NAME

Type::Params – Params::Validate–like parameter validation using Type::Tiny type constraints and coercions

SYNOPSIS

```
use v5.12;
use strict;
use warnings;

package Horse {
  use Moo;
  use Types::Standard qw( Object );
  use Type::Params qw( compile );
  use namespace::autoclean;

  ...;   # define attributes, etc

  sub add_child {
    state $check = compile( Object, Object );  # method signature

    my ($self, $child) = $check->(@_);         # unpack @_
    push @{ $self->children }, $child;

    return $self;
  }
}

package main;

my $boldruler = Horse->new;

$boldruler->add_child( Horse->new );

$boldruler->add_child( 123 );   # dies (123 is not an Object!)
```

STATUS

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

DESCRIPTION

This documents the details of the Type::Params package. Type::Tiny::Manual is a better starting place if you're new.

Type::Params uses Type::Tiny constraints to validate the parameters to a sub. It takes the slightly unorthodox approach of separating validation into two stages:

1. Compiling the parameter specification into a coderef; then
2. Using the coderef to validate parameters.

The first stage is slow (it might take a couple of milliseconds), but you only need to do it the first time the sub is called. The second stage is fast; according to my benchmarks faster even than the XS version of Params::Validate.

If you're using a modern version of Perl, you can use the `state` keyword which was a feature added to Perl in 5.10. If you're stuck on Perl 5.8, the example from the SYNOPSIS could be rewritten as:

```
my $add_child_check;
sub add_child {
  $add_child_check ||= compile( Object, Object );

  my ($self, $child) = $add_child_check->(@_);  # unpack @_
```

```
    push @{ $self->children }, $child;

    return $self;
  }
```

Not quite as neat, but not awful either.

If you don't like the two step, there's a shortcut reducing it to one step:

```
  use Type::Params qw( validate );

  sub add_child {
    my ($self, $child) = validate(\@_, Object, Object);
    push @{ $self->children }, $child;
    return $self;
  }
```

Type::Params has a few tricks up its sleeve to make sure performance doesn't suffer too much with the shortcut, but it's never going to be as fast as the two stage compile/execute.

Functions

compile(@spec)

Given specifications for positional parameters, compiles a coderef that can check against them.

The generalized form of specifications for positional parameters is:

```
state $check = compile(
  \%general_opts,
  $type_for_arg_1, \%opts_for_arg_1,
  $type_for_arg_2, \%opts_for_arg_2,
  $type_for_arg_3, \%opts_for_arg_3,
  ...,
  slurpy($slurpy_type),
);
```

If a hashref of options is empty, it can simply be omitted. Much of the time, you won't need to specify any options.

```
# In this example, we omit all the hashrefs
#
my $check = compile(
  Str,
  Int,
  Optional[ArrayRef],
);

my ($str, $int, $arr) = $check->("Hello", 42, []);    # ok
my ($str, $int, $arr) = $check->("", -1);             # ok
my ($str, $int, $arr) = $check->("", -1, "bleh");     # dies
```

The coderef returned (i.e. `$check`) will check the arguments passed to it conform to the spec (coercing them if appropriate), and return them as a list if they do. If they don't, it will throw an exception.

The first hashref, before any type constraints, is for general options which affect the entire compiled coderef. Currently supported general options are:

`want_source` **Bool**
: Instead of returning a coderef, return Perl source code string. Handy for debugging.

`want_details` **Bool**
: Instead of returning a coderef, return a hashref of stuff including the coderef. This is mostly for people extending Type::Params and I won't go into too many details about what else this hashref contains.

`description` **Str**
: Description of the coderef that will show up in stack traces. Defaults to ''parameter validation for X'' where X is the caller sub name.

`subname` **Str**
: If you wish to use the default description, but need to change the sub name, use this.

`caller_level` **Int**
: If you wish to use the default description, but need to change the caller level for detecting the sub name, use this.

The types for each parameter may be any Type::Tiny type constraint, or anything that Type::Tiny knows how to coerce into a Type::Tiny type constraint, such as a MooseX::Types type constraint or a coderef.

Type coercions are automatically applied for all types that have coercions.

If you wish to avoid coercions for a type, use Type::Tiny's `no_coercions` method.

```
my $check = compile(
  Int,
  ArrayRef->of(Bool)->no_coercions,
);
```

Note that having any coercions in a specification, even if they're not used in a particular check, will slightly slow down `$check` because it means that `$check` can't just check `@_` and return it unaltered if it's valid — it needs to build a new array to return.

Optional parameters can be given using the **Optional[]** type constraint. In the example above, the third parameter is optional. If it's present, it's required to be an arrayref, but if it's absent, it is ignored.

