example:

```
package MyWrapper;
use strict;
use MRO::Compat;
use base 'MooseX::Types::Wrapper';

sub coercion_export_generator {
    my $class = shift;
    my $code = $class->next::method(@_);
    return sub {
```

```
            my $value = $code->(@_);
            warn "Coercion returned undef!"
                unless defined $value;
            return $value;
        };
    }

    1;
```

This class wraps the coercion generator (e.g., `to_Int()`) and warns if a coercion returned an undefined value. You can wrap any library with this:

```
    package Foo;
    use strict;
    use MyWrapper MyLibrary => [qw( Foo Bar )],
                  Moose     => [qw( Str Int )];

    ...
    1;
```

The `Moose` library name is a special shortcut for MooseX::Types::Moose.

Generator methods you can overload

type_export_generator(`$short`, `$full`)
: Creates a closure returning the type's Moose::Meta::TypeConstraint object.

check_export_generator(`$short`, `$full`, `$undef_message`)
: This creates the closure used to test if a value is valid for this type.

coercion_export_generator(`$short`, `$full`, `$undef_message`)
: This is the closure that's doing coercions.

Provided Parameters

`$short`
: The short, exported name of the type.

`$full`
: The fully qualified name of this type as Moose knows it.

`$undef_message`
: A message that will be thrown when type functionality is used but the type does not yet exist.

RECURSIVE SUBTYPES

As of version 0.08, Moose::Types has experimental support for Recursive subtypes. This will allow:

```
    subtype Tree() => as HashRef[Str|Tree];
```

Which validates things like:

```
    {key=>'value'};
    {key=>{subkey1=>'value', subkey2=>'value'}}
```

And so on. This feature is new and there may be lurking bugs so don't be afraid to hunt me down with patches and test cases if you have trouble.

NOTES REGARDING TYPE UNIONS

MooseX::Types uses MooseX::Types::TypeDecorator to do some overloading which generally allows you to easily create union types:

```
    subtype StrOrArrayRef,
        as Str|ArrayRef;
```

As with parameterized constraints, this overloading extends to modules using the types you define in a type library.

```
    use Moose;
    use MooseX::Types::Moose qw(HashRef Int);

    has 'attr' => ( isa => HashRef | Int );
```

And everything should just work as you'd think.

METHODS

import

Installs the MooseX::Types::Base class into the caller and exports types according to the specification described in ''LIBRARY DEFINITION''. This will continue to Moose::Util::TypeConstraints' `import` method to export helper functions you will need to declare your types.

type_export_generator

Generate a type export, e.g. `Int()`. This will return either a Moose::Meta::TypeConstraint object, or alternatively a MooseX::Types::UndefinedType object if the type was not yet defined.

create_arged_type_constraint ($name, @args)

Given a String `$name` with `@args` find the matching type constraint and parameterize it with `@args`.

create_base_type_constraint ($name)

Given a String `$name`, find the matching type constraint.

create_type_decorator ($type_constraint)

Given a `$type_constraint`, return a lightweight MooseX::Types::TypeDecorator instance.

coercion_export_generator

This generates a coercion handler function, e.g. `to_Int($value)`.

check_export_generator

Generates a constraint check closure, e.g. `is_Int($value)`.

CAVEATS

The following are lists of gotchas and their workarounds for developers coming from the standard string based type constraint names

Uniqueness

A library makes the types quasi-unique by prefixing their names with (by default) the library package name. If you're only using the type handler functions provided by MooseX::Types, you shouldn't ever have to use a type's actual full name.

Argument separation ('=>' versus ',')

The perlop manpage has this to say about the '=>' operator: ''The => operator is a synonym for the comma, but forces any word (consisting entirely of word characters) to its left to be interpreted as a string (as of 5.001). This includes words that might otherwise be considered a constant or function call.''

Due to this stringification, the following will NOT work as you might think:

```
subtype StrOrArrayRef => as Str | ArrayRef;
```

The `StrOrArrayRef` type will have its stringification activated — this causes the subtype to not be created. Since the bareword type constraints are not strings you really should not try to treat them that way. You will have to use the ',' operator instead. The authors of this package realize that all the Moose documentation and examples nearly uniformly use the '=>' version of the comma operator and this could be an issue if you are converting code.

Patches welcome for discussion.

Compatibility with Sub::Exporter

If you want to use Sub::Exporter with a Type Library, you need to make sure you export all the type constraints declared AS WELL AS any additional export targets. For example if you do:

```
package TypeAndSubExporter;

use MooseX::Types::Moose qw(Str);
use MooseX::Types -declare => [qw(MyStr)];
use Sub::Exporter -setup => { exports => [qw(something)] };

subtype MyStr, as Str;

sub something {
    return 1;
```

```
}

# then in another module ...

package Foo;
use TypeAndSubExporter qw(MyStr);
```

You'll get a `"MyStr" is not exported by the TypeAndSubExporter module` error. It can be worked around by:

```
- use Sub::Exporter -setup => { exports => [ qw(something) ] };
+ use Sub::Exporter -setup => { exports => [ qw(something MyStr) ] };
```

This is a workaround and I am exploring how to make these modules work better together. I realize this workaround will lead a lot of duplication in your export declarations and will be onerous for large type libraries. Patches and detailed test cases welcome. See the tests directory for a start on this.

COMBINING TYPE LIBRARIES

You may want to combine a set of types for your application with other type libraries, like MooseX::Types::Moose or MooseX::Types::Common::String.

The MooseX::Types::Combine module provides a simple API for combining a set of type libraries together.

SEE ALSO

Moose, Moose::Util::TypeConstraints, MooseX::Types::Moose, Sub::Exporter

ACKNOWLEDGEMENTS

Many thanks to the #moose cabal on irc.perl.org.

SUPPORT

Bugs may be submitted through the RT bug tracker <https://rt.cpan.org/Public/Dist/Display.html?Name=MooseX-Types> (or bug-MooseX-Types AT rt DOT cpan DOT org <mailto:bug-MooseX-Types AT rt DOT cpan DOT org>).

There is also a mailing list available for users of this distribution, at <http://lists.perl.org/list/moose.html>.

There is also an irc channel available for users of this distribution, at #moose on irc.perl.org <irc://irc.perl.org/#moose>.

AUTHOR

Robert ‘‘phaylon’’ Sedlacek <rs AT 474 DOT at>

CONTRIBUTORS

- Karen Etheridge <ether AT cpan DOT org>
- Dave Rolsky <autarch AT urth DOT org>
- John Napiorkowski <jjnapiork AT cpan DOT org>
- Robert ’phaylon’ Sedlacek <phaylon AT cpan DOT org>
- Rafael Kitover <rkitover AT cpan DOT org>
- Florian Ragwitz <rafl AT debian DOT org>
- Matt S Trout <mst AT shadowcat DOT co DOT uk>
- Tomas Doran (t0m) <bobtfish AT bobtfish DOT net>
- Jesse Luehrs <doy AT tozt DOT net>
- Mark Fowler <mark AT twoshortplanks DOT com>
- Hans Dieter Pearcey <hdp AT weftsoar DOT net>
- Graham Knop <haarg AT haarg DOT org>
- Paul Fenwick <pjf AT perltraining DOT com DOT au>
- Kent Fredric <kentfredric AT gmail DOT com>

- Justin Hunter <justin DOT d DOT hunter AT gmail DOT com>

COPYRIGHT AND LICENCE