Writing Self-Modifying Perl

Spencer Tipping

February 18, 2012

Contents

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 (this guide is a self-modifying Perl file, in fact). 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] The full source code for this guide and accompanying examples is available at http://github.com/spencertipping/writing-self-modifying-perl.

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.

Part I

The Basics

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/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/quine-with-data`

```
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/quine-with-data-and-foo`

```
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/quine-with-data-and-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 of 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.[5]

[5] Alternatively, it probably also works to accept the code so far as magic and take my word for it that future code snippets do what they should. But it's probably less fun without the "aha!" moment.

Chapter 3

Building the interface

Now that we've got attribute storage working, let's build a command-line interface 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/cli-basic

```
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";
```

```
24 print $fh '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
25 print $fh $data{bootstrap};
26 close $fh;
27 EOF
28 $data{bootstrap} = <<'EOF';
29 eval $data{code};
30 EOF
31 eval $data{code};
```

Now we can modify its state:

```
$ perl examples/cli-basic cat cat
sub cat {
  print join "\n", @data{@_};
}
$ perl examples/cli-basic set foo
bar
^D
$ perl examples/cli-basic 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/cli-basic 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/cli-editor edit cat
# hack away
$
```

Here's the object at this point:

Listing 3.2 `examples/cli-editor`

```
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::main
```

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:

Listing 4.1 snippets/create-function

```
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:

Listing 4.2 snippets/define-function-define-code

```
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:

Listing 4.3 snippets/define-definer

```
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`:

Listing 4.4 snippets/meta-define-form

```
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`:

Listing 4.5 snippets/meta-define-form-function-code

```
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 4.6 `examples/namespace-basic`

```
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
```

```
72
73 __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:

Listing 4.7 snippets/bootstrapped-code-main

```
1 meta::code('main', <<'EOF');
2 # Run specified command:
3 my $command = shift @ARGV;
4 &$command(@ARGV);
5
6 # Save new state:
7 open my $file, '>', $0;
8 print $file $data{'bootstrap::initialization'};
```

[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.

```
9
10 # Serialize attributes (everything else before code):
11 for (grep(!/^code::/, keys %data), grep(/^code::/, keys %data)) {
12   my ($namespace, $name) = split /::/, $_, 2;
13   print $file "meta::$namespace('$name', <<'EOF');\n$data{$_}\nEOF\n";
14 }
15
16 # Just for good measure:
17 print $file "\n__END__";
18 close $file;
19 EOF
```

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

Listing 4.8 examples/namespace-full

```
1  my %data;
2  my %datatypes;
3
4  sub meta::define_form {
5    my ($namespace, $delegate) = @_;
6    $datatypes{$namespace} = $delegate;
7    *{"meta::${namespace}::implementation"} = $delegate;
8    *{"meta::$namespace"} = sub {
9      my ($name, $value) = @_;
10     chomp $value;
11     $data{"${namespace}::$name"} = $value;
12     $delegate->($name, $value);
13   };
14 }
15 meta::define_form 'bootstrap', sub {};
16 meta::define_form 'function', sub {
17   my ($name, $body) = @_;
18   *{$name} = eval "sub {\n$body\n}";
19 };
20 meta::define_form 'code', sub {
21   my ($name, $value) = @_;
22   eval $value;
23 };
24
25 meta::bootstrap('initialization', <<'EOF');
26 my %data;
27 my %datatypes;
28
29 sub meta::define_form {
30   my ($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);

```

```
77 # Save new state:
78 open my $file, '>', $0;
79 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 Fixing the `EOF` markers

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 one attribute at a time:

Listing 5.1 `snippets/serialize-and-serialize-single`

```
meta::function('serialize', <<'EOF');
my @keys = sort keys %data;
join "\n", $data{'bootstrap::initialization'},
           map(serialize_single($_), grep !/^code::/, @keys),
           map(serialize_single($_), grep  /^code::/, @keys),
           "\n__END__";
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
```

Sorting the keys is important. We'll be verifying the output of the serialization function, so it needs to be stable.

Now `code::main` is a bit simpler. With these new functions the file logic becomes:

```
open my $file, '>', $0;
print $file serialize();
close $file;
```

5.2 Verifying serialization

What we've been doing is very unsafe. There isn't a backup file, so if the serialization goes wrong then we'll blindly nuke our original script. This is a big problem, so let's fix it. The new strategy will be to serialize to a temporary file, have that file generate a checksum, and make sure that the checksum is what we expect. Before we can implement such a mechanism, though, we'll need a string hash function.

5.2.1 Implementing the Fowler-Noll Vo hash

At its core, the FNV-1a hash[2] is just a multiply-xor in a loop. Generally it's written like this:

[2] http://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function

Listing 5.2 snippets/fnv-hash.c

```c
int hash (char *s) {
  const int fnv_prime  = 16777619;      // Magic numbers
  const int fnv_offset = 2166136261;
  int result = fnv_offset;
  char c;
  while (c = *s++) {
    result ^= c;
    result *= fnv_prime;
  }
  return result;
}
```

In Perl it's advantageous to vectorize this function for performance reasons. It isn't necessarily sound to do this, but empirically the results seem reasonably well-distributed. Here's the function I ended up with:

Listing 5.3 snippets/fnv-hash-function

```
meta::function('fnv_hash', <<'EOF');
my ($data) = @_;

my ($fnv_prime, $fnv_offset) = (16777619, 2166136261);
my $hash                     = $fnv_offset;
my $modulus                  = 2 ** 32;

$hash = ($hash ^ ($_ & 0xffff) ^ ($_ >> 16)) * $fnv_prime % $modulus
  for unpack 'L*', $data . substr($data, -4) x 8;
$hash;
EOF
```

This produces a 32-bit hash. Ideally we have something of at least 128 bits, just to reduce the likelihood of collision. When I was writing the 128-bit hash I went a bit overboard with hash chaining (which doesn't matter because it isn't a cryptographic hash), but here's the full hash:

Listing 5.4 snippets/fast-hash-function

```
meta::function('fast_hash', <<'EOF');
my ($data)     = @_;
my $piece_size = length($data) >> 3;

my @pieces     = (substr($data, $piece_size * 8) . length($data),
                  map(substr($data, $piece_size * $_, $piece_size), 0 .. 7));
my @hashes     = (fnv_hash($pieces[0]));

push @hashes, fnv_hash($pieces[$_ + 1] . $hashes[$_]) for 0 .. 7;

$hashes[$_] ^= $hashes[$_ + 4] >> 16 | ($hashes[$_ + 4] & 0xffff) << 16 for 0 .. 3;
```

```
$hashes[0]  ^= $hashes[8];

sprintf '%08x' x 4, @hashes[0 .. 3];
EOF
```

The convolutedness of this logic is partially to accommodate for very short strings.

5.2.2 Fixing `EOF` markers again

It's probably fine to use random numbers for EOF markers, but I prefer using a hash of the content. While it's probably about the same either way, it intuitively feels less likely that a string will contain its own hash.[3]

Listing 5.5 `snippets/serialize-single-hash`

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

We can also use the script state to get a tempfile in the `edit` function.[4]

5.2.3 Implementing the `state` function

The "state" of an object is just the hash of its serialization. (This is why it's useful to have the serialization logic factored out.)

Listing 5.6 `snippets/state-function-hash`

```
meta::function('state', <<'EOF');
fast_hash(serialize());
EOF
```

5.2.4 Implementing the `verify` function

`verify` writes a temporary copy, checks its checksum, and returns `0` or `1` depending on whether the checksum came out invalid or valid, respectively. If invalid, it leaves the temporary file there for debugging purposes.

Listing 5.7 `snippets/verify-function`

```
meta::function('verify', <<'EOF');
my $serialized_data = serialize();
my $state           = state();
```

[3] And as we all know, intuition is key when making decisions in math and computer science...

[4] `object` uses `File::Temp` to get temporary filenames. This is a better solution than anything involving pseudorandom names in `/tmp`.

```
4
5  my $temporary_filename = "$0.$state";
6  open my $file, '>', $temporary_filename;
7  print $file $serialized_data;
8  close $file;
9  chmod 0700, $temporary_filename;
10
11 chomp(my $observed_state = join '', qx|perl '$temporary_filename' state|);
12
13 my $result = $observed_state eq $state;
14 unlink $temporary_filename if $result;
15 $result;
16 EOF
```

5.3 Save logic

Now we can use `verify` before overwriting `$0`.

Listing 5.8 `snippets/save-function-and-broken-usage`

```
1  meta::function('save', <<'EOF');
2  if (verify()) {
3    open my $file, '>', $0;
4    print $file serialize();
5    close $file;
6  } else {
7    warn 'Verification failed';
8  }
9  EOF
10
11 meta::code('main', <<'EOF');
12 ...
13 save();
14 EOF
```

5.4 `code::main` fixes

There's actually a fairly serious problem at this point. Every script saves itself unconditionally, which involves creating a temporary filename and verifying its contents. What happens when we run one then? Something like this:

```
$ perl some-script cat function::cat
join "\n", @data{@_};     # Gets this much right
# Now calls save(), which calls verify() to create a new temp script:
```

```
> perl some-script.hash1 state
  hash1                   # Gets this much right
  # Now calls save(), which calls verify() to create a new temp script:
  > perl some-script.hash1.hash2 state
    ...
```

That's not what we want at all. There's no reason to call `save` unless a modification has occurred, so we can make this modification to `code::main`:

Listing 5.9 snippets/code-main-with-fixed-save

```
meta::code('main', <<'EOF');
my $initial_state = state();
my $command = shift @ARGV;
print &$command(@ARGV);    # Also printing the result -- important for state
save() if state() ne $initial_state;
EOF
```

5.5 Final result

At this point we have an extensible and reasonably robust script. Here's what we've got so far:

Listing 5.10 examples/verified

```
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('serialize', <<'EOF');
my @keys = sort keys %data;
join "\n", $data{'bootstrap::initialization'},
           map(serialize_single($_), grep !/^code::/, @keys),
           map(serialize_single($_), grep  /^code::/, @keys),
           "\n__END__";
EOF

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

meta::function('fnv_hash', <<'EOF');
my ($data) = @_;
my ($fnv_prime, $fnv_offset) = (16777619, 2166136261);
my $hash                     = $fnv_offset;
```

```
my $modulus                  = 2 ** 32;
$hash = ($hash ^ ($_ & 0xffff) ^ ($_ >> 16)) * $fnv_prime % $modulus
  for unpack 'L*', $data . substr($data, -4) x 8;
$hash;
EOF

meta::function('fast_hash', <<'EOF');
my ($data)     = @_;
my $piece_size = length($data) >> 3;
my @pieces     = (substr($data, $piece_size * 8) . length($data),
                  map(substr($data, $piece_size * $_, $piece_size), 0 .. 7));
my @hashes     = (fnv_hash($pieces[0]));
push @hashes, fnv_hash($pieces[$_ + 1] . $hashes[$_]) for 0 .. 7;
$hashes[$_] ^= $hashes[$_ + 4] >> 16 | ($hashes[$_ + 4] & 0xffff) << 16 for 0 .. 3;
$hashes[0]  ^= $hashes[8];
sprintf '%08x' x 4, @hashes[0 .. 3];
EOF

meta::function('state', <<'EOF');
fast_hash(serialize());
EOF

meta::function('verify', <<'EOF');
my $serialized_data = serialize();
my $state           = state();

my $temporary_filename = "$0.$state";
open my $file, '>', $temporary_filename;
print $file $serialized_data;
close $file;
chmod 0700, $temporary_filename;
chomp(my $observed_state = join '', qx|perl '$temporary_filename' state|);
my $result = $observed_state eq $state;
unlink $temporary_filename if $result;
$result;
EOF

meta::function('save', <<'EOF');
if (verify()) {
  open my $file, '>', $0;
  print $file serialize();
  close $file;
} else {
  warn 'Verification failed';
}
EOF
```

```
115
116 meta::function('cat', <<'EOF');
117 join "\n", @data{@_};
118 EOF
119
120 meta::function('set', <<'EOF');
121 $data{$_[0]} = join '', <STDIN>;
122 EOF
123
124 meta::function('edit', <<'EOF');
125 my $filename = '/tmp/' . rand();
126 open my $file, '>', $filename;
127 print $file $data{$_[0]};
128 close $file;
129 system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);
130 open my $file, '<', $filename;
131 $data{$_[0]} = join '', <$file>;
132 close $file;
133 EOF
134
135 meta::code('main', <<'EOF');
136 my $initial_state = state();
137 my $command = shift @ARGV;
138 print &$command(@ARGV);
139 save() if state() ne $initial_state;
140 EOF
141
142 __END__
```

Chapter 6

Adding a REPL

There are some ergonomic problems with the script as it stands. First, it should have a shebang line so that we don't have to use `perl` explicitly. But more importantly, it should provide a REPL so that we don't have to keep calling it by name.

The first question is how this should be invoked. It would be cool if we could run the script without arguments and get the REPL, but that will require some changes to the current `code::main`. The "right way" to do it also requires a new data type.

6.1 The `data` data type

Sometimes we just want to store pieces of data without any particular meaning. We could use `bootstrap::` for this, but it's cleaner to introduce a new data type altogether.

Listing 6.1 `snippets/define-form-data`

```
meta::define_form 'data', sub {
  # Define a basic editing interface:
  my ($name, $value) = @_;
  *{$name} = sub {
    my ($command, $value) = @_;
    return $data{"data::$name"} unless @_;
    $data{"data::$name"} = $value if $command eq '=';
  };
};
```

This function we're defining lets us inspect and change a data attribute from the command line. Assuming `data::foo`, for example:

```
$ perl script foo = bar
bar
```

```
$ perl script foo
bar
$ perl script foo = baz
baz
$
```

6.2 Setting up the default action

The default action can be stored in a `data::` attribute:

```
meta::data('default-action', <<'EOF');
shell
EOF

meta::code('main', <<'EOF');
...
my $command = shift @ARGV || $data{'data::default-action'};
print &$command(@ARGV);
...
EOF
```

Since all values are chomped already, we don't have to worry about the newline caused by the heredoc.

6.3 Making the script executable

This isn't hard at all. It means one extra line in the bootstrap logic, and another extra line in `save`:

```
meta::bootstrap('initialization', <<'EOF');
#!/usr/bin/perl
...
EOF

meta::function('save', <<'EOF');
...
  close $file;
  chmod 0744, $0; # Not perfect, but will fix later
...
EOF
```

6.4 The `shell` function

The idea here is to listen for commands from the user and simulate the `@ARGV` interaction pattern. Readline is the simplest way to go about this:

Listing 6.2 snippets/shell-function-1

```
meta::function('shell', <<'EOF');
use Term::ReadLine;
my $term = new Term::ReadLine "$0 shell";
$term->ornaments(0);
my $output = $term->OUT || \*STDOUT;
while (defined($_ = $term->readline("$0\$ "))) {
  my @args = grep length, split /\s+|("[^"\\]*(?:\\.)?")/o;
  my $function_name = shift @args;
  s/^"(.*)"$/\1/o, s/\\\\"/"/go for @args;

  if ($function_name) {
    chomp(my $result = eval {&$function_name(@args)});
    warn $@ if $@;
    print $output $result, "\n" unless $@;
  }
}
EOF
```

This shell function does some minimal quotation-mark parsing so that you can use multi-word arguments, but otherwise it's fairly basic. The script's name is used as the shell prompt.

It's OK to use `use` inside of `eval`ed functions. I think what happens is that it gets processed when the function is first created by `meta::function`. But basically, Perl does the right thing and it works just fine as long as the module exists.

6.5 Taking it to the max: tab-completion

If you have the GNU Readline library installed (Perl defaults to something else otherwise), you can get tab-autocompletion just like you can in Bash. Here's a `complete` function written by my wife Joyce, modified slightly to make sense with this implementation:

Listing 6.3 snippets/complete-function-1

```
meta::function('complete', <<'EOF');
my @attributes = sort keys %data;

sub match {
  my ($text, @options) = @_;
  my @matches = sort grep /^$text/, @options;

  if    (@matches == 0) {return undef;}
  elsif (@matches == 1) {return $matches [0];}
  elsif (@matches >  1) {
```

```
    return ((longest ($matches [0], $matches [@matches - 1])), @matches);
  }
}

sub longest {
  my ($s1, $s2) = @_;
  return substr ($s1, 0, length $1) if ($s1 ^ $s2) =~ /^(\0*)/;
  return '';
}

my ($text, $line) = @_;
match ($text, @attributes);
EOF
```

Using this function is easy; we just add one line to `shell`:

```
$term->Attribs->{attempted_completion_function} = \&complete;
while (defined($_ = $term->readline("$0\$ "))) {
...
```

6.6 Final result

Merging the shell and executable behavior in with the script from the last chapter, we now have:[1]

Listing 6.4 examples/shell

```
#!/usr/bin/perl
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 {
```

[1] You might notice that I'm still using `EOF` as the marker in these scripts. As soon as the script is rewritten it will replace the `EOF`s with hashes; in general, you can use any valid delimiter the first time around and the script will take it from there.

```
  my ($name, $body) = @_;
  *{$name} = eval "sub {\n$body\n}";
};
meta::define_form 'code', sub {
  my ($name, $value) = @_;
  eval $value;
};
meta::define_form 'data', sub {
  # Define a basic editing interface:
  my ($name, $value) = @_;
  *{$name} = sub {
    my ($command, $value) = @_;
    return $data{"data::$name"} unless @_;
    $data{"data::$name"} = $value if $command eq '=';
  };
};

meta::bootstrap('initialization', <<'EOF');
#!/usr/bin/perl
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 $value;
};
meta::define_form 'data', sub {
  # Define a basic editing interface:
  my ($name, $value) = @_;
  *{$name} = sub {
```

```
    my ($command, $value) = @_;
    return $data{"data::$name"} unless @_;
    $data{"data::$name"} = $value if $command eq '=';
  };
};
EOF

meta::data('default-action', <<'EOF');
shell
EOF

meta::function('serialize', <<'EOF');
my @keys = sort keys %data;
join "\n", $data{'bootstrap::initialization'},
           map(serialize_single($_), grep !/^code::/, @keys),
           map(serialize_single($_), grep  /^code::/, @keys),
           "\n__END__";
EOF

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

meta::function('fnv_hash', <<'EOF');
my ($data) = @_;
my ($fnv_prime, $fnv_offset) = (16777619, 2166136261);
my $hash                     = $fnv_offset;
my $modulus                  = 2 ** 32;
$hash = ($hash ^ ($_ & 0xffff) ^ ($_ >> 16)) * $fnv_prime % $modulus
  for unpack 'L*', $data . substr($data, -4) x 8;
$hash;
EOF

meta::function('fast_hash', <<'EOF');
my ($data)     = @_;
my $piece_size = length($data) >> 3;
my @pieces     = (substr($data, $piece_size * 8) . length($data),
                  map(substr($data, $piece_size * $_, $piece_size), 0 .. 7));
my @hashes     = (fnv_hash($pieces[0]));
push @hashes, fnv_hash($pieces[$_ + 1] . $hashes[$_]) for 0 .. 7;
$hashes[$_] ^= $hashes[$_ + 4] >> 16 | ($hashes[$_ + 4] & 0xffff) << 16 for 0 .. 3;
$hashes[0]  ^= $hashes[8];
sprintf '%08x' x 4, @hashes[0 .. 3];
EOF
```

```
110
111 meta::function('state', <<'EOF');
112 fast_hash(serialize());
113 EOF
114
115 meta::function('verify', <<'EOF');
116 my $serialized_data = serialize();
117 my $state           = state();
118
119 my $temporary_filename = "$0.$state";
120 open my $file, '>', $temporary_filename;
121 print $file $serialized_data;
122 close $file;
123 chmod 0700, $temporary_filename;
124 chomp(my $observed_state = join '', qx|perl '$temporary_filename' state|);
125 my $result = $observed_state eq $state;
126 unlink $temporary_filename if $result;
127 $result;
128 EOF
129
130 meta::function('save', <<'EOF');
131 if (verify()) {
132   open my $file, '>', $0;
133   print $file serialize();
134   close $file;
135   chmod 0744, $0;
136 } else {
137   warn 'Verification failed';
138 }
139 EOF
140
141 meta::function('cat', <<'EOF');
142 join "\n", @data{@_};
143 EOF
144
145 meta::function('set', <<'EOF');
146 $data{$_[0]} = join '', <STDIN>;
147 EOF
148
149 meta::function('complete', <<'EOF');
150 my @attributes = sort keys %data;
151 sub match {
152   my ($text, @options) = @_;
153   my @matches = sort grep /^$text/, @options;
154
155   if    (@matches == 0) {return undef;}
```

```
  elsif (@matches == 1) {return $matches [0];}
  elsif (@matches >  1) {
    return ((longest ($matches [0], $matches [@matches - 1])), @matches);
  }
}
sub longest {
  my ($s1, $s2) = @_;
  return substr ($s1, 0, length $1) if ($s1 ^ $s2) =~ /^(\0*)/;
  return '';
}
my ($text, $line) = @_;
match ($text, @attributes);
EOF

meta::function('shell', <<'EOF');
use Term::ReadLine;
my $term = new Term::ReadLine "$0 shell";
$term->ornaments(0);
my $output = $term->OUT || \*STDOUT;
$term->Attribs->{attempted_completion_function} = \&complete;
while (defined($_ = $term->readline("$0\$ "))) {
  my @args = grep length, split /\s+|("[^"\\]*(?:\\.)?")/o;
  my $function_name = shift @args;
  s/^"(.*)"$/\1/o, s/\\\\"/"/go for @args;

  if ($function_name) {
    chomp(my $result = eval {&$function_name(@args)});
    warn $@ if $@;
    print $output $result, "\n" unless $@;
  }
}
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');
my $initial_state = state();
```

```
my $command = shift @ARGV || $data{'data::default-action'};
print &$command(@ARGV);
save() if state() ne $initial_state;
EOF

__END__
```

Chapter 7

Some improvements

Let's step back for a minute and improve things a bit in preparation for some real awesomeness. There are few places that could use improvement. First, there isn't a way to get a list of defined attributes on an object without opening it by hand. Second, the interface exposes too many functions to the user; in particular, things like `complete` aren't useful from the command line. Finally, every data type we define gets put into `bootstrap::initialization`, which causes $O(n)$ redundancy in the size of the data type constructors.

7.1 Useful functions

The most important thing to add is `ls`, which gives you a listing of attributes:[1] Related are `cp` and `rm`, which do what you would expect:

Listing 7.1 `snippets/ls-cp-and-rm-functions`

```
1  meta::function('ls', <<'EOF');
2  join "\n", sort keys %data;
3  EOF
4
5  meta::function('cp', <<'EOF');
6  $data{$_[1]} = $data{$_[0]};
7  EOF
8
9  meta::function('rm', <<'EOF');
10 delete @data{@_};
11 EOF
```

[1] `object` contains a much more sophisticated version of `ls`. It parses options and applies filters to the listing, much like the UNIX `ls` command. I'll go over how to implement this stuff in a later chapter.

Another useful function is `create`, which opens an editor for a new attribute:[2]

Listing 7.2 `snippets/create-function`

```
meta::function('create', <<'EOF');
return edit($_[0]) if exists $data{$_[0]};
$data{$_[0]} = $_[1] || "# Attribute $_[0]";
edit($_[0]);
EOF
```

Now we can create stuff from inside the shell or command-line and have a civilized text-editor interface to do it.

7.2 Making some functions internal

It would be nice to have a distinction between functions meant for public consumption and functions used just inside the script. For example, nobody's going to call `fnv_hash` from the command-line; they'd have to pass it a string in `@ARGV`, which isn't practical. So it's time for a new toplevel mechanism, the `%externalized_functions` table:

```
# In bootstrap::initialization:
my %data;
my %externalized_functions;
my %datatypes;
```

`%externalized_functions` maps every callable function to the attribute that defines it, and only the listed functions will be usable directly from the shell or the command-line. This has an additional benefit of providing much better autocompletion, since the first word in the REPL always names a function.

```
meta::define_form 'data', sub {
  my ($name, $value) = @_;
  $externalized_functions{$name} = "data::$name";
  *{$name} = ...;
};

meta::define_form 'function', sub {
  my ($name, $value) = @_;
  $externalized_functions{$name} = "function::$name";
  *{$name} = ...;
};
```

And here's the new data type:

[2] We can already do this with `edit`, but `object` doesn't let you edit attributes that don't exist. I'll include that behavior in these scripts before too long.

Listing 7.3 snippets/internal-function-type

```
meta::define_form 'internal_function', sub {
  my ($name, $value) = @_;
  *{$name} = eval "sub {\n$value\n}";
};
```

We can now move `fnv_hash`, `fast_hash`, and `complete` into this namespace. We'll need to update `shell` and `complete` to leverage this new information:

Listing 7.4 snippets/shell-2

```
meta::function('shell', <<'EOF');
use Term::ReadLine;
my $term = new Term::ReadLine "$0 shell";
$term->ornaments(0);
my $output = $term->OUT || \*STDOUT;
$term->Attribs->{attempted_completion_function} = \&complete;
while (defined($_ = $term->readline("$0\$ "))) {
  my @args = grep length, split /\s+|("[^"\\]*(?:\\.)?")/o;
  my $function_name = shift @args;
  s/^"(.*)"$/\1/o, s/\\\\"/"/go for @args;

  if ($function_name) {
    if ($externalized_functions{$function_name}) {
      chomp(my $result = eval {&$function_name(@args)});
      warn $@ if $@;
      print $output $result, "\n" unless $@;
    } else {
      warn "Command not found: '$function_name' (use 'ls' to see available commands)";
    }
  }
}
EOF
```

Listing 7.5 snippets/complete-2

```
meta::function('complete', <<'EOF');
my @functions  = sort keys %externalized_functions;
my @attributes = sort keys %data;
sub match {
  my ($text, @options) = @_;
  my @matches = sort grep /^$text/, @options;
  if    (@matches == 0) {return undef;}
  elsif (@matches == 1) {return $matches [0];}
  elsif (@matches >  1) {
    return ((longest ($matches [0], $matches [@matches - 1])), @matches);
  }
}
```

```
sub longest {
  my ($s1, $s2) = @_;
  return substr ($s1, 0, length $1) if ($s1 ^ $s2) =~ /^(\0*)/;
  return '';
}
my ($text, $line) = @_;
if ($line =~ / /) {
  # Start matching attribute names.
  match ($text, @attributes);
} else {
  # Start of line, so it's a function.
  match ($text, @functions);
}
EOF
```

7.3 Separate attributes for data types

It's cumbersome to have all of the data types go in `bootstrap::initialization`. Better is to break the code into separate attributes. To do this we'll need to restructure the scripts a little bit.

Up until now the "stuff first, code second" approach has worked out all right. But now we want to evaluate stuff at the beginning and at the end, and if this keeps up it could get out of hand. Better is to have `serialize` generate a call into some function that will be defined, and do away with `code::` altogether. We can use a new namespace `meta::` for stuff that needs to be evaluated at the beginning. So basically, instead of this:

```
bootstrap
  types
functions
code
```

we'd have this:

```
bootstrap
meta definitions
functions
call to internal::main()
```

Here's what the new `serialize` looks like:

Listing 7.6 snippets/serialize-with-internal-main

```
my @keys = sort keys %data;
join "\n", $data{'bootstrap::initialization'},
           map(serialize_single($_),
```

```
4               grep( /^meta::/, @keys),
5               grep(!/^meta::/, @keys)),
6             "internal::main();",
7             "__END__";
```

And here's the definition for `meta::` (it's identical to the one we used to have for `code::`). This is the only `define_form` invocation in `bootstrap::initialization`; the others now reside in their own attributes.

Listing 7.7 snippets/define-form-meta

```
1 meta::define_form 'meta', sub {
2   my ($name, $value) = @_;
3   eval $value;
4 };
```

Here are the new type definitions:

```
meta::meta('type::data', <<'EOF');
meta::define_form 'data', sub {...};
EOF
meta::meta('type::function', <<'EOF');
meta::define_form 'function', sub {...};
EOF
meta::meta('type::bootstrap', <<'EOF');
meta::define_form 'bootstrap', sub {};
EOF
...
```

7.3.1 Factoring externalization

While we're cleaning up meta-stuff, it's worth thinking about factoring out externalization. There isn't a particularly good reason to keep manually assigning to `%externalized_functions`; better is to abstract this detail into a function. To do this, we'll want a meta-library:

Listing 7.8 snippets/meta-externalize

```
1 meta::meta('externalize', <<'EOF');
2 sub meta::externalize {
3   my ($name, $attribute, $implementation) = @_;
4   $externalized_functions{$name} = $attribute;
5   *{$name} = $implementation;
6 }
7 EOF
```

This meta-definition is available to the others because it sorts first.[3] Now instead of manually externalizing stuff, data types like `function::` and `data::` can just use `meta::externalize`:

[3] Which is a horrible way to manage dependencies, but it's worked so far.

Listing 7.9 snippets/function-type-with-externalize