Optional parameters need to be *after* required parameters in the spec.

An alternative way to specify optional parameters is using a parameter options hashref.

```
my $check = compile(
  Str,
  Int,
  ArrayRef, { optional => 1 },
);
```

The following parameter options are supported:

`optional` **Bool**
: This is an alternative way of indicating that a parameter is optional.

  ```
  state $check = compile(
    Int,
    Int, { optional => 1 },
    Optional[Int],
  );
  ```

  The two are not *exactly* equivalent. The exceptions thrown will differ in the type name they mention. (**Int** versus **Optional[Int]**.)

`default` **CodeRef|Ref|Str|Undef**
: A default may be provided for a parameter.

  ```
  state $check = compile(
    Int,
    Int, { default => "666" },
    Int, { default => "999" },
  );
  ```

  Supported defaults are any strings (including numerical ones), `undef`, and empty hashrefs and arrayrefs. Non-empty hashrefs and arrayrefs are *not allowed as defaults*.

Alternatively, you may provide a coderef to generate a default value:

```
state $check = named(
  Int,
  Int, { default => sub { 6 * 111 } },
  Int, { default => sub { 9 * 111 } },
);
```

That coderef may generate any value, including non-empty arrayrefs and non-empty hashrefs. For undef, simple strings, numbers, and empty structures, avoiding using a coderef will make your parameter processing faster.

The default *will* be validated against the type constraint, and potentially coerced.

Note that having any defaults in a specification, even if they're not used in a particular check, will slightly slow down `$check` because it means that `$check` can't just check `@_` and return it unaltered if it's valid — it needs to build a new array to return.

As a special case, the numbers 0 and 1 may be used as shortcuts for **Optional[Any]** and **Any**.

```
# Positional parameters
state $check = compile(1, 0, 0);
my ($foo, $bar, $baz) = $check->(@_);  # $bar and $baz are optional
```

After any required and optional parameters may be a slurpy parameter. Any additional arguments passed to `$check` will be slurped into an arrayref or hashref and checked against the slurpy parameter. Defaults are not supported for slurpy parameters.

Example with a slurpy ArrayRef:

```
sub xyz {
  state $check = compile(Int, Int, slurpy ArrayRef[Int]);
  my ($foo, $bar, $baz) = $check->(@_);
}

xyz(1..5);  # $foo = 1
            # $bar = 2
            # $baz = [ 3, 4, 5 ]
```

Example with a slurpy HashRef:

```
my $check = compile(
  Int,
  Optional[Str],
  slurpy HashRef[Int],
);

my ($x, $y, $z) = $check->(1, "y", foo => 666, bar => 999);
# $x is 1
# $y is "y"
# $z is { foo => 666, bar => 999 }
```

Any type constraints derived from **ArrayRef** or **HashRef** should work, but a type does need to inherit from one of those because otherwise Type::Params cannot know what kind of structure to slurp the remaining arguments into.

slurpy Any is also allowed as a special case, and is treated as **slurpy ArrayRef**.

From Type::Params 1.005000 onwards, slurpy hashrefs can be passed in as a true hashref (which will be shallow cloned) rather than key-value pairs.

```
sub xyz {
  state $check = compile(Int, slurpy HashRef);
  my ($num, $hr) = $check->(@_);
  ...
}

xyz( 5,   foo => 1, bar => 2   );   # works
xyz( 5, { foo => 1, bar => 2 } );   # works from 1.005000
```

This feature is only implemented for slurpy hashrefs, not slurpy arrayrefs.

Note that having a slurpy parameter will slightly slow down `$check` because it means that `$check` can't just check `@_` and return it unaltered if it's valid — it needs to build a new array to return.

`validate(\@_, @spec)`

This example of `compile`:

```
sub foo {
  state $check = compile(@spec);
  my @args = $check->(@_);
  ...;
}
```

Can be written using `validate` as:

```
sub foo {
  my @args = validate(\@_, @spec);
  ...;
}
```

Performance using `compile` will *always* beat `validate` though.

`compile_named(@spec)`

`compile_named` is a variant of `compile` for named parameters instead of positional parameters.

The format of the specification is changed to include names for each parameter:

```
state $check = compile_named(
  \%general_opts,
  foo => $type_for_foo, \%opts_for_foo,
  bar => $type_for_bar, \%opts_for_bar,
  baz => $type_for_baz, \%opts_for_baz,
  ...,
  slurpy($slurpy_type),
);
```

The `$check` coderef will return a hashref.

```
my $check = compile_named(
  foo => Int,
  bar => Str, { default => "hello" },
);

my $args = $check->(foo => 42);
# $args->{foo} is 42
# $args->{bar} is "hello"
```

The `%general_opts` hash supports the same options as `compile` plus a few additional options:

`class` **ClassName**
: The check coderef will, instead of returning a simple hashref, call `$class->new($hashref)` and return the result.

`constructor` **Str**
: Specifies an alternative method name instead of `new` for the `class` option described above.

`class` **Tuple[ClassName, Str]**
: Shortcut for declaring both the `class` and `constructor` options at once.

`bless` **ClassName**
: Like `class`, but bypass the constructor and directly bless the hashref.

`named_to_list` **Bool**
: Instead of returning a hashref, return a hash slice.

```
myfunc(bar => "x", foo => "y");

sub myfunc {
   state $check = compile_named(
      { named_to_list => 1 },
      foo => Str, { optional => 1 },
      bar => Str, { optional => 1 },
   );
   my ($foo, $bar) = $check->(@_);
   ...; ## $foo is "y" and $bar is "x"
}
```

The order of keys for the hash slice is the same as the order of the names passed to `compile_named`. For missing named parameters, `undef` is returned in the list.

Basically in the above example, `myfunc` takes named parameters, but receieves positional parameters.

`named_to_list` **ArrayRef[Str]**
: As above, but explicitly specify the keys of the hash slice.

Like `compile`, the numbers 0 and 1 may be used as shortcuts for **Optional[Any]** and **Any**.

```
state $check = compile_named(foo => 1, bar => 0, baz => 0);
my $args = $check->(@_);  # $args->{bar} and $args->{baz} are optional
```

Slurpy parameters are supported as you'd expect.

slurpy Any is treated as **slurpy HashRef**.

`validate_named(\@_, @spec)`

Like `compile` has `validate`, `compile_named` has `validate_named`. Just like `validate`, it's the slower way to do things, so stick with `compile_named`.

`compile_named_oo(@spec)`

Here's a quick example function:

```
sub add_contact_to_database {
   state $check = compile_named(
      dbh     => Object,
      id      => Int,
      name    => Str,
   );
   my $arg = $check->(@_);

   my $sth = $arg->{db}->prepare('INSERT INTO contacts VALUES (?, ?)');
   $sth->execute($arg->{id}, $arg->{name});
}
```

Looks simple, right? Did you spot that it will always die with an error message *Can't call method "prepare" on an undefined value*?

This is because we defined a parameter called 'dbh' but later tried to refer to it as `$arg{db}`. Here, Perl gives us a pretty clear error, but sometimes the failures will be far more subtle. Wouldn't it be nice if instead we could do this?

```
sub add_contact_to_database {
   state $check = compile_named_oo(
      dbh     => Object,
      id      => Int,
      name    => Str,
   );
   my $arg = $check->(@_);

   my $sth = $arg->dbh->prepare('INSERT INTO contacts VALUES (?, ?)');
   $sth->execute($arg->id, $arg->name);
}
```

If we tried to call `$arg->db`, it would fail because there was no such method.

Well, that's exactly what `compile_named_oo` does.

As well as giving you nice protection against mistyped parameter names, It also looks kinda pretty, I think. Hash lookups are a little faster than method calls, of course (though Type::Params creates the methods using Class::XSAccessor if it's installed, so they're still pretty fast).

An optional parameter `foo` will also get a nifty `$arg->has_foo` predicate method. Yay!

`compile_named_oo` gives you some extra options for parameters.

```
sub add_contact_to_database {
   state $check = compile_named_oo(
      dbh     => Object,
      id      => Int,    { default => '0', getter => 'identifier' },
      name    => Str,    { optional => 1, predicate => 'has_name' },
   );
   my $arg = $check->(@_);

   my $sth = $arg->dbh->prepare('INSERT INTO contacts VALUES (?, ?)');
   $sth->execute($arg->identifier, $arg->name) if $arg->has_name;
}
```

`getter` **Str**
: The `getter` option lets you choose the method name for getting the argument value.

`predicate` **Str**
: The `predicate` option lets you choose the method name for checking the existence of an argument. By setting an explicit predicate method name, you can force a predicate method to be generated for non-optional arguments.

The objects returned by `compile_named_oo` are blessed into lightweight classes which have been generated on the fly. Don't expect the names of the classes to be stable or predictable. It's probably a bad idea to be checking `can`, `isa`, or `DOES` on any of these objects. If you're doing that, you've missed the point of them.

They don't have any constructor (`new` method). The `$check` coderef effectively *is* the constructor.

`validate_named_oo(\@_, @spec)`

This function doesn't even exist. :D

`multisig(@alternatives)`

Type::Params can export a `multisig` function that compiles multiple alternative signatures into one, and uses the first one that works:

```
state $check = multisig(
   [ Int, ArrayRef ],
   [ HashRef, Num ],
   [ CodeRef ],
);

my ($int, $arrayref) = $check->( 1, [] );      # okay
my ($hashref, $num)  = $check->( {}, 1.1 );    # okay
my ($code)           = $check->( sub { 1 } );  # okay

$check->( sub { 1 }, 1.1 );  # throws an exception
```

Coercions, slurpy parameters, etc still work.

The magic global `${^TYPE_PARAMS_MULTISIG}` is set to the index of the first signature which succeeded.

The present implementation involves compiling each signature independently, and trying them each (in their given order!) in an `eval` block. The only slightly intelligent part is that it checks if `scalar(@_)` fits into the signature properly (taking into account optional and slurpy parameters), and skips evals which couldn't possibly succeed.

It's also possible to list coderefs as alternatives in `multisig`:

```
state $check = multisig(
   [ Int, ArrayRef ],
   sub { ... },
   [ HashRef, Num ],
   [ CodeRef ],
   compile_named( needle => Value, haystack => Ref ),
);
```

The coderef is expected to die if that alternative should be abandoned (and the next alternative tried), or return the list of accepted parameters. Here's a full example:

```
sub get_from {
   state $check = multisig(
      [ Int, ArrayRef ],
      [ Str, HashRef ],
      sub {
         my ($meth, $obj);
         die unless is_Object($obj);
         die unless $obj->can($meth);
         return ($meth, $obj);
      },
   );

   my ($needle, $haystack) = $check->(@_);

   for (${^TYPE_PARAMS_MULTISIG}) {
      return $haystack->[$needle] if $_ == 0;
      return $haystack->{$needle} if $_ == 1;
      return $haystack->$needle   if $_ == 2;
   }
}

get_from(0, \@array);      # returns $array[0]
get_from('foo', \%hash);   # returns $hash{foo}
get_from('foo', $obj);     # returns $obj->foo
```

The default error message is just `"Parameter validation failed"`. You can pass an option hashref as the first argument with an informative message string:

```
sub foo {
   state $OptionsDict = Dict[...];
   state $check = multisig(
      { message => 'USAGE: $object->foo(\%options?, $string)' },
      [ Object, $OptionsDict, StringLike ],
      [ Object, StringLike ],
   );
   my ($self, @args) = $check->(@_);
   my ($opts, $str)  = ${^TYPE_PARAMS_MULTISIG} ? ({}, @args) : @_;
   ...;
}

$obj->foo(\%opts, "Hello");
$obj->foo("World");
```

`wrap_subs( $subname1, $wrapper1, ... )`

It's possible to turn the check inside-out and instead of the sub calling the check, the check can call the original sub.

Normal way:

```
use Type::Param qw(compile);
use Types::Standard qw(Int Str);

sub foobar {
   state $check = compile(Int, Str);
   my ($foo, $bar) = @_;
   ...;
}
```

Inside-out way:

```
use Type::Param qw(wrap_subs);
use Types::Standard qw(Int Str);

sub foobar {
   my ($foo, $bar) = @_;
   ...;
}

wrap_subs foobar => [Int, Str];
```

`wrap_subs` takes a hash of subs to wrap. The keys are the sub names and the values are either arrayrefs of arguments to pass to `compile` to make a check, or coderefs that have already been built by `compile`, `compile_named`, or `compile_named_oo`.

`wrap_methods( $subname1, $wrapper1, ... )`

`wrap_methods` also exists, which shifts off the invocant from `@_` before the check, but unshifts it before calling the original sub.

```
use Type::Param qw(wrap_subs);
use Types::Standard qw(Int Str);

sub foobar {
   my ($self, $foo, $bar) = @_;
   ...;
```

```
  }

  wrap_subs foobar => [Int, Str];
```

Invocant

Type::Params exports a type **Invocant** on request. This gives you a type constraint which accepts classnames *and* blessed objects.

```
use Type::Params qw( compile Invocant );

sub my_method {
  state $check = compile(Invocant, ArrayRef, Int);
  my ($self_or_class, $arr, $ix) = $check->(@_);

  return $arr->[ $ix ];
}
```

ENVIRONMENT

`PERL_TYPE_PARAMS_XS`

Affects the building of accessors for `compile_named_oo`. If set to true, will use Class::XSAccessor. If set to false, will use pure Perl. If this environment variable does not exist, will use Class::XSAccessor if it is available.

BUGS

Please report any bugs to <http://rt.cpan.org/Dist/Display.html?Queue=Type−Tiny>.

SEE ALSO

The Type::Tiny homepage <http://typetiny.toby.ink/>.

Type::Tiny, Type::Coercion, Types::Standard.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES