NAME

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

SYNOPSIS

```
use v5.10;
use strict;
use warnings;

use Type::Params qw( compile );
use Types::Standard qw( slurpy Str ArrayRef Num );

sub deposit_monies
{
   state $check = compile( Str, Str, slurpy ArrayRef[Num] );
   my ($sort_code, $account_number, $monies) = $check->(@_);

   my $account = Local::BankAccount->new($sort_code, $account_number);
   $account->deposit($_) for @$monies;
}

deposit_monies("12-34-56", "11223344", 1.2, 3, 99.99);
```

STATUS

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

DESCRIPTION

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 $deposit_monies_check;
sub deposit_monies
{
   $deposit_monies_check ||= compile( Str, Str, slurpy ArrayRef[Num] );
   my ($sort_code, $account_number, $monies) = $deposit_monies_check->(@_);

   ...;
}
```

Not quite as neat, but not awful either.

There's a shortcut reducing it to one step:

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

sub deposit_monies
{
   my ($sort_code, $account_number, $monies) =
      validate( \@_, Str, Str, slurpy ArrayRef[Num] );

   ...;
```

```
}
```

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.

VALIDATE VERSUS COMPILE

This module offers one-stage (''validate'') and two-stage (''compile'' then ''check'') variants of parameter checking for you to use. Performance with the two-stage variant will *always* beat the one stage variant — I cannot think of many reasons you'd want to use the one-stage version.

```
# One-stage, positional parameters
my @args = validate(\@_, @spec);

# Two-stage, positional parameters
state $check = compile(@spec);
my @args = $check->(@_);

# One-stage, named parameters
my $args = validate_named(\@_, @spec);

# Two-stage, named parameters
state $check = compile_named(@spec);
my $args = $check->(@_);
```

Use `compile` and `compile_named`, not `validate` and `validate_named`.

VALIDATION SPECIFICATIONS

The `@spec` is where most of the magic happens.

The generalized form of specifications for positional parameters is:

```
@spec = (
  \%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),
);
```

And for named parameters:

```
@spec = (
  \%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),
);
```

Option hashrefs can simply be omitted if you don't need to specify any particular options.

The `slurpy` function is exported by Types::Standard. It may be omitted if not needed.

General Options

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.

`class => ClassName`
: **Named parameters only.** The check coderef will, instead of returning a simple hashref, call `$class->new($hashref)` and return a proper object.

`constructor => Str`
: **Named parameters only.** Specify an alternative method name instead of `new` for the `class` option described above.

`class => Tuple[ClassName, Str]`
: **Named parameters only.** Given a class name and constructor name pair, the check coderef will, instead of returning a simple hashref, call `$class->$constructor($hashref)` and return a proper object. Shortcut for declaring both the `class` and `constructor` options at once.

`bless => ClassName`
: **Named parameters only.** Bypass the constructor entirely and directly bless the hashref.

`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.

Type Constraints

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.

Optional Parameters

The `Optional` parameterizable type constraint from Types::Standard may be used to indicate optional parameters.

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

# Named parameters
state $check = compile(
  foo => Int,
  bar => Optional[Int],
  baz => Optional[Int],
);
my $args = $check->(@_);  # $args->{bar} and $args->{baz} are optional
```

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

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

If you're using positional parameters, then required parameters must precede any optional ones.

Slurpy Parameters

Specifications may include a single slurpy parameter which should have a type constraint derived from `ArrayRef` or `HashRef`. (`Any` is also allowed, which is interpreted as `ArrayRef` in the case of positional parameters, and `HashRef` in the case of named parameters.)

If a slurpy parameter is provided in the specification, the `$check` coderef will slurp up any remaining arguments from `@_` (after required and optional parameters have been removed), validate it against the given slurpy type, and return it as a single arrayref/hashref.

For example:

```
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 ]
```

A specification have one or zero slurpy parameters. If there is a slurpy parameter, it must be the final one.

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.

Type Coercion

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

```
my $RoundedInt = Int->plus_coercions(Num, q{ int($_) });

state $check = compile($RoundedInt, $RoundedInt);
my ($foo, $bar) = $check->(@_);

# if @_ is (1.1, 2.2), then $foo is 1 and $bar is 2.
```

Coercions carry over into structured types such as `ArrayRef` automatically:

```
sub delete_articles
{
  state $check = compile( Object, slurpy ArrayRef[$RoundedInt] );
  my ($db, $articles) = $check->(@_);

  $db->select_article($_)->delete for @$articles;
}

# delete articles 1, 2 and 3
delete_articles($my_db, 1.1, 2.2, 3.3);
```

That's a Types::Standard feature rather than something specific to Type::Params.

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.

Parameter Options

The type constraint for a parameter may be followed by a hashref of options for it.

The following options are supported:

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

```
state $check = compile_named(
  foo => Int,
  bar => Int, { optional => 1 },
  baz => Optional[Int],
);
```

The two are not *exactly* equivalent. If you were to set `bar` to a non-integer, it would throw an exception about the `Int` type constraint being violated. If `baz` were a non-integer, the exception would mention the `Optional[Int]` type constraint instead.

`default => CodeRef|Ref|Str|Undef`

A default may be provided for a parameter.

```
state $check = compile_named(
  foo => Int,
  bar => Int, { default => "666" },
  baz => 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 = compile_named(
  foo => Int,
  bar => Int, { default => sub { 6 * 111 } },
  baz => 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.

Defaults are not supported for slurpy parameters.

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.

MULTIPLE SIGNATURES

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
```

PARAMETER OBJECTS

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!

Options

`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;
}
```

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

Classes

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.

COOKBOOK

Mixed Positional and Named Parameters

This can be faked using positional parameters and a slurpy dictionary.

```
state $check = compile(
  Int,
  slurpy Dict[
    foo => Int,
    bar => Optional[Int],
    baz => Optional[Int],
  ],
);

@_ = (42, foo => 21);                  # ok
@_ = (42, foo => 21, bar  => 84);      # ok
@_ = (42, foo => 21, bar  => 10.5);    # not ok
@_ = (42, foo => 21, quux => 84);      # not ok
```

Method Calls

Some people like to `shift` off the invocant before running type checks:

```
sub my_method {
  my $self = shift;
  state $check = compile_named(
    haystack => ArrayRef,
    needle   => Int,
  );
  my $arg = $check->(@_);

  return $arg->{haystack}[ $self->base_index + $arg->{needle} ];
}

$object->my_method(haystack => \@somelist, needle => 42);
```

If you're using positional parameters, there's really no harm in including the invocant in the check:

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

  return $arr->[ $self->base_index + $ix ];
}

$object->my_method(\@somelist, 42);
```

Some methods will be designed to be called as class methods rather than instance methods. Remember to use `ClassName` instead of `Object` in those cases.

Type::Params exports an additional keyword `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 ];
}
```

There is no `coerce => 0`

If you give `compile` a type constraint which has coercions, then `$check` will *always coerce*. It cannot be switched off.

Luckily, Type::Tiny gives you a very easy way to create a type constraint without coercions from one that has coercions:

```
state $check = compile(
  $RoundedInt->no_coercions,
  $RoundedInt->minus_coercions(Num),
);
```

That's a Type::Tiny feature rather than a Type::Params feature though.

Extra Coercions

Type::Tiny provides an easy shortcut for adding coercions to a type constraint:

```
# We want an arrayref, but accept a hashref and coerce it
state $check => compile(
  ArrayRef->plus_coercions( HashRef, sub { [sort values %$_] } ),
);
```

Value Constraints

You may further constrain a parameter using `where`:

```
state $check = compile(
  Int->where('$_ % 2 == 0'),   # even numbers only
);
```

This is also a Type::Tiny feature rather than a Type::Params feature.

Smarter Defaults

This works:

```
sub print_coloured {
  state $check = compile(
    Str,
    Str, { default => "black" },
  );

  my ($text, $colour) = $check->(@_);

  ...;
}
```

But so does this (and it might benchmark a little faster):

```
sub print_coloured {
  state $check = compile(
    Str,
    Str, { optional => 1 },
  );

  my ($text, $colour) = $check->(@_);
  $colour = "black" if @_ < 2;

  ...;
}
```

Just because Type::Params now supports defaults, doesn't mean you can't do it the old-fashioned way. The latter is more flexible. In the example, we've used `if @_ < 2`, but we could instead have done something like:

```
$colour ||= "black";
```

Which would have defaulted `$colour` to "black" if it were the empty string.

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.

COMPARISONS WITH OTHER MODULES

Params::Validate

Type::Params is not really a drop-in replacement for Params::Validate; the API differs far too much to claim that. Yet it performs a similar task, so it makes sense to compare them.

- Type::Params will tend to be faster if you've got a sub which is called repeatedly, but may be a little slower than Params::Validate for subs that are only called a few times. This is because it does a bunch of work the first time your sub is called to make subsequent calls a lot faster.
- Params::Validate doesn't appear to have a particularly natural way of validating a mix of positional and named parameters.
- Type::Utils allows you to coerce parameters. For example, if you expect a Path::Tiny object, you could coerce it from a string.
- If you are primarily writing object-oriented code, using Moose or similar, and you are using Type::Tiny type constraints for your attributes, then using Type::Params allows you to use the same constraints for method calls.
- Type::Params comes bundled with Types::Standard, which provides a much richer vocabulary of types than the type validation constants that come with Params::Validate. For example, Types::Standard provides constraints like `ArrayRef[Int]` (an arrayref of integers), while the closest from Params::Validate is `ARRAYREF`, which you'd need to supplement with additional callbacks if you wanted to check that the arrayref contained integers.

  Whatsmore, Type::Params doesn't just work with Types::Standard, but also any other Type::Tiny type constraints.

Params::ValidationCompiler

Params::ValidationCompiler does basically the same thing as Type::Params.

- Params::ValidationCompiler and Type::Params are likely to perform fairly similarly. In most cases, recent versions of Type::Params seem to be *slightly* faster, but except in very trivial cases, you're unlikely to notice the speed difference. Speed probably shouldn't be a factor when choosing between them.
- Type::Params's syntax is more compact:

```
state $check = compile(Object, Optional[Int], slurpy ArrayRef);
```

  Versus:

```
state $check = validation_for(
   params => [
      { type => Object },
      { type => Int,      optional => 1 },
      { type => ArrayRef, slurpy => 1 },
   ],
);
```

- Params::ValidationCompiler probably has slightly better exceptions.

BUGS

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

SEE ALSO

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

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES