## NAME

Types::Standard − bundled set of built−in types for Type::Tiny

## STATUS

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

## DESCRIPTION

Type::Tiny bundles a few types which seem to be useful.

### Moose-like

The following types are similar to those described in Moose::Util::TypeConstraints.

`Any`
Absolutely any value passes this type constraint (even undef).

`Item`
Essentially the same as `Any`. All other type constraints in this library inherit directly or indirectly from `Item`.

`Bool`
Values that are reasonable booleans. Accepts 1, 0, the empty string and undef.

``Maybe[`a]``
Given another type constraint, also accepts undef. For example, `Maybe[Int]` accepts all integers plus undef.

`Undef`
Only undef passes this type constraint.

`Defined`
Only undef fails this type constraint.

`Value`
Any defined, non-reference value.

`Str`
Any string.

(The only difference between `Value` and `Str` is that the former accepts typeglobs and vstrings.)

Other customers also bought: `StringLike` from Types::TypeTiny.

`Num`
See `LaxNum` and `StrictNum` below.

`Int`
An integer; that is a string of digits 0 to 9, optionally prefixed with a hyphen-minus character.

`ClassName`
The name of a loaded package. The package must have `@ISA` or `$VERSION` defined, or must define at least one sub to be considered a loaded package.

`RoleName`
Like `ClassName`, but the package must *not* define a method called `new`. This is subtly different from Moose's type constraint of the same name; let me know if this causes you any problems. (I can't promise I'll change anything though.)

``Ref[`a]``
Any defined reference value, including blessed objects.

Unlike Moose, `Ref` is a parameterized type, allowing Scalar::Util::reftype checks, a la

```
Ref["HASH"]  # hashrefs, including blessed hashrefs
```

``ScalarRef[`a]``
A value where `ref($value) eq "SCALAR" or ref($value) eq "REF"`.

If parameterized, the referred value must pass the additional constraint. For example, `ScalarRef[Int]` must be a reference to a scalar which holds an integer value.

`ArrayRef[`a]`
A value where `ref($value) eq "ARRAY"`.

If parameterized, the elements of the array must pass the additional constraint. For example, `ArrayRef[Num]` must be a reference to an array of numbers.

Other customers also bought: `ArrayLike` from Types::TypeTiny.

`HashRef[`a]`
A value where `ref($value) eq "HASH"`.

If parameterized, the values of the hash must pass the additional constraint. For example, `HashRef[Num]` must be a reference to an hash where the values are numbers. The hash keys are not constrained, but Perl limits them to strings; see `Map` below if you need to further constrain the hash values.

Other customers also bought: `HashLike` from Types::TypeTiny.

`CodeRef`
A value where `ref($value) eq "CODE"`.

Other customers also bought: `CodeLike` from Types::TypeTiny.

`RegexpRef`
A reference where `re::is_regexp($value)` is true, or a blessed reference where `$value->isa("Regexp")` is true.

`GlobRef`
A value where `ref($value) eq "GLOB"`.

`FileHandle`
A file handle.

`Object`
A blessed object.

(This also accepts regexp refs.)

**Structured**
OK, so I stole some ideas from MooseX::Types::Structured.

`Map[`k, `v]`
Similar to `HashRef` but parameterized with type constraints for both the key and value. The constraint for keys would typically be a subtype of `Str`.

`Tuple[...]`
Subtype of `ArrayRef`, accepting a list of type constraints for each slot in the array.

`Tuple[Int, HashRef]` would match `[1, {}]` but not `[{}, 1]`.

`Dict[...]`
Subtype of `HashRef`, accepting a list of type constraints for each slot in the hash.

For example `Dict[name => Str, id => Int]` allows `{ name => "Bob", id => 42 }`.

`Optional[`a]`
Used in conjunction with `Dict` and `Tuple` to specify slots that are optional and may be omitted (but not necessarily set to an explicit undef).

`Dict[name => Str, id => Optional[Int]]` allows `{ name => "Bob" }` but not `{ name => "Bob", id => "BOB" }`.

Note that any use of `Optional[`a]` outside the context of parameterized `Dict` and `Tuple` type constraints makes little sense, and its behaviour is undefined. (An exception: it is used by Type::Params for a similar purpose to how it's used in `Tuple`.)

This module also exports a `slurpy` function, which can be used as follows.

It can cause additional trailing values in a `Tuple` to be slurped into a structure and validated. For example, slurping into an ArrayRef:

```
my $type = Tuple[Str, slurpy ArrayRef[Int]];

$type->( ["Hello"] );                # ok
$type->( ["Hello", 1, 2, 3] );       # ok
$type->( ["Hello", [1, 2, 3]] );     # not ok
```

Or into a hashref:

```
my $type2 = Tuple[Str, slurpy Map[Int, RegexpRef]];

$type2->( ["Hello"] );                               # ok
$type2->( ["Hello", 1, qr/one/i, 2, qr/two/] );      # ok
```

It can cause additional values in a `Dict` to be slurped into a hashref and validated:

```
my $type3 = Dict[ values => ArrayRef, slurpy HashRef[Str] ];

$type3->( { values => [] } );                        # ok
$type3->( { values => [], name => "Foo" } );         # ok
$type3->( { values => [], name => [] } );            # not ok
```

In either `Tuple` or `Dict`, `slurpy Any` can be used to indicate that additional values are acceptable, but should not be constrained in any way.

`slurpy Any` is an optimized code path. Although the following are essentially equivalent checks, the former should run a lot faster:

```
Tuple[Int, slurpy Any]
Tuple[Int, slurpy ArrayRef]
```

**Objects**

OK, so I stole some ideas from MooX::Types::MooseLike::Base.

`InstanceOf[`a]`

Shortcut for a union of Type::Tiny::Class constraints.

`InstanceOf["Foo", "Bar"]` allows objects blessed into the `Foo` or `Bar` classes, or subclasses of those.

Given no parameters, just equivalent to `Object`.

`ConsumerOf[`a]`

Shortcut for an intersection of Type::Tiny::Role constraints.

`ConsumerOf["Foo", "Bar"]` allows objects where `$o->DOES("Foo")` and `$o->DOES("Bar")` both return true.

Given no parameters, just equivalent to `Object`.

`HasMethods[`a]`

Shortcut for a Type::Tiny::Duck constraint.

`HasMethods["foo", "bar"]` allows objects where `$o->can("foo")` and `$o->can("bar")` both return true.

Given no parameters, just equivalent to `Object`.

**More**

There are a few other types exported by this function:

`Overload[`a]`

With no parameters, checks that the value is an overloaded object. Can be given one or more string parameters, which are specific operations to check are overloaded. For example, the following checks for objects which overload addition and subtraction.

```
Overload["+", "-"]
```

`Tied[`a]`

A reference to a tied scalar, array or hash.

Can be parameterized with a type constraint which will be applied to the object returned by the `tied()` function. As a convenience, can also be parameterized with a string, which will be inflated to a Type::Tiny::Class.

```
use Types::Standard qw(Tied);
use Type::Utils qw(class_type);

my $My_Package = class_type { class => "My::Package" };

tie my %h, "My::Package";
\%h ~~ Tied;                   # true
\%h ~~ Tied[ $My_Package ];    # true
\%h ~~ Tied["My::Package"];    # true

tie my $s, "Other::Package";
\$s ~~ Tied;                   # true
$s  ~~ Tied;                   # false !!
```

If you need to check that something is specifically a reference to a tied hash, use an intersection:

```
use Types::Standard qw( Tied HashRef );

my $TiedHash = (Tied) & (HashRef);

tie my %h, "My::Package";
tie my $s, "Other::Package";

\%h ~~ $TiedHash;     # true
\$s ~~ $TiedHash;     # false
```

`StrMatch[`a]`

A string that matches a regular expression:

```
declare "Distance",
   as StrMatch[ qr{^([0-9]+)\s*(mm|cm|m|km)$} ];
```

You can optionally provide a type constraint for the array of subexpressions:

```
declare "Distance",
   as StrMatch[
      qr{^([0-9]+)\s*(.+)$},
      Tuple[
         Int,
         enum(DistanceUnit => [qw/ mm cm m km /]),
      ],
   ];
```

Here's an example using Regexp::Common:

```
package Local::Host {
   use Moose;
   use Regexp::Common;
   has ip_address => (
      is         => 'ro',
      required   => 1.
      isa        => StrMatch[/^$RE{net}{IPv4}$/],
      default    => '127.0.0.1',
   );
}
```

On certain versions of Perl, type constraints of the forms `StrMatch[qr/../` and `StrMatch[qr/\A..\z/` with any number of intervening dots can be optimized to simple length checks.

`Enum[`a]`

As per MooX::Types::MooseLike::Base:

```
has size => (is => "ro", isa => Enum[qw( S M L XL XXL )]);
```

`OptList`

An arrayref of arrayrefs in the style of Data::OptList output.

`LaxNum`, `StrictNum`

In Moose 2.09, the `Num` type constraint implementation was changed from being a wrapper around Scalar::Util's `looks_like_number` function to a stricter regexp (which disallows things like "−Inf" and "Nan").

Types::Standard provides *both* implementations. `LaxNum` is measurably faster.

The `Num` type constraint is currently an alias for `LaxNum` unless you set the `PERL_TYPES_STANDARD_STRICTNUM` environment variable to true before loading Types::Standard, in which case it becomes an alias for `StrictNum`. The constant `Types::Standard::STRICTNUM` can be used to check if `Num` is being strict.

Most people should probably use `Num` or `StrictNum`. Don't explicitly use `LaxNum` unless you specifically need an attribute which will accept things like "Inf".

`CycleTuple[`a]`

Similar to Tuple, but cyclical.

```
CycleTuple[Int, HashRef]
```

will allow `[1,{}]` and `[1,{},2,{}]` but disallow `[1,{},2]` and `[1,{},2,[]]`.

I think you understand CycleTuples already.

Currently `Optional` and `slurpy` parameters are forbidden. There are fairly limited use cases for them, and it's not exactly clear what they should mean.

The following is an efficient way of checking for an even-sized arrayref:

```
CycleTuple[Any, Any]
```

The following is an arrayref which would be suitable for coercing to a hashref:

```
CycleTuple[Str, Any]
```

All the examples so far have used two parameters, but the following is also a possible CycleTuple:

```
CycleTuple[Str, Int, HashRef]
```

This will be an arrayref where the 0th, 3rd, 6th, etc values are strings, the 1st, 4th, 7th, etc values are integers, and the 2nd, 5th, 8th, etc values are hashrefs.

### Coercions

Most of the types in this type library have no coercions by default. The exception is `Bool` as of Types::Standard 1.003_003, which coerces from `Any` via `!!$_`.

Some standalone coercions may be exported. These can be combined with type constraints using the `plus_coercions` method.

`MkOpt`
: A coercion from `ArrayRef`, `HashRef` or `Undef` to `OptList`. Example usage in a Moose attribute:

```
use Types::Standard qw( OptList MkOpt );

has options => (
   is     => "ro",
   isa    => OptList->plus_coercions( MkOpt ),
   coerce => 1,
);
```

`Split[`a]`
: Split a string on a regexp.

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

has name => (
   is     => "ro",
   isa    => (ArrayRef[Str])->plus_coercions(Split[qr/\s/]),
   coerce => 1,
);
```

`Join[`a]`
: Join an array of strings with a delimiter.

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

my $FileLines = Str->plus_coercions(Join["\n"]);

has file_contents => (
   is     => "ro",
   isa    => $FileLines,
   coerce => 1,
);
```

### Constants

`Types::Standard::STRICTNUM`
: Indicates whether `Num` is an alias for `StrictNum`. (It is usually an alias for `LaxNum`.)

### Environment

`PERL_TYPES_STANDARD_STRICTNUM`
: Switches to more strict regexp-based number checking instead of using `looks_like_number`.

`PERL_TYPE_TINY_XS`
: If set to false, can be used to suppress the loading of XS implementions of some type constraints.

`PERL_ONLY`
: If `PERL_TYPE_TINY_XS` does not exist, can be set to true to suppress XS usage similarly. (Several other CPAN distributions also pay attention to this environment variable.)

## 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, Type::Coercion.

Moose::Util::TypeConstraints, Mouse::Util::TypeConstraints, MooseX::Types::Structured.

Types::XSD provides some type constraints based on XML Schema's data types; this includes constraints for ISO8601−formatted datetimes, integer ranges (e.g. `PositiveInteger[maxInclusive=>10]` and so on.

Types::Encodings provides `Bytes` and `Chars` type constraints that were formerly found in Types::Standard.

Types::Common::Numeric and Types::Common::String provide replacements for MooseX::Types::Common.

## AUTHOR

Toby Inkster <tobyink@cpan.org>.

## COPYRIGHT AND LICENCE

## DISCLAIMER OF WARRANTIES