```
meta::meta('type::function', <<'EOF');
meta::define_form 'function', sub {
  my ($name, $value) = @_;
  meta::externalize $name, "function::$name", eval "sub {\n$value\n}";
};
EOF
```

7.4 Abstracting %data

Another issue worth fixing is that you can assign into %data arbitrarily, particularly in ways that end up breaking deserialization. For instance, nothing is stopping you from creating a key called foo::bar even though there isn't a namespace called foo::. This problem can be solved at the interface level (i.e. inside edit, set, and such), but it's probably more useful to go a step further and abstract all access to %data.

Rather than writing to %data, then, we'll use an internal function called associate; and to read from it we'll use retrieve. These two functions also benefit from a couple more to separate out namespace components. The namespace function gives you the base part, and the attribute function gives you the rest.[4]

Listing 7.10 snippets/namespace-attribute-retrieve-associate-functions

```
meta::internal_function('namespace', <<'EOF');
my ($name) = @_;
$name =~ s/::.*$//;
$name;
EOF

meta::internal_function('attribute', <<'EOF');
my ($name) = @_;
$name =~ s/^[^:]*::/;
$name;
EOF

meta::internal_function('retrieve', <<'EOF');
my @results = map defined $data{$_} ? $data{$_} : file::read($_), @_;
wantarray ? @results : $results[0];
EOF

meta::internal_function('associate', <<'EOF');
my ($name, $value, %options) = @_;
my $namespace = namespace($name);
```

[4] All four of these functions are taken directly from object.

```
die "Namespace $namespace does not exist" unless $datatypes{$namespace};
$data{$name} = $value;
execute($name) if $options{'execute'};
EOF
```

7.4.1 Dynamic execution

One problem with the way we've defined `cp` is that you'll have to close and reopen the shell to get new functions to take effect. This is because while we're assigning into `%data`, we're not calling the handler associated with the namespace. The simplest way to fix this is to dynamically invoke that handler:

Listing 7.11 `snippets/execute-function`

```
meta::internal_function('execute', <<'EOF');
my ($name, %options) = @_;
my $namespace = namespace($name);
eval {&{"meta::$namespace"}(attribute($name), retrieve($name))};
warn $@ if $@ && $options{'carp'};
EOF
```

`associate` is already hooked up to use this function; all you have to do is pass an extra option:

```
associate('function::foo', '...', execute => 1);
```

7.5 Final result

Integrating all of these improvements into the previous chapter's script yields this monumental piece of work:[5]

Listing 7.12 `examples/some-improvements`

```
#!/usr/bin/perl
my %data;
my %externalized_functions;
my %datatypes;

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

[5] This is the last full listing I'll provide here. The remaining chapters cover the concepts required to get from here to `object`. At this point the stuff going on in `object` should more or less make sense, though you'll want to use `ls-a` rather than `ls` to get a full listing of attributes.

```
    chomp $value;
    $data{"${namespace}::$name"} = $value;
    $delegate->($name, $value);
  };
}

meta::define_form 'meta', sub {
  my ($name, $value) = @_;
  eval $value;
};

meta::meta('externalize', <<'EOF');
sub meta::externalize {
  my ($name, $attribute, $implementation) = @_;
  $externalized_functions{$name} = $attribute;
  *{$name} = $implementation;
}
EOF

meta::meta('type::bootstrap', <<'EOF');
meta::define_form 'bootstrap', sub {};
EOF

meta::meta('type::function', <<'EOF');
meta::define_form 'function', sub {
  my ($name, $body) = @_;
  meta::externalize $name, "function::$name", eval "sub {\n$body\n}";
};
EOF

meta::meta('type::internal_function', <<'EOF');
meta::define_form 'internal_function', sub {
  my ($name, $value) = @_;
  *{$name} = eval "sub {\n$value\n}";
};
EOF

meta::meta('type::data', <<'EOF');
meta::define_form 'data', sub {
  # Define a basic editing interface:
  my ($name, $value) = @_;
  meta::externalize $name, "data::$name", sub {
    my ($command, $value) = @_;
    return $data{"data::$name"} unless @_;
    $data{"data::$name"} = $value if $command eq '=';
  };
```

```
};
EOF

meta::bootstrap('initialization', <<'EOF');
#!/usr/bin/perl
my %data;
my %externalized_functions;
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 'meta', sub {
  my ($name, $value) = @_;
  eval $value;
};
EOF

meta::data('default-action', <<'EOF');
shell
EOF

meta::internal_function('namespace', <<'EOF');
my ($name) = @_;
$name =~ s/::.*$//;
$name;
EOF

meta::internal_function('attribute', <<'EOF');
my ($name) = @_;
$name =~ s/^[^:]*::/;
$name;
EOF

meta::internal_function('retrieve', <<'EOF');
my @results = map defined $data{$_} ? $data{$_} : file::read($_), @_;
wantarray ? @results : $results[0];
```

```
EOF

meta::internal_function('associate', <<'EOF');
my ($name, $value, %options) = @_;
my $namespace = namespace($name);
die "Namespace $namespace does not exist" unless $datatypes{$namespace};
$data{$name} = $value;
execute($name) if $options{'execute'};
EOF

meta::internal_function('execute', <<'EOF');
my ($name, %options) = @_;
my $namespace = namespace($name);
eval {&{"meta::$namespace"}(attribute($name), retrieve($name))};
warn $@ if $@ && $options{'carp'};
EOF

meta::function('serialize', <<'EOF');
my @keys = sort keys %data;
join "\n", $data{'bootstrap::initialization'},
           map(serialize_single($_),
             grep( /^meta::/, @keys),
             grep(!/^meta::/, @keys)),
           "internal::main();",
           "__END__";
EOF

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

meta::function('fnv_hash', <<'EOF');
my ($data) = @_;
my ($fnv_prime, $fnv_offset) = (16777619, 2166136261);
my $hash                     = $fnv_offset;
my $modulus                  = 2 ** 32;
$hash = ($hash ^ ($_ & 0xffff) ^ ($_ >> 16)) * $fnv_prime % $modulus
  for unpack 'L*', $data . substr($data, -4) x 8;
$hash;
EOF

meta::function('fast_hash', <<'EOF');
my ($data)     = @_;
my $piece_size = length($data) >> 3;
```

```
my @pieces     = (substr($data, $piece_size * 8) . length($data),
                  map(substr($data, $piece_size * $_, $piece_size), 0 .. 7));
my @hashes     = (fnv_hash($pieces[0]));
push @hashes, fnv_hash($pieces[$_ + 1] . $hashes[$_]) for 0 .. 7;
$hashes[$_] ^= $hashes[$_ + 4] >> 16 | ($hashes[$_ + 4] & 0xffff) << 16 for 0 .. 3;
$hashes[0]  ^= $hashes[8];
sprintf '%08x' x 4, @hashes[0 .. 3];
EOF

meta::function('state', <<'EOF');
fast_hash(serialize());
EOF

meta::function('verify', <<'EOF');
my $serialized_data = serialize();
my $state           = state();

my $temporary_filename = "$0.$state";
open my $file, '>', $temporary_filename;
print $file $serialized_data;
close $file;
chmod 0700, $temporary_filename;
chomp(my $observed_state = join '', qx|perl '$temporary_filename' state|);
my $result = $observed_state eq $state;
unlink $temporary_filename if $result;
$result;
EOF

meta::function('save', <<'EOF');
if (verify()) {
  open my $file, '>', $0;
  print $file serialize();
  close $file;
  chmod 0744, $0;
} else {
  warn 'Verification failed';
}
EOF

meta::function('ls', <<'EOF');
join "\n", sort keys %data;
EOF

meta::function('cp', <<'EOF');
associate($_[1], retrieve($_[0]));
EOF
```

```
196
197 meta::function('rm', <<'EOF');
198 delete @data{@_};
199 EOF
200
201 meta::function('cat', <<'EOF');
202 join "\n", @data{@_};
203 EOF
204
205 meta::function('create', <<'EOF');
206 return edit($_[0]) if exists $data{$_[0]};
207 associate($_[0], $_[1] || "# Attribute $_[0]");
208 edit($_[0]);
209 EOF
210
211 meta::function('set', <<'EOF');
212 $data{$_[0]} = join '', <STDIN>;
213 EOF
214
215 meta::function('complete', <<'EOF');
216 my @functions  = sort keys %externalized_functions;
217 my @attributes = sort keys %data;
218 sub match {
219   my ($text, @options) = @_;
220   my @matches = sort grep /^$text/, @options;
221   if    (@matches == 0) {return undef;}
222   elsif (@matches == 1) {return $matches [0];}
223   elsif (@matches >  1) {
224     return ((longest ($matches [0], $matches [@matches - 1])), @matches);
225   }
226 }
227 sub longest {
228   my ($s1, $s2) = @_;
229   return substr ($s1, 0, length $1) if ($s1 ^ $s2) =~ /^(\0*)/;
230   return '';
231 }
232 my ($text, $line) = @_;
233 if ($line =~ / /) {
234   # Start matching attribute names.
235   match ($text, @attributes);
236 } else {
237   # Start of line, so it's a function.
238   match ($text, @functions);
239 }
240 EOF
241
```

```
meta::internal_function('shell', <<'EOF');
use Term::ReadLine;
my $term = new Term::ReadLine "$0 shell";
$term->ornaments(0);
my $output = $term->OUT || \*STDOUT;
$term->Attribs->{attempted_completion_function} = \&complete;
while (defined($_ = $term->readline("$0\$ "))) {
  my @args = grep length, split /\s+|("[^"\\]*(?:\\.)?")/o;
  my $function_name = shift @args;
  s/^"(.*)"$/\1/o, s/\\\\"/"/go for @args;

  if ($function_name) {
    if ($externalized_functions{$function_name}) {
      chomp(my $result = eval {&$function_name(@args)});
      warn $@ if $@;
      print $output $result, "\n" unless $@;
    } else {
      warn "Command not found: '$function_name' (use 'ls' to see available commands)";
    }
  }
}
EOF

meta::function('edit', <<'EOF');
my $filename = '/tmp/' . rand();
open my $file, '>', $filename;
print $file retrieve($_[0]);
close $file;
system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);
open my $file, '<', $filename;
associate($_[0]}, join '', <$file>);
close $file;
EOF

meta::internal_function('internal::main', <<'EOF');
my $initial_state = state();
my $command = shift @ARGV || retrieve('data::default-action');
print &$command(@ARGV);
save() if state() ne $initial_state;
EOF

internal::main();

__END__
```

Part II

The Fun Stuff

Chapter 8

Rendering as HTML

It turns out that browsers are very lenient about what they're willing to render as HTML. From what I've seen, anything that ends with `.html` will be rendered this way, with extra stuff inserted into the <body> as toplevel text. This is perfect for doing some fun stuff with self-modifying Perl files.

First, I should note that the actual UI implementation is in Caterwaul 0.x and is beyond the scope of this guide. But the basic ideas still apply. There are two things you'll want to do to make your script HTML-renderable. The first is to create a `bootstrap::` attribute with some markup in it. It's useful for it to be a `bootstrap::` attribute because these are unevaluated. Here's `bootstrap::html` from `object`:

Listing 8.1 snippets/object/bootstrap::html

```
<html>
  <head>
  <meta http-equiv='content-type' content='text/html; charset=UTF-8' />
  <link rel='stylesheet' href='http://spencertipping.com/perl-objects/web/style.css'/>

  <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.5.2/jquery.min.js'></script>
  <script src='http://spencertipping.com/caterwaul/caterwaul.all.min.js'></script>
  <script src='http://spencertipping.com/montenegro/montenegro.client.js'></script>
  <script src='http://spencertipping.com/perl-objects/web/attribute-parser.js'></script>
  <script src='http://spencertipping.com/perl-objects/web/interface.js'></script>
  </head>
  <body></body>
</html>
```

If you copy this attribute into a Perl file, it should indeed render as a web page. There's really only one problem with it, which is that before `bootstrap::html` is loaded you'll see a bunch of garbage on the screen. Rather than solve this in any sane and correct way, I just implemented this hack in

`bootstrap::initialization`:[1]

Listing 8.2 `snippets/object/bootstrap::initialization-div`

```
1 # For the benefit of HTML viewers (this is hack):
2 # <div id='cover' style='position: absolute; left: 0; top: 0; \
3     width: 10000px; height: 10000px; background: white'></div>
```

This covers up the garbage until a function in `interface.js` removes the covering `<div>`.

By the way, I highly recommend using the HTML view of Perl objects while reading the rest of this guide. You can get to them here: http://spencertipping.com/perl-objects (each `.html` file is just a symlink to the Perl object of the same name).

[1] The backslash before the end of the line isn't in the real `object`, I just put it here to indicate that the line should be continued, not broken.

Chapter 9

`eval` backtraces

Our script is fairly awesome so far. It prevents us from creating attributes in namespaces that don't exist, since that would cause incorrect serialization, it verifies before it saves, etc. But there's one problem. Take a look at the error messages we get:

```
$ perl examples/some-improvements
examples/some-improvements$ create foo::bar
Namespace foo does not exist at (eval 9) line 4.
examples/some-improvements$
```

If there's a problem in some attribute, we have no information about the location of the error other than "eval *n*" and the line number relative to that. `object` solves this problem:

```
$ object
object$ create foo::bar
  [error] Namespace foo does not exist at internal_function::associate line 4.
object$
```

The key is to wrap `eval` in such a way that we can later resolve the meaningless numbers into useful locations. And to do this, we're going to need to modify the bootstrap code again.

```
my %data;
my %externalized_functions;
my %datatypes;
my %locations;      # Maps eval-numbers to attribute names
```

There's a beautiful hack to handle the `eval` processing. Watch this (also in `bootstrap::initialization`):[1]

[1] It actually doesn't have to be inside the bootstrap code, but it doesn't change often and is useful to have around, so I decided to put it there to save time.

Listing 9.1 snippets/meta-eval-in

```
sub meta::eval_in {
  my ($what, $where) = @_;
  # Obtain next eval-number and alias it to the designated location
  @locations{eval('__FILE__') =~ /\(eval (\d+)\)/} = ($where);  # <- step 1
  my $result = eval $what;                                      # <- step 2
  $@ =~ s/\(eval \d+\)/$where/ if $@;
  warn $@ if $@;
  $result;
}
```

By evaling `__FILE__`, we get the current eval number. So the next one will be whatever we `eval` next. This means that in the shell sessions above, `%locations` contains a mapping from `9` to `internal_function::associate`. Here's the function that converts an `eval` index into an attribute name:

Listing 9.2 snippets/translate-backtrace-function

```
meta::internal_function('translate_backtrace', <<'EOF');
my ($trace) = @_;
$trace =~ s/\(eval (\d+)\)/$locations{$1 - 1}/g;
$trace;
EOF
```

Notice that we're subtracting one. The `eval` number that triggered the error will be one greater than the one we stored.[2]

Now that we have this mechanism, we can go back and convert `eval` calls into `meta::eval_in`:

Listing 9.3 snippets/using-eval-in

```
meta::define_form 'function', sub {
  my ($name, $value) = @_;
  meta::externalize $name, "function::$name",
    meta::eval_in("sub {\n$value\n}", "function::$name");
};

meta::define_form 'internal_function', sub {
  my ($name, $value) = @_;
  *{$name} =
    meta::eval_in("sub {\n$value\n}", "internal_function::$name");
};
```

[2] Good API design would resolve this ahead-of-time rather than at lookup time. I haven't gotten around to changing it yet though.

Chapter 10

Archiving state

Suppose you're about to do something risky with a script and you want to take a snapshot that you can restore to. You could copy into another file, but that's a brute-force approach and it requires you to exit the script's shell. Better is to have some kind of internal state management, and that's where explicit states come into play.

Remember that `%data` is just a variable; we can do all of the usual things with it. We can store a state by doing a partial serialization into an attribute, and we can restore from that state by `eval`ing that attribute. To do this we're going to need another namespace.

Listing 10.1 `snippets/state-type`

```
meta::meta('type::state', <<'EOF');
# No action when a state is defined
meta::define_form 'state', \&meta::bootstrap::implementation;
EOF
```

10.1 Saving state

It's tempting to think that this code would do what we want:

```
# Won't work:
associate("state::$_[0]", serialize());
```

Unfortunately, `serialize` generates three things that we don't want. These are the bootstrap section at the beginning, the call to `internal::main()` at the end, and any attribute in the `state::` namespace.[1] We'll need to write a separate function to serialize just what we want:

[1] If some states contained others, the script size would grow exponentially in the number of states.

Listing 10.2 snippets/current-state-function

```
meta::function('current-state', <<'EOF');
my @valid_keys   = grep ! /^state::/, sort keys %data;
my @ordered_keys = (grep(/^meta::/, @valid_keys), grep(! /^meta::/, @valid_keys));
join "\n", map serialize_single($_), @ordered_keys;
EOF
```

And here's a `save-state` function to automate the state creation process:

Listing 10.3 snippets/save-state-function

```
meta::function('save-state', <<'EOF');
my ($state_name) = @_;
associate("state::$state_name", &{'current-state'}());
EOF
```

10.2 Loading state

This is not as straightforward as saving state. Because we're modifying `%data` live, we have to be careful about what happens in the event that something goes wrong. We also don't want to have stray `%data` elements or externalized functions. The easiest way to defend against errors is to save the current state before applying a new one. Here's the implementation of `load-state`:

Listing 10.4 snippets/load-state-function

```
meta::function('load-state', <<'EOF');
my ($state_name) = @_;
my $state = retrieve("state::$state_name");
&{'save-state'}('_');     # Make a backup
delete $data{$_} for grep ! /^state::/, keys %data;
%externalized_functions = ();
eval($state);             # Apply the new state
warn $@ if $@;
verify();                 # Make sure it worked
EOF
```

If the load failed for some reason, you can restore using `load-state _`. If it failed badly enough to bork your `load-state` function, then you have a problem.

10.3 The `hypothetically` function

Related to state management is a function called `hypothetically`, which lets you try something out and then revert. It's used internally to examine the

state of a modified copy without actually committing changes.[2] Here's how it's defined:

Listing 10.5 snippets/hypothetically-function

```
meta::internal_function('hypothetically', <<'EOF');
my %data_backup   = %data;
my ($side_effect) = @_;
my $return_value  = eval {&$side_effect()};
%data = %data_backup;
die $@ if $@;
$return_value;
EOF
```

You can use it like this:

```
my $x = hypothetically(sub {
  associate('data::foo', '10');
  retrieve('data::foo');
});
my $y = retrieve('data::foo');
# now $x eq '10' and $y is undef
```

[2] This is covered in chapter 11.

Chapter 11

Cloning and inheritance

This is probably the single coolest thing about self-modifying Perl programs. You've probably had this looming feeling that propagating updated versions of functions between scripts was going to be a complete nightmare. For a long time this was indeed the case; I had shell scripts that copied attributes out of one script and into another. Luckily I got tired of doing things that way and came up with the inheritance mechanism that's used now.

Inheritance isn't as simple as copying all of the attributes from one script into another. Certain namespaces like `data::` are script-specific, for instance. We'll need to have some way to keep track of which namespaces should be inherited.

Another issue is getting attributes from one script into another. My first implementation of inheritance retrieved each attribute individually. It used `ls` and `cat` for the transfer, which involved $O(n)$ runs of whichever script was being inherited from. Obviously it was really slow. $O(n)$ runs of a function containing n functions means $O(n^2)$ total time, and Perl isn't blazingly fast at `eval`ing functions. Later on I extended `serialize` to return a bundle of attributes that the child then `eval`ed.

11.1 Tracking inheritability

We're going to need another toplevel field if we want to store data about data types. We can't use `%data`, since we don't really want to save it (whatever we're storing would be regenerated automatically anyway). What we really need is a way to store transient information:

```
my %data;
my %externalized_functions;
my %datatypes;
my %transient;
```

`%transient` does nothing except store stuff while the script is running, and all of its information is discarded when the script exits. It's basically just a temporary workspace where we can stash stuff.

We can now use `%transient` to store things about data types. For convenience let's define `meta::configure` to do this for us:[1]

Listing 11.1 snippets/meta-configure

```
meta::meta('configure', <<'EOF');
sub meta::configure {
  my ($datatype, %options) = @_;
  $transient{$_}{$datatype} = $options{$_} for keys %options;
}
EOF
```

Now we can add a configuration to each datatype we define:

```
meta::meta('type::function', <<'EOF');
meta::configure 'function', inherit => 1;
meta::define_form 'function', ...;
EOF

meta::meta('type::data', <<'EOF');
meta::configure 'data', inherit => 0;
meta::define_form 'data', ...;
EOF

meta::meta('type::internal_function', <<'EOF');
meta::configure 'internal_function', inherit => 1;
...
EOF

meta::meta('type::bootstrap', <<'EOF');
meta::configure 'bootstrap', inherit => 1;
...
EOF

meta::meta('type::state', <<'EOF');
meta::configure 'state', inherit => 0;
...
EOF
```

[1] For some reason I decided to store the keys in the odd order of `option-namespace` instead of the other way around. I'm still not sure why I did it this way, but it doesn't seem to cause problems.

11.2 Extensions to `serialize`

`serialize` needs to be able to give us a bundle of code to create just the attributes that should be inherited. While we're at it, it would also be nice if it handed us just the `meta::` attributes and then just the non-`meta::` attributes. This way we can make sure that the `meta::` attributes didn't break anything and bail out early if they did.

