NAME

Type::Tiny::Manual::Params – coerce and validate arguments to functions and methods

DESCRIPTION

There is a module called Type::Params available to wrap up type coercion and constraint checks into a single, simple and fast check. If you care about speed, and your sub signatures are fairly simple, then this is the way to go...

```
use feature qw( state );
use Types::Standard qw( Str );
use Type::Utils;
use Type::Params qw( compile );

my $Invocant = class_type { class => __PACKAGE__ };

sub set_name
{
   state $check = compile($Invocant, Str);
   my ($self, $name) = $check->(@_);

   ...;
}
```

See the COOKBOOK section of Type::Params for further information.

The Somewhat More Manual Way...

In general, Type::Params should be sufficient to cover most needs, and will probably run faster than almost anything you could cook up yourself. However, sometimes you need to deal with unusual function signatures that it does not support. For example, imagine function `format_string` takes an optional hashref of formatting instructions, followed by a required string. You might expect to be able to handle it like this:

```
sub format_string
{
   state $check = compile(Optional[HashRef], Str);
   my ($instructions, $string) = $check->(@_);

   ...;
}
```

However, this won't work, as Type::Params expects required parameters to always precede optional ones. So there are times you need to handle parameters more manually.

In these cases, bear in mind that for any type constraint object you have several useful checking methods available:

```
Str->check($var)            # returns a boolean
is_Str($var)                # ditto
Str->($var)                 # returns $var or dies
assert_Str($var)            # ditto
```

Here's how you might handle the `format_string` function:

```
sub format_string
{
   my $instructions;
   $instructions = shift if HashRef->check($_[0]);

   my $string = Str->(shift);
```

```
      ...;
   }
```

Alternatively, you could manipulate @_ before passing it to the compiled check:

```
   sub format_string
   {
      state $check = compile(HashRef, Str);
      my ($instructions, $str) = $check->(@_==1 ? ({}, @_) : @_);

      ...;
   }
```

Signatures

Don't you wish your subs could look like this?

```
   sub set_name (Object $self, Str $name)
   {
      $self->{name} = $name;
   }
```

Well; here are a few solutions for sub signatures that work with Type::Tiny...

Kavorka

Kavorka is a sub signatures implementation written to natively use Type::Utils' `dwim_type` for type constraints, and take advantage of Type::Tiny's features such as inlining, and coercions.

```
   method set_name (Str $name)
   {
      $self->{name} = $name;
   }
```

Kavorka's signatures provide a lot more flexibility, and slightly more speed than Type::Params. (The speed comes from inlining almost all type checks into the body of the sub being declared.)

Kavorka also includes support for type checking of the returned value.

Kavorka can also be used as part of Moops, a larger framework for object oriented programming in Perl.

Function::Parameters

The following should work with Function::Parameters 1.0201 or above:

```
   use Type::Utils;
   use Function::Parameters {
      method => {
         strict     => 1,
         reify_type => sub { Type::Utils::dwim_type($_[0]) },
      },
   };

   method set_name (Str $name)
   {
      $self->{name} = $name;
   }
```

Note that by default, Function::Parameters uses Moose's type constraints. The `reify_type` option above (introduced in Function::Parameters 1.0201) allows you to "divert" type constraint lookups. Using Type::Tiny constraints will gain you about a 7% speed-up in function signature checks.

An alternative way to use Function::Parameter with Type::Tiny is to provide type constraint expressions in parentheses:

```
use Types::Standard;
use Function::Parameters ':strict';

method set_name ((Str) $name)
{
   $self->{name} = $name;
}
```

Attribute::Contract

Both Kavorka and Function::Parameters require a relatively recent version of Perl. Attribute::Contract supports older versions by using a lot less magic.

You want Attribute::Contract 0.03 or above.

```
use Attribute::Contract -types => [qw/Object Str/];

sub set_name :ContractRequires(Object, Str)
{
   my ($self, $name) = @_;
   $self->{name} = $name;
}
```

Attribute::Contract also includes support for type checking of the returned value.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES