NAME

Type::Utils – utility functions to make defining and using type constraints a little easier

SYNOPSIS

```
package Types::Mine;

use Type::Library -base;
use Type::Utils -all;

BEGIN { extends "Types::Standard" };

declare "AllCaps",
   as "Str",
   where { uc($_) eq $_ },
   inline_as { my $varname = $_[1]; "uc($varname) eq $varname" };

coerce "AllCaps",
   from "Str", via { uc($_) };
```

STATUS

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

DESCRIPTION

This module provides utility functions to make defining and using type constraints a little easier.

Type declaration functions

Many of the following are similar to the similarly named functions described in Moose::Util::TypeConstraints.

`declare $name, %options`
`declare %options`

Declare a named or anonymous type constraint. Use `as` and `where` to specify the parent type (if any) and (possibly) refine its definition.

```
declare EvenInt, as Int, where { $_ % 2 == 0 };

my $EvenInt = declare as Int, where { $_ % 2 == 0 };
```

NOTE: If the caller package inherits from Type::Library then any non-anonymous types declared in the package will be automatically installed into the library.

Hidden gem: if you're inheriting from a type constraint that includes some coercions, you can include `coercion => 1` in the `%options` hash to inherit the coercions.

`subtype $name, %options`
`subtype %options`

Declare a named or anonymous type constraint which is descended from an existing type constraint. Use `as` and `where` to specify the parent type and refine its definition.

Actually, you should use `declare` instead; this is just an alias.

This function is not exported by default.

`type $name, %options`
`type %options`

Declare a named or anonymous type constraint which is not descended from an existing type constraint. Use `where` to provide a coderef that constrains values.

Actually, you should use `declare` instead; this is just an alias.

This function is not exported by default.

`as $parent`
: Used with `declare` to specify a parent type constraint:

```
declare EvenInt, as Int, where { $_ % 2 == 0 };
```

`where { BLOCK }`
: Used with `declare` to provide the constraint coderef:

```
declare EvenInt, as Int, where { $_ % 2 == 0 };
```

The coderef operates on `$_`, which is the value being tested.

`message { BLOCK }`
: Generate a custom error message when a value fails validation.

```
declare EvenInt,
   as Int,
   where { $_ % 2 == 0 },
   message {
      Int->validate($_) or "$_ is not divisible by two";
   };
```

Without a custom message, the messages generated by Type::Tiny are along the lines of *Value "33" did not pass type constraint "EvenInt"*, which is usually reasonable.

`inline_as { BLOCK }`
: Generate a string of Perl code that can be used to inline the type check into other functions. If your type check is being used within a Moose or Moo constructor or accessor methods, or used by Type::Params, this can lead to significant performance improvements.

```
declare EvenInt,
   as Int,
   where { $_ % 2 == 0 },
   inline_as {
      my ($constraint, $varname) = @_;
      my $perlcode =
         $constraint->parent->inline_check($varname)
         . "&& ($varname % 2 == 0)";
      return $perlcode;
   };

warn EvenInt->inline_check('$xxx');  # demonstration
```

Experimental: your `inline_as` block can return a list, in which case these will be smushed together with "&&". The first item on the list may be undef, in which case the undef will be replaced by the inlined parent type constraint. (And will throw an exception if there is no parent.)

```
declare EvenInt,
   as Int,
   where { $_ % 2 == 0 },
   inline_as {
      return (undef, "($_ % 2 == 0)");
   };
```

Returning a list like this is considered experimental, is not tested very much, and I offer no guarantees that it will necessarily work with Moose/Mouse/Moo.

`class_type $name, { class => $package, %options }`
`class_type { class => $package, %options }`

`class_type $name`
: Shortcut for declaring a Type::Tiny::Class type constraint.

  If `$package` is omitted, is assumed to be the same as `$name`. If `$name` contains "::" (which would be an invalid name as far as Type::Tiny is concerned), this will be removed.

  So for example, `class_type("Foo::Bar")` declares a Type::Tiny::Class type constraint named "FooBar" which constrains values to objects blessed into the "Foo::Bar" package.

`role_type $name, { role => $package, %options }`
`role_type { role => $package, %options }`
`role_type $name`
: Shortcut for declaring a Type::Tiny::Role type constraint.

  If `$package` is omitted, is assumed to be the same as `$name`. If `$name` contains "::" (which would be an invalid name as far as Type::Tiny is concerned), this will be removed.

`duck_type $name, \@methods`
`duck_type \@methods`
: Shortcut for declaring a Type::Tiny::Duck type constraint.

`union $name, \@constraints`
`union \@constraints`
: Shortcut for declaring a Type::Tiny::Union type constraint.

`enum $name, \@values`
`enum \@values`
: Shortcut for declaring a Type::Tiny::Enum type constraint.

`intersection $name, \@constraints`
`intersection \@constraints`
: Shortcut for declaring a Type::Tiny::Intersection type constraint.

Coercion declaration functions

Many of the following are similar to the similarly named functions described in Moose::Util::TypeConstraints.

`coerce $target, @coercions`
: Add coercions to the target type constraint. The list of coercions is a list of type constraint, conversion code pairs. Conversion code can be either a string of Perl code or a coderef; in either case the value to be converted is `$_`.

`from $source`
: Sugar to specify a type constraint in a list of coercions:

```
coerce EvenInt, from Int, via { $_ * 2 };  # As a coderef...
coerce EvenInt, from Int, q { $_ * 2 };    # or as a string!
```

`via { BLOCK }`
: Sugar to specify a coderef in a list of coercions.

`declare_coercion $name, \%opts, $type1, $code1, ...`
`declare_coercion \%opts, $type1, $code1, ...`
: Declares a coercion that is not explicitly attached to any type in the library. For example:

```
declare_coercion "ArrayRefFromAny", from "Any", via { [$_] };
```

  This coercion will be exportable from the library as a Type::Coercion object, but the ArrayRef type exported by the library won't automatically use it.

  Coercions declared this way are immutable (frozen).

`to_type $type`
: Used with `declare_coercion` to declare the target type constraint for a coercion, but still without explicitly attaching the coercion to the type constraint:

```
declare_coercion "ArrayRefFromAny",
   to_type "ArrayRef",
   from "Any", via { [$_] };
```

You should pretty much always use this when declaring an unattached coercion because it's exceedingly useful for a type coercion to know what it will coerce to – this allows it to skip coercion when no coercion is needed (e.g. avoiding coercing `[]` to `[ [] ]`) and allows `assert_coerce` to work properly.

Type library management

`extends @libraries`

Indicates that this type library extends other type libraries, importing their type constraints.

Should usually be executed in a `BEGIN` block.

This is not exported by default because it's not fun to export it to Moo, Moose or Mouse classes! `use Type::Utils -all` can be used to import it into your type library.

Other

`match_on_type $value => ($type => \&action, ..., \&default?)`

Something like a `switch`/`case` or `given`/`when` construct. Dispatches along different code paths depending on the type of the incoming value. Example blatantly stolen from the Moose documentation:

```
sub to_json
{
   my $value = shift;

   return match_on_type $value => (
      HashRef() => sub {
         my $hash = shift;
         '{ '
            . (
            join ", " =>
            map { '"' . $_ . '" : ' . to_json( $hash->{$_} ) }
            sort keys %$hash
         ) . ' }';
      },
      ArrayRef() => sub {
         my $array = shift;
         '[ '.( join ", " => map { to_json($_) } @$array ).' ]';
      },
      Num()   => q {$_},
      Str()   => q { '"' . $_ . '"' },
      Undef() => q {'null'},
      => sub { die "$_ is not acceptable json type" },
   );
}
```

Note that unlike Moose, code can be specified as a string instead of a coderef. (e.g. for `Num`, `Str` and `Undef` above.)

For improved performance, try `compile_match_on_type`.

This function is not exported by default.

`my $coderef = compile_match_on_type($type => \&action, ..., \&default?)`

Compile a `match_on_type` block into a coderef. The following JSON converter is about two orders of magnitude faster than the previous example:

```
sub to_json;
*to_json = compile_match_on_type(
   HashRef() => sub {
      my $hash = shift;
      '{ '
         . (
         join ", " =>
         map { '"' . $_ . '" : ' . to_json( $hash->{$_} ) }
         sort keys %$hash
      ) . ' }';
   },
   ArrayRef() => sub {
      my $array = shift;
      '[ '.( join ", " => map { to_json($_) } @$array ).' ]';
   },
   Num()   => q {$_},
   Str()   => q { '"' . $_ . '"' },
   Undef() => q {'null'},
   => sub { die "$_ is not acceptable json type" },
);
```

Remember to store the coderef somewhere fairly permanent so that you don't compile it over and over. `state` variables (in Perl >= 5.10) are good for this. (Same sort of idea as Type::Params.)

This function is not exported by default.

`my $coderef = classifier(@types)`

Returns a coderef that can be used to classify values according to their type constraint. The coderef, when passed a value, returns a type constraint which the value satisfies.

```
use feature qw( say );
use Type::Utils qw( classifier );
use Types::Standard qw( Int Num Str Any );

my $classifier = classifier(Str, Int, Num, Any);

say $classifier->( "42"  )->name;   # Int
say $classifier->( "4.2" )->name;   # Num
say $classifier->( []    )->name;   # Any
```

Note that, for example, "42" satisfies Int, but it would satisfy the type constraints Num, Str, and Any as well. In this case, the classifier has picked the most specific type constraint that "42" satisfies.

If no type constraint is satisfied by the value, then the classifier will return undef.

`dwim_type($string, %options)`

Given a string like "ArrayRef[Int|CodeRef]", turns it into a type constraint object, hopefully doing what you mean.

It uses the syntax of Type::Parser. Firstly the Type::Registry for the caller package is consulted; if that doesn't have a match, Types::Standard is consulted for standard type constraint names.

If none of the above yields a type constraint, and the caller class is a Moose-based class, then `dwim_type` attempts to look the type constraint up in the Moose type registry. If it's a Mouse-based class, then the Mouse type registry is used instead.

If no type constraint can be found via these normal methods, several fallbacks are available:

`lookup_via_moose`
: Lookup in Moose registry even if caller is non-Moose class.

`lookup_via_mouse`
: Lookup in Mouse registry even if caller is non-Mouse class.

`make_class_type`
: Create a new Type::Tiny::Class constraint.

`make_role_type`
: Create a new Type::Tiny::Role constraint.

You can alter which should be attempted, and in which order, by passing an option to `dwim_type`:

```
my $type = Type::Utils::dwim_type(
   "ArrayRef[Int]",
   fallback      => [ "lookup_via_mouse" , "make_role_type" ],
);
```

For historical reasons, by default the fallbacks attempted are:

```
lookup_via_moose, lookup_via_mouse, make_class_type
```

You may set `fallback` to an empty arrayref to avoid using any of these fallbacks.

You can specify an alternative for the caller using the `for` option.

```
my $type = dwim_type("ArrayRef", for => "Moose::Object");
```

While it's probably better overall to use the proper Type::Registry interface for resolving type constraint strings, this function often does what you want.

It should never die if it fails to find a type constraint (but may die if the type constraint string is syntactically malformed), preferring to return undef.

This function is not exported by default.

`english_list(\$conjunction, @items)`
: Joins the items with commas, placing a conjunction before the final item. The conjunction is optional, defaulting to "and".

```
english_list(qw/foo bar baz/);        # "foo, bar, and baz"
english_list(\"or", qw/quux quuux/); # "quux or quuux"
```

This function is not exported by default.

EXPORT

By default, all of the functions documented above are exported, except `subtype` and `type` (prefer `declare` instead), `extends`, `dwim_type`, `match_on_type`/`compile_match_on_type`, `classifier`, and `english_list`.

This module uses Exporter::Tiny; see the documentation of that module for tips and tricks importing from Type::Utils.

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, Types::Standard, Type::Coercion.

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

Moose::Util::TypeConstraints, Mouse::Util::TypeConstraints.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES