NAME

Ref::Util – Utility functions for checking references

VERSION

version 0.204

SYNOPSIS

```
use Ref::Util qw( is_plain_arrayref is_plain_hashref );

if ( is_plain_arrayref( $something ) ) {
    print for @{ $something };
} elsif ( is_plain_hashref( $something ) ) {
    print for sort values %{ $something };
}
```

DESCRIPTION

Ref::Util introduces several functions to help identify references in a **smarter** (and usually faster) way. In short:

```
# conventional approach             # with Ref::Util

ref( $foo ) eq 'ARRAY'              is_plain_arrayref( $foo )

use Scalar::Util qw( reftype );
reftype( $foo ) eq 'ARRAY'          is_arrayref( $foo )
```

The difference:

- No comparison against a string constant

  When you call `ref`, you stringify the reference and then compare it to some string constant (like `ARRAY` or `HASH`). Not just awkward, it's brittle since you can mispell the string.

  If you use Scalar::Util's `reftype`, you still compare it as a string:

  ```
  if ( reftype($foo) eq 'ARRAY' ) { ... }
  ```

- Supports blessed variables

  Note: In future versions, the idea is to make the default functions use the **plain** variation, which means explicitly non-blessed references.

  If you want to explicitly check for **blessed** references, you should use the `is_blessed_*` functions. There will be an `is_any_*` variation which will act like the current main functions – not caring whether it's blessed or not.

  When calling `ref`, you receive either the reference type (**SCALAR**, **ARRAY**, **HASH**, etc.) or the package it's blessed into.

  When calling `is_arrayref` (et. al.), you check the variable flags, so even if it's blessed, you know what type of variable is blessed.

  ```
  my $foo = bless {}, 'PKG';
  ref($foo) eq 'HASH'; # fails

  use Ref::Util 'is_hashref';
  my $foo = bless {}, 'PKG';
  is_hashref($foo); # works
  ```

  On the other hand, in some situations it might be better to specifically exclude blessed references. The rationale for that might be that merely because some object happens to be implemented using a hash doesn't mean it's necessarily correct to treat it as a hash. For these situations, you can use `is_plain_hashref` and friends, which have the same performance benefits as `is_hashref`.

There is also a family of functions with names like `is_blessed_hashref`; these return true for blessed object instances that are implemented using the relevant underlying type.

- Supports tied variables and magic

  Tied variables (used in Readonly, for example) are supported.

  ```
  use Ref::Util qw<is_plain_hashref>;
  use Readonly;

  Readonly::Scalar my $rh2 => { a => { b => 2 } };
  is_plain_hashref($rh2); # success
  ```

  Ref::Util added support for this in 0.100. Prior to this version the test would fail.

- Ignores overloading

  These functions ignore overloaded operators and simply check the variable type. Overloading will likely not ever be supported, since I deem it problematic and confusing.

  Overloading makes your variables opaque containers and hides away **what** they are and instead require you to figure out **how** to use them. This leads to code that has to test different abilities (in `eval`, so it doesn't crash) and to interfaces that get around what a person thought you would do with a variable. This would have been alright, except there is no clear way of introspecting it.

- Ignores subtle types:

  The following types, provided by Scalar::Util's `reftype`, are not supported:

  - `VSTRING`

    This is a `PVMG` ("normal" variable) with a flag set for VSTRINGs. Since this is not a reference, it is not supported.

  - `LVALUE`

    A variable that delegates to another scalar. Since this is not a reference, it is not supported.

  - `INVLIST`

    I couldn't find documentation for this type.

  Support might be added, if a good reason arises.

- Usually fast

  When possible, Ref::Util uses Ref::Util::XS as its implementation. (If you don't have a C compiler available, it uses a pure Perl fallback that has all the other advantages of Ref::Util, but isn't as fast.)

  In fact, Ref::Util::XS has two alternative implementations available internally, depending on the features supported by the version of Perl you're using. For Perls that supports custom OPs, we actually add an OP (which is faster); for other Perls, the implementation that simply calls an XS function (which is still faster than the pure-Perl equivalent).

  See below for benchmark results.

EXPORT

Nothing is exported by default. You can ask for specific subroutines (described below) or ask for all subroutines at once:

```
use Ref::Util qw<is_scalarref is_arrayref is_hashref ...>;

# or

use Ref::Util ':all';
```

SUBROUTINES

is_ref($ref)

Check for a reference to anything.

```
is_ref([]);
```

is_scalarref($ref)

Check for a scalar reference.

```
is_scalarref(\"hello");
is_scalarref(\30);
is_scalarref(\$value);
```

Note that, even though a reference is itself a type of scalar value, a reference to another reference is not treated as a scalar reference:

```
!is_scalarref(\\1);
```

The rationale for this is two-fold. First, callers that want to decide how to handle inputs based on their reference type will usually want to treat a ref-ref and a scalar-ref differently. Secondly, this more closely matches the behavior of the `ref` built-in and of "reftype" in Scalar::Util, which report a ref-ref as `REF` rather than `SCALAR`.

is_arrayref($ref)

Check for an array reference.

```
is_arrayref([]);
```

is_hashref($ref)

Check for a hash reference.

```
is_hashref({});
```

is_coderef($ref)

Check for a code reference.

```
is_coderef( sub {} );
```

is_regexpref($ref)

Check for a regular expression (regex, regexp) reference.

```
is_regexpref( qr// );
```

is_globref($ref)

Check for a glob reference.

```
is_globref( \*STDIN );
```

is_formatref($ref)

Check for a format reference.

```
# set up format in STDOUT
format STDOUT =
.

# now we can test it
is_formatref( *main::STDOUT{'FORMAT'} );
```

This function is not available in Perl 5.6 and will trigger a `croak()`.

is_ioref($ref)

Check for an IO reference.

```
is_ioref( *STDOUT{IO} );
```

is_refref($ref)

Check for a reference to a reference.

```
    is_refref( \[] ); # reference to array reference
```

is_plain_scalarref($ref)
Check for an unblessed scalar reference.

```
    is_plain_scalarref(\"hello");
    is_plain_scalarref(\30);
    is_plain_scalarref(\$value);
```

is_plain_ref($ref)
Check for an unblessed reference to anything.

```
    is_plain_ref([]);
```

is_plain_arrayref($ref)
Check for an unblessed array reference.

```
    is_plain_arrayref([]);
```

is_plain_hashref($ref)
Check for an unblessed hash reference.

```
    is_plain_hashref({});
```

is_plain_coderef($ref)
Check for an unblessed code reference.

```
    is_plain_coderef( sub {} );
```

is_plain_globref($ref)
Check for an unblessed glob reference.

```
    is_plain_globref( \*STDIN );
```

is_plain_formatref($ref)
Check for an unblessed format reference.

```
    # set up format in STDOUT
    format STDOUT =
    .

    # now we can test it
    is_plain_formatref(bless *main::STDOUT{'FORMAT'} );
```

is_plain_refref($ref)
Check for an unblessed reference to a reference.

```
    is_plain_refref( \[] ); # reference to array reference
```

is_blessed_scalarref($ref)
Check for a blessed scalar reference.

```
    is_blessed_scalarref(bless \$value);
```

is_blessed_ref($ref)
Check for a blessed reference to anything.

```
    is_blessed_ref(bless [], $class);
```

is_blessed_arrayref($ref)
Check for a blessed array reference.

```
    is_blessed_arrayref(bless [], $class);
```

is_blessed_hashref($ref)
Check for a blessed hash reference.

```
    is_blessed_hashref(bless {}, $class);
```

is_blessed_coderef($ref)
Check for a blessed code reference.

```
is_blessed_coderef( bless sub {}, $class );
```

is_blessed_globref($ref)
Check for a blessed glob reference.

```
is_blessed_globref( bless \*STDIN, $class );
```

is_blessed_formatref($ref)
Check for a blessed format reference.

```
# set up format for FH
format FH =
.

# now we can test it
is_blessed_formatref(bless *FH{'FORMAT'}, $class );
```

is_blessed_refref($ref)
Check for a blessed reference to a reference.

```
is_blessed_refref( bless \[], $class ); # reference to array reference
```

BENCHMARKS

Here is a benchmark comparing similar checks.

```
my $bench = Dumbbench->new(
    target_rel_precision => 0.005,
    initial_runs         => 20,
);

my $amount = 1e7;
my $ref    = [];
$bench->add_instances(
    Dumbbench::Instance::PerlSub->new(
        name => 'Ref::Util::is_plain_arrayref (CustomOP)',
        code => sub {
            Ref::Util::is_plain_arrayref($ref) for ( 1 .. $amount )
        },
    ),

    Dumbbench::Instance::PerlSub->new(
        name => 'ref(), reftype(), !blessed()',
        code => sub {
            ref $ref
                && Scalar::Util::reftype($ref) eq 'ARRAY'
                && !Scalar::Util::blessed($ref)
                for ( 1 .. $amount );
        },
    ),

    Dumbbench::Instance::PerlSub->new(
        name => 'ref()',
        code => sub { ref($ref) eq 'ARRAY' for ( 1 .. $amount ) },
    ),

    Dumbbench::Instance::PerlSub->new(
        name => 'Data::Util::is_array_ref',
```

```
            code => sub { is_array_ref($ref) for ( 1 .. $amount ) },
        ),

    );
```

The results:

```
    ref():                                    5.335e+00 +/- 1.8e-02 (0.3%)
    ref(), reftype(), !blessed():             1.5545e+01 +/- 3.1e-02 (0.2%)
    Ref::Util::is_plain_arrayref (CustomOP): 2.7951e+00 +/- 6.2e-03 (0.2%)
    Data::Util::is_array_ref:                 5.9074e+00 +/- 7.5e-03 (0.1%)
```

(Rounded run time per iteration)

A benchmark against Data::Util:

```
    Ref::Util::is_plain_arrayref: 3.47157e-01 +/- 6.8e-05 (0.0%)
    Data::Util::is_array_ref:     6.7562e-01 +/- 7.5e-04 (0.1%)
```

SEE ALSO

- Params::Classify
- Scalar::Util
- Data::Util

THANKS

The following people have been invaluable in their feedback and support.

- Yves Orton
- Steffen Müller
- Jarkko Hietaniemi
- Mattia Barbon
- Zefram
- Tony Cook
- Sergey Aleynikov

AUTHORS

- Aaron Crane
- Vikentiy Fesunov
- Sawyer X
- Gonzalo Diethelm
- p5pclub

LICENSE

AUTHORS

- Sawyer X <xsawyerx@cpan.org>
- Aaron Crane <arc@cpan.org>
- Vikenty Fesunov <vyf@cpan.org>
- Gonzalo Diethelm <gonzus@cpan.org>
- Karen Etheridge <ether@cpan.org>

COPYRIGHT AND LICENSE

```
The MIT (X11) License
```