None of this is particularly challenging, but given that we're going to invoke `serialize` externally we should probably fix the `%options` stuff. (The last thing we want to write is something like `qx($script serialize partial 1 meta 1 inheritable 1)`). What we need is an adapter that turns command-line options into Perl hashes.[2] Here's a function that uses `Getopt`-style parsing:

Listing 11.2 `snippets/separate-options-function`

```
meta::internal_function('separate_options', <<'EOF');
# Things with one dash are short-form options, two dashes are long-form.
# Characters after short-form are combined; so -auv4 becomes -a -u -v -4.
# Also finds equivalences; so --foo=bar separates into $$options{'--foo'} eq 'bar'.
# Stops processing at the -- option, and removes it. Everything after that
# is considered to be an 'other' argument.

# The only form not supported by this function is the short-form with argument.
# To pass keyed arguments, you need to use long-form options.
my @parseable;
push @parseable, shift @_ until ! @_ or $_[0] eq '--';

my @singles = grep /^-[^-]/, @parseable;
my @longs   = grep /^--/,    @parseable;
my @others  = grep ! /^-/,   @parseable;
my @singles = map /-(.{2,})/ ? map("-$_", split(//, $1)) : $_, @singles;
my %options;
  $options{$1} = $2 for grep /^([^=]+)=(.*)$/, @longs;
++$options{$_}      for grep ! /=/, @singles, @longs;

({%options}, @others, @_);
EOF
```

The output of this function is a reference to a hash of any keyword arguments (where short-form arguments are treated as increments) followed by any non-switch arguments (either because they came after `--` or because they didn't start with a dash at all. For example, processing the arguments `-xy --z=foo bar` would yield `({-x => 1, -y => 1, --z => foo}, bar)`.[3]

Given the ability to pipe options into `serialize` on the command-line, we

[2] OK, I'm making a jump here. Later it will become clearer why it's good to do it this way.

[3] Given the similarity, I don't remember why I didn't just use `Getopt::Long` for this stuff. I think I must have been having a NIH day.

just need to have it support a reasonably flexible selection interface. We'll later need to have `ls` support the same options, so let's factor the key selector into its own function:

Listing 11.3 snippets/select-keys-function

```
meta::internal_function('select_keys', <<'EOF');
my %options  = @_;
my $criteria = $options{'--criteria'} ||
               $options{'--namespace'} && "^$options{'--namespace'}::" || '.';
grep /$criteria/ && (! $options{'-i'} ||   $transient{inherit}{namespace($_)}) &&
                    (! $options{'-I'} || ! $transient{inherit}{namespace($_)}) &&
                    (! $options{'-S'} || ! /^state::/o) &&
                    (! $options{'-m'} ||   /^meta::/o) &&
                    (! $options{'-M'} || ! /^meta::/o), sort keys %data;
EOF
```

This function takes the `%options` hash output by `separate_options` as input and returns a list of keys into `%data`. The somewhat odd logical structure of the `grep` predicate is just implication: "if `$options{'-i'}` is set, then the key's namespace must be inheritable."

The new `serialize` function is fairly simple; most of the heavy lifting is already done:

Listing 11.4 snippets/serialize-final

```
meta::function('serialize', <<'EOF');
my ($options, @criteria) = separate_options(@_);
my $partial = $$options{'-p'};
my $criteria = join '|', @criteria;
my @attributes = map serialize_single($_),
  select_keys(%$options, '-m' => 1, '--criteria' => $criteria),
  select_keys(%$options, '-M' => 1, '--criteria' => $criteria);
my @final_array = @{$partial ? \@attributes :
                                [retrieve('bootstrap::initialization'),
                                 @attributes,
                                 'internal::main();', '',
                                 '__END__']};
join "\n", @final_array;
EOF
```

`-m` and `-M` select `meta::` and non-`meta::` attributes, respectively. We also provide criteria if the user has selected any. They're joined together with a pipe symbol because that forms a disjunction inside a regular expression (and criteria are regexps for attribute names). We also, somewhat importantly, have a `-p` switch to produce a partial serialization. This leaves off the bootstrap code and the `internal::main()` call. The only difference between this and `current-state` is that `current-state` also leaves out `state::` attributes.[4]

[4] This version of `serialize` also will do that for you if you pass the `-S` option.

11.3 The `update-from` function

Here's where things start to get interesting. `update-from` handles the case where you have two scripts `base` and `child`, and you want `child` to inherit stuff from `base`.[5] Here's a basic implementation:

Listing 11.5 `snippets/update-from-function`

```
meta::function('update-from', <<'EOF');
my ($options, @targets) = separate_options(@_);
my %options = %$options;
@targets or die 'Must specify at least one target to update from';
my $save_state = ! ($options{'-n'} || $options{'--no-save'});
my $force      =    $options{'-f'} || $options{'--force'};

&{'save-state'}('before-update') if $save_state;
for my $target (@targets) {
  eval qx($target serialize -ipm);
  eval qx($target serialize -ipM);
  reload();     # We're about to define this
  unless (verify()) {
    if ($force) {
      warn 'Verification failed, but keeping new state';
    } else {
      warn "Verification failed after $target; reverting";
      return &{'load-state'}('before-update') if $save_state;
    }
  }
}
EOF
```

If `child` has this function, you can update it this way (assuming you've set execute permissions):

```
$ ./child update-from ./base
$
```

If all goes well it will execute without failing. `base` will be run twice: once to grab inheritable `meta::`-attributes, again to grab inheritable other attributes.

The `reload` function just calls `execute` on each member of `%data`. It's there to make sure that the new attributes and the old attributes work well together. Here's the definition:

Listing 11.6 `snippets/reload-function`

```
meta::function('reload', <<'EOF');
execute($_) for grep ! /^bootstrap::/, keys %data;
EOF
```

[5] "Stuff" is deliberately vague. Presumably we want to inherit every inheritable attribute though.

11.4 Managing parents

If you're using scripts as add-on modules, it gets tiring to issue *n* `update-from` commands when you're using *n* modules. Further, you have to type out the whole path of the module, which might be in a separate directory. Finally, attributes that later get deleted or renamed in the addon modules won't be cleaned up in the child script. The way we've got inheritance set up is woefully incomplete.

The missing piece is parent tracking. We need to keep track of two things:

1. Which scripts have I inherited from?
2. Which properties did I inherit from each one?

Fortunately this isn't too hard. We just need a new namespace and some new functions.

11.4.1 The `parent::` namespace

We can use the `parent::` namespace to answer both of the above questions. Each attribute will take its name from the path to a script (e.g. `parent::/usr/bin/script1`), and its contents will be a newline-separated string of the attributes inherited from that script. Here's the type definition:

Listing 11.7 `snippets/parent-type`

```
meta::meta('type::parent', <<'EOF');
meta::configure 'parent', inherit => 1; # Transitive parents (explained below)
meta::define_form 'parent', \&meta::bootstrap::implementation;
EOF
```

Now we need to update `update-from` to populate this namespace. Before we do, though, we need to define uniqueness.

11.4.2 Uniqueness

Let's suppose you've got three objects `a`, `b`, and `c`, and each inherits from the previous one.[6] Then `b`'s parent is `a`, and `c`'s parent is `b`. It's important for `c` to know that `a` is also its parent. The reason is that otherwise you're forced to update `b` before `c`, since `c` inherits only from `b`.

It's ultimately for this reason that `parent::` attributes get inherited. Inheritance is certainly transitive (if `c` inherits from `b`, which inherits from `a`, then `c` inherits from `a`). There are, however, some logistical matters to be dealt with. The most important one has to do with ordering.

It shouldn't matter in which order the parents are inherited from. This is an interesting requirement, because it means that the things an object inherits

[6] I'm presupposing that inheritance works automatically even though we haven't defined it quite yet.

from each of its parents form disjoint sets. The only way to pull this off is if each object keeps track of how it's different from its own parents. The attributes that the child has but the parent doesn't are considered unique.

11.4.3 Updating `ls` and `serialize`

We need a way to ask `ls` for the list of unique attributