Writing Self-Modifying Perl

Spencer Tipping

December 25, 2010

Contents

1 Introduction **2**

2 A Big Quine **3**
2.1 A basic quine 3
2.2 Reducing duplication 4
2.3 Using `eval` 4

3 Building the Shell **6**
3.1 Using an editor 7

4 Namespaces **10**
4.1 Handling functions more usefully 10
4.2 Catching attribute creation 11
4.3 Putting it all together 13
4.4 Separating bootstrap code 15

5 Serialization **19**
5.1 Separating the serialization logic 20

Chapter 1

Introduction

I've gotten a lot of WTF's[1] about self-modifying Perl scripts. Rightfully so, too. There's no documentation (until now), the interface is opaque and not particularly portable, and they aren't even very human-readable when edited:

```
...
meta::define_form 'meta', sub {
  my ($name, $value) = @_;
  meta::eval_in($value, "meta::$name");
};
meta::meta('configure', <<'__25976e07665878d3fae18f050160343f');
# A function to configure transients. Transients can be used to store any number of
# different things, but one of the more common usages is type descriptors.
sub meta::configure {
  my ($datatype, %options) = @_;
  $transient{$_}{$datatype} = $options{$_} for keys %options;
}
__25976e07665878d3fae18f050160343f
...
```

Despite these shortcomings, though, I think they're fairly useful. So rather than vindicate the idea (which is probably irredeemable), I've written this guide to dive into the mayhem and go from zero to a self-modifying Perl script. At the end, you'll have a script that is functionally equivalent to the `object` script, which I use as the prototype for all of the other ones.[2]

This guide probably isn't for the faint of heart, but if you're not afraid of `eval` then you might like it. Proceed only with fortitude, determination, and Perl v5.10.

[1] http://www.osnews.com/story/19266/WTFs_m

[2] See http://github.com/spencertipping/perl-objects for the full source.

Chapter 2

A Big Quine

At the core of things, a self-modifying Perl script is just a big quine.[1] There are only two real differences:

1. Self-modifying Perl scripts print into their own files rather than to standard output.
2. They print modified versions of themselves, not the original source.

If we're going to write such a script, it's good to start with a simple quine.

2.1 A basic quine

Some languages make quine-writing easier than others. Perl actually makes it very simple. Here's one:

Listing 2.1 `examples/basic-quine`

```
1 my $code = <<'EOF';
2 print 'my $code = <<\'EOF\';', "\n", $code, "EOF\n"; print $code;
3 EOF
4 print 'my $code = <<\'EOF\';', "\n", $code, "EOF\n"; print $code;
```

The logic is fairly straightforward, though it may not look like it. We're quoting a bunch of stuff using `<<'EOF'`,[2] and storing that into a string. We then put the quoted content outside of the heredoc to let it execute. The duplication is necessary; we want to quote the content and then run it.[3] The key is this line:

```
print 'my $code = <<\'EOF\';', "\n", $code, "EOF\n"; print $code;
```

This code prints the setup to define a new variable `$code` and prints its existing content after that.

[1] A "quine" being a program that prints its own source.

[2] The single-quoted heredoc form doesn't do any interpolation inside the document, which is ideal since we don't want to worry about escaping stuff.

[3] Later on I'll use `eval` to reduce the amount of duplication.

2.2 Reducing duplication

We don't want to write everything in our quine twice. Rather, we want to store most stuff just once and have a quine that scales well. The easiest way to do this is to use a hash to store the state, and serialize each key of the hash in the self-printing code. So instead of creating `$code`, we'll create `%data`:

Listing 2.2 `examples/data-quine`

```
my %data;
$data{code} = <<'EOF';
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{code};
EOF
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{code};
```

This is a good start. Here's how to add attributes without duplication:

Listing 2.3 `examples/data-quine-with-field`

```
my %data;
$data{foo} = <<'EOF';
a string
EOF
$data{code} = <<'EOF';
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{code};
EOF
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{code};
```

2.3 Using `eval`

The business about duplicating `$data{code}` is easily remedied by just `eval`ing `$data{code}` at the end. This requires the `eval` section to be duplicated, but it's smaller than `$data{code}`. Here's the quine with that transformation:[4]

Listing 2.4 `examples/data-quine-with-eval`

```
my %data;
$data{foo} = <<'EOF';
```

[4] Note that these quines might not actually print themselves identically due to hash-key ordering. This is fine; all of the keys are printed before we use them.

```
a string
EOF
$data{code} = <<'EOF';
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{bootstrap};
EOF
$data{bootstrap} = <<'EOF';
eval $data{code};
EOF
eval $data{code};
```

The advantage to this approach is that all we'll ever have to duplicate is `eval $data{code}` and `my %data;`, which is fairly trivial. It's important that you understand what's going on here, since this idea is integral to everything going forward.

Chapter 3

Building the Shell

Now that we've got attribute storage working, let's build a shell so that we don't have to edit these files by hand anymore. There are a couple of things that need to happen. First, we need to get these scripts to overwrite themselves instead of printing to standard output. Second, we need a way to get and set entries in `%data`. Starting with the quine from the last section, here's one way to go about it:

Listing 3.1 examples/basic-shell

```
my %data;
$data{cat} = <<'EOF';
sub cat {
  print join "\n", @data{@_};
}
EOF
$data{set} = <<'EOF';
sub set {
  $data{$_[0]} = join '', <STDIN>;
}
EOF
$data{code} = <<'EOF';
# Eval functions into existence:
eval $data{cat};
eval $data{set};

# Run specified command:
my $command = shift @ARGV;
&$command(@ARGV);

# Save new state:
open my $fh, '>', $0;
print $fh 'my %data;', "\n";
```

```
print $fh '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $fh $data{bootstrap};
close $fh;
EOF
$data{bootstrap} = <<'EOF';
eval $data{code};
EOF
eval $data{code};
```

Now we can modify its state:

```
$ perl examples/basic-shell cat cat
sub cat {
  print join "\n", @data{@_};
}
$ perl examples/basic-shell set foo
bar
^D
$ perl examples/basic-shell cat foo
bar
$
```

Not bad for a first implementation. This is a very minimal self-modifying Perl file, though it's useless at this point. It also has some fairly serious deficiencies (other than being useless). I'll cover the serious problems later on, but first let's address the usability.

3.1 Using an editor

The first thing that would help this script be more useful is a function that let you edit things with a real text editor. Fortunately this isn't difficult:

```
$ cp examples/basic-shell temp
$ perl temp set edit
sub edit {
  my $filename = '/tmp/' . rand();
  open my $file, '>', $filename;
  print $file $data{$_[0]};
  close $file;

  system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);

  open my $file, '<', $filename;
  $data{$_[0]} = join '', <$file>;
  close $file;
}
```

```
^D
$
```

It won't work yet though. The reason is that we aren't `eval`ing `edit` yet; we need to manually edit the `code` section and insert this line:

```
...
eval $data{cat};
eval $data{set};
eval $data{edit};         # <- insert this
...
```

Now you can invoke a text editor on any defined attribute:[1]

```
$ perl examples/editor-shell edit cat
# hack away
$
```

Here's the object at this point:

Listing 3.2 `examples/editor-shell`

```
my %data;
$data{cat} = <<'EOF';
sub cat {
  print join "\n", @data{@_};
}
EOF
$data{set} = <<'EOF';
sub set {
  $data{$_[0]} = join '', <STDIN>;
}
EOF
$data{edit} = <<'EOF';
sub edit {
  my $filename = '/tmp/' . rand();
  open my $file, '>', $filename;
  print $file $data{$_[0]};
  close $file;

  system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);

  open my $file, '<', $filename;
  $data{$_[0]} = join '', <$file>;
  close $file;
}
```

[1] Don't modify `bootstrap` or break the print code though! This will possibly nuke your object.

```
EOF
$data{code} = <<'EOF';
# Eval functions into existence:
eval $data{cat};
eval $data{set};
eval $data{edit};

# Run specified command:
my $command = shift @ARGV;
&$command(@ARGV);

# Save new state:
open my $fh, '>', $0;
print $fh 'my %data;', "\n";
print $fh '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $fh $data{bootstrap};
close $fh;
EOF
$data{bootstrap} = <<'EOF';
eval $data{code};
EOF
eval $data{code};
```

Chapter 4

Namespaces

It's a bummer to have to add a new `eval` line for every function we want to define. We could merge all of the functions into a single hash key, but that's too easy.[1] More appropriate is to assign a type to each hash key. This can be encoded in the name. For example, we might convert the names like this:

```
set  -> function::set
cat  -> function::cat
edit -> function::edit
code -> code::code
```

For reasons that I'll explain in a moment, we no longer need `bootstrap`. The rules governing these types are:

1. When we see a new `function::` key, evaluate its contents.
2. When we see a new `code::` key, evaluate its contents.

Rule 2 is why we don't need `bootstrap` anymore. Now you've probably noticed that these rules do exactly the same thing – why are we differentiating between these types then? Two reasons. First, we need to make sure that functions are evaluated before the code section is evaluated (otherwise the functions won't exist when we need them). Second, it's because functions can be handled in a more useful way.

4.1 Handling functions more usefully

Remember how we had to write `sub X {` and `}` every time we wrote a function, despite the fact that the function name was identical to the name of the key in `%data`? That's fairly lame, and it could become misleading if the names ever weren't the same. We really should have the script handle this for us. So instead of writing the function signature, we would just write its body:

[1] Aside from being a lame cop-out, it also limits extensibility, as I'll explain later.

```
# The body of 'cat':
print join "\n", @data{@_};
```

and infer its name from the key. Perl is helpful here by giving us first-class access to the symbol table:

```
sub create_function {
  my ($name, $body) = @_;
  *{$name} = eval "sub {\n$body\n}";
}
```

If we're going to handle functions this way, we need to change the rule for `function::` keys:

> When we see a new `function::` key, call `create_function` on the key name (without the `function::` part) and the value.

4.2 Catching attribute creation

We can't observe when a new key is added to `%data` as things are now. Fortunately this is easy to fix. Instead of writing lines that read `$data{...} = ...`, we can write some functions that perform this assignment for us, and in the process we can handle any side-effects like function creation. Here's a naive implementation:

```
sub define_function {
  my ($name, $value) = @_;
  $data{$name} = $value;
  create_function $name, $value;
}
sub define_code {
  my ($name, $value) = @_;
  $data{$name} = $value;
}
```

Since we're always going to assign into `%data`, we can abstract that step out:

```
sub define_definer {
  my ($name, $handler) = @_;
  *{$name} = sub {
    my ($name, $value) = @_;
    $data{$name} = $value;
    &$handler($name, $value);
  }
}
define_definer 'define_function', \&create_function;
```

```
define_definer 'define_code', sub {
  my ($name, $value) = @_;
  eval $value;
};
```

To avoid the possibility of later collisions we should probably use a separate namespace for all of these functions, since really bad things happen if you inadvertently replace one. I use the `meta::` namespace for this purpose in my scripts.

At this point we've got the foundation for namespace creation. This is actually used with few modifications in the Perl objects I use on a regular basis. Here's `meta::define_form` lifted from `object`:

```
sub meta::define_form {
  my ($namespace, $delegate) = @_;
  $datatypes{$namespace} = $delegate;
  *{"meta::${namespace}::implementation"} = $delegate;
  *{"meta::$namespace"} = sub {
    my ($name, $value) = @_;
    chomp $value;
    $data{"${namespace}::$name"} = $value;
    $delegate->($name, $value);
  };
}
```

The idea is the same as `define_definer`, but with a few extra lines. We stash the delegate in a `%datatypes` table for later reference. We also (redundantly, I notice) create a function in the `meta::` package so that we can refer to it when defining other forms. This lets us copy the behavior of namespaces but still have them be separate. The third line that's different is `chomp $value`, which is used because heredocs put an extra newline on the end of strings. `meta::define_form` has the same interface as `define_definer`:

```
meta::define_form 'function', \&create_function;
meta::define_form 'code', sub {
  my ($name, $value) = @_;
  eval $value;
};
```

Attribute definitions look a little different than they did before. The two `define_form` calls above create the functions `meta::function` and `meta::code`, which will need to be called this way:

```
meta::function('cat', <<'EOF');
print join "\n", @data{@_};
EOF
meta::code('main', <<'EOF');
```

```
# No more eval statements!
# Run command
...
# Save stuff
...
EOF
```

Notice that we don't specify the full name of the attributes being created. `meta::function('x', ...)` creates a key called `function::x`; this was handled in the `define_form` logic.

4.3 Putting it all together

At this point we're all set to write another script. The overall structure is still basically the same even though each piece has changed a little:

Listing 4.1 examples/basic-meta

```
my %data;
my %datatypes;

sub meta::define_form {
  my ($namespace, $delegate) = @_;
  $datatypes{$namespace} = $delegate;
  *{"meta::${namespace}::implementation"} = $delegate;
  *{"meta::$namespace"} = sub {
    my ($name, $value) = @_;
    chomp $value;
    $data{"${namespace}::$name"} = $value;
    $delegate->($name, $value);
  };
}
meta::define_form 'function', sub {
  my ($name, $body) = @_;
  *{$name} = eval "sub {\n$body\n}";
};
meta::define_form 'code', sub {
  my ($name, $value) = @_;
  eval $value;
};

meta::function('cat', <<'EOF');
print join "\n", @data{@_};
EOF

meta::code('main', <<'EOF');
```

```
# Run specified command:
my $command = shift @ARGV;
&$command(@ARGV);

# Save new state:
open my $file, '>', $0;

# Copy above bootstrapping logic:
print $file <<'EOF2';
my %data;
my %datatypes;

sub meta::define_form {
  my ($namespace, $delegate) = @_;
  $datatypes{$namespace} = $delegate;
  *{"meta::${namespace}::implementation"} = $delegate;
  *{"meta::$namespace"} = sub {
    my ($name, $value) = @_;
    chomp $value;
    $data{"${namespace}::$name"} = $value;
    $delegate->($name, $value);
  };
}
meta::define_form 'function', sub {
  my ($name, $body) = @_;
  *{$name} = eval "sub {\n$body\n}";
};
meta::define_form 'code', sub {
  my ($name, $value) = @_;
  eval $value;
};
EOF2

# Serialize attributes (everything else before code):
for (grep(!/^code::/, keys %data), grep(/^code::/, keys %data)) {
  my ($namespace, $name) = split /::/, $_, 2;
  print $file "meta::$namespace('$name', <<'EOF');\n$data{$_}\nEOF\n";
}

# Just for good measure:
print $file "\n__END__";
close $file;
EOF

__END__
```

The most substantial changes were:

1. We're defining two hashes at the beginning, though we still just use `%data`.
2. We're using delegate functions to define attributes rather than assigning directly into `%data`.
3. Quoted values now get `chomp`ed. I've added another `\n` in the serialization logic to compensate for this.
4. The serialization logic is now order-specific; it puts `code::` entries after other things.
5. The file now has an `__END__` marker on it.

4.4 Separating bootstrap code

The bootstrap code is now large quoted string inside `code::main`, which isn't optimal. Better is to break it out into its own attribute. To do this, we'll need a new namespace that has no side-effect.[2] I'll call this namespace `bootstrap::`.

```
meta::define_form 'bootstrap', sub {};
```

There's a special member of the `bootstrap::` namespace that contains the code in the beginning of the file:

```
meta::bootstrap('initialization', <<'EOF');
my %data;
my %datatypes;
...
EOF
```

This condenses `code::main` by a lot:

```
meta::code('main', <<'EOF');
# Run specified command:
my $command = shift @ARGV;
&$command(@ARGV);

# Save new state:
open my $file, '>', $0;
print $file $data{'bootstrap::initialization'};

# Serialize attributes (everything else before code):
for (grep(!/^code::/, keys %data), grep(/^code::/, keys %data)) {
```

[2] We can't use `code::` because then the code would be evaluated twice; once because it's printed directly, and again because of the `eval` in the `code::` delegate.

```
  my ($namespace, $name) = split /::/, $_, 2;
  print $file "meta::$namespace('$name', <<'EOF');\n$data{$_}\nEOF\n";
}

# Just for good measure:
print $file "\n__END__";
close $file;
EOF
```

Here's the final product, after adding the `set` and `edit` functions from before:

Listing 4.2 examples/basic-meta-with-functions

```
my %data;
my %datatypes;

sub meta::define_form {
  my ($namespace, $delegate) = @_;
  $datatypes{$namespace} = $delegate;
  *{"meta::${namespace}::implementation"} = $delegate;
  *{"meta::$namespace"} = sub {
    my ($name, $value) = @_;
    chomp $value;
    $data{"${namespace}::$name"} = $value;
    $delegate->($name, $value);
  };
}
meta::define_form 'bootstrap', sub {};
meta::define_form 'function', sub {
  my ($name, $body) = @_;
  *{$name} = eval "sub {\n$body\n}";
};
meta::define_form 'code', sub {
  my ($name, $value) = @_;
  eval $value;
};

meta::bootstrap('initialization', <<'EOF');
my %data;
my %datatypes;

sub meta::define_form {
  my ($namespace, $delegate) = @_;
  $datatypes{$namespace} = $delegate;
  *{"meta::${namespace}::implementation"} = $delegate;
  *{"meta::$namespace"} = sub {
```

```
    my ($name, $value) = @_;
    chomp $value;
    $data{"${namespace}::$name"} = $value;
    $delegate->($name, $value);
  };
}
meta::define_form 'bootstrap', sub {};
meta::define_form 'function', sub {
  my ($name, $body) = @_;
  *{$name} = eval "sub {\n$body\n}";
};
meta::define_form 'code', sub {
  my ($name, $value) = @_;
  eval $value;
};
EOF

meta::function('cat', <<'EOF');
print join "\n", @data{@_};
EOF

meta::function('set', <<'EOF');
$data{$_[0]} = join '', <STDIN>;
EOF

meta::function('edit', <<'EOF');
my $filename = '/tmp/' . rand();
open my $file, '>', $filename;
print $file $data{$_[0]};
close $file;

system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);

open my $file, '<', $filename;
$data{$_[0]} = join '', <$file>;
close $file;
EOF

meta::code('main', <<'EOF');
# Run specified command:
my $command = shift @ARGV;
&$command(@ARGV);

# Save new state:
open my $file, '>', $0;
print $file $data{'bootstrap::initialization'};
```

```
80
81  # Serialize attributes (everything else before code):
82  for (grep(!/^code::/, keys %data), grep(/^code::/, keys %data)) {
83    my ($namespace, $name) = split /::/, $_, 2;
84    print $file "meta::$namespace('$name', <<'EOF');\n$data{$_}\nEOF\n";
85  }
86
87  # Just for good measure:
88  print $file "\n__END__";
89  close $file;
90  EOF
91
92  __END__
```

Chapter 5

Serialization

Earlier I alluded to a glaring problem with these scripts as they stand. The issue is the `EOF` marker we've been using. Here's what happens if we put a line containing `EOF` into an attribute:

```
$ cp examples/basic-meta-with-functions temp
$ perl temp set function::bif
print <<'EOF';
uh-oh...
EOF
^D
$ perl temp cat function::bif
Can't locate object method "EOF" via package "meta::function" at temp line 31.
$
```

It's not hard to see what went wrong: `temp` now has an attribute definition that looks like this:

```
meta::function('bif', <<'EOF');
print <<'EOF';
uh-oh...
EOF

EOF
```

We need to come up with some end marker that isn't in the value being stored. For the moment let's use random numbers.[1]

[1] `object` implements a simple FNV-hash and uses the hash of the contents. I'll go over how to implement this a bit later.

5.1 Separating the serialization logic

There isn't a particularly compelling reason to inline the serialization logic in `code::main`. Since we have a low-overhead way of defining functions, let's make a `serialize` function to return the state of a script as a string, along with a helper method `serialize_single` to handle just one attribute:

```
meta::function('serialize', <<'EOF');
join "\n", map(serialize_single($_), grep !/^code::/, keys %data),
           map(serialize_single($_), grep  /^code::/, keys %data);
EOF

meta::function('serialize_single', <<'EOF');
my ($namespace, $name) = split /::/, $_[0], 2;
my $marker = '__' . int(rand(1 << 31));
"meta::$namespace('$name', <<'$marker');\n$data{$_[0]}\n$marker";
EOF
```