NAME

Type::Tiny::Manual::Optimization – squeeze the most out of your CPU

DESCRIPTION

Various tips to help you squeeze the most out of your CPU.

XS

The simplest thing you can do to increase performance of many of the built-in type constraints is to install Type::Tiny::XS, a set of ultra-fast type constraint checks implemented in C.

Type::Tiny will attempt to load Type::Tiny::XS and use its type checks. If Type::Tiny::XS is not available, it will then try to use Mouse **if it is already loaded**, but Type::Tiny won't attempt to load Mouse for you.

Types that can be accelerated by Type::Tiny::XS

The following simple type constraints from Types::Standard will be accelerated by Type::Tiny::XS: `Any`, `ArrayRef`, `Bool`, `ClassName`, `CodeRef`, `Defined`, `FileHandle`, `GlobRef`, `HashRef`, `Int`, `Item`, `Object`, `Map`, `Ref`, `ScalarRef`, `Str`, `Tuple`, `Undef`, and `Value`. (Note that `Num` and `RegexpRef` are **not** on that list.)

The parameterized form of `Ref` cannot be accelerated.

The parameterized forms of `ArrayRef`, `HashRef`, and `Map` can be accelerated only if their parameters are.

The parameterized form of `Tuple` can be accelerated if its parameters are, it has no `Optional` components, and it does not use `slurpy`.

Certain type constraints may benefit partially from Type::Tiny::XS. For example, `RoleName` inherits from `ClassName`, so part of the type check will be conducted by Type::Tiny::XS.

The parameterized `InstanceOf`, `HasMethods`, and `Enum` type constraints will be accelerated. So will Type::Tiny::Class, Type::Tiny::Duck, and Type::Tiny::Enum objects. (But enums will only be accelerated if the list of allowed string values consist entirely of word characters and hyphens – that is: `not grep /[^\w-]/, @values`.)

The `PositiveInt` and `PositiveOrZeroInt` type constraints from Types::Common::Numeric will be accelerated, as will the `NonEmptyStr` type constraint from Types::Common::String.

Type::Tiny::Union and Type::Tiny::Intersection will also be accelerated if their constituent type constraints are.

Types that can be accelerated by Mouse

The following simple type constraints from Types::Standard will be accelerated by Type::Tiny::XS: `Any`, `ArrayRef`, `Bool`, `ClassName`, `CodeRef`, `Defined`, `FileHandle`, `GlobRef`, `HashRef`, `Ref`, `ScalarRef`, `Str`, `Undef`, and `Value`. (Note that `Item`, `Num`, `Int`, `Object`, and `RegexpRef` are **not** on that list.)

The parameterized form of `Ref` cannot be accelerated.

The parameterized forms of `ArrayRef` and `HashRef` can be accelerated only if their parameters are.

Certain type constraints may benefit partially from Mouse. For example, `RoleName` inherits from `ClassName`, so part of the type check will be conducted by Mouse.

The parameterized `InstanceOf` and `HasMethods` type constraints will be accelerated. So will Type::Tiny::Class and Type::Tiny::Duck objects.

Common Sense

The `HashRef[ArrayRef]` type constraint can probably be checked faster than `HashRef[ArrayRef[Num]]`. If you find yourself using very complex and slow type constraints, you should consider switching to simpler and faster ones. (Though this means you have to place a little more trust in your caller to not supply you with bad data.)

(A counter-intuitive exception to this: even though `Int` is more restrictive than `Num`, in most circumstances

`Int` checks will run faster.)

Inlining Type Constraints

If your type constraint can be inlined, this can not only speed up Type::Tiny's own checks and coercions, it may also allow your type constraint to be inlined into generated methods such as Moose attribute accessors.

All of the constraints from `Types::Standard` can be inlined, as can enum, class_type, role_type and duck_type constraints. Union and intersection constraints can be inlined if their sub-constraints can be. So if you can define your own types purely in terms of these types, you automatically get inlining:

```
declare HashLike, as union [
   Ref["HASH"],
   Overload["&{}"],
];
```

However, sometimes these base types are not powerful enough and you'll need to write a constraint coderef:

```
declare NonEmptyHash, as HashLike,
   where     { scalar values %$_ };
```

... and you've suddenly sacrificed a lot of speed.

Inlining to the rescue! You can define an inlining coderef which will be passed two parameters: the constraint itself and a variable name as a string. For example, the variable name might be `'$_'` or `'$_[0]'`. Your coderef should return a Perl expression string, interpolating that variable name.

```
declare NonEmptyHash, as HashLike,
   where     { scalar values %$_ },
   inline_as {
      my ($constraint, $varname) = @_;
      return sprintf(
         '%s and scalar values %%{%s}',
         $constraint->parent->inline_check($varname),
         $varname,
      );
   };
```

The Perl expression could be inlined within a function or a `if` clause or potentially anywhere, so it really must be an expression, not a statement. It should not `return` or `exit` and probably shouldn't `die`. (If you need loops and so on, you can output a `do` block.)

Note that if you're subtyping an existing type constraint, your `inline_as` block is also responsible for checking the parent type's constraint. This can be done quite easily, as shown in the example above.

Note that defining a type constraint in terms of a constraint coderef and an inlining coderef can be a little repetitive. Sub::Quote provides an alternative that reduces repetition (though the inlined code might not be as compact/good/fast).

```
declare NonEmptyHash, as HashLike,
   constraint => quote_sub q{ scalar values %$_ };
```

Aside: it's been pointed out that ''might not be as fast'' above is a bit hand-wavy. When Type::Tiny does inlining from Sub::Quote coderefs, it needs to inline all the ancestor type constraints, and smush them together with `&&`. This may result in duplicate checks. For example, if 'MyArray' inherits from 'MyRef' which inherits from 'MyDef', the inlined code might end up as:

```
defined($_)              # check MyDef
&& ref($_)               # check MyRef
&& ref($_) eq 'ARRAY'    # check MyArray
```

When just the last check would have been sufficient. A custom `inline_as` allows you finer control over how the type constraint is inlined.

Optimizing Coercions

Coercions are often defined using coderefs:

```
PathTiny->plus_coercions(
   Str,   sub { "Path::Tiny"->new($_) },
   Undef, sub { "Path::Tiny"->new("/etc/myapp/default.conf") },
);
```

But you can instead define them as strings of Perl code operating on `$_`:

```
PathTiny->plus_coercions(
   Str,   q{ "Path::Tiny"->new($_) },
   Undef, q{ "Path::Tiny"->new("/etc/myapp/default.conf") },
);
```

The latter will run faster, so is preferable at least for simple coercions.

This makes the most difference when used with Moo, which supports inlining of coercions. Moose does not inline coercions, but providing coercions as strings still allows Type::Tiny to optimize the coercion coderef it provides to Moose.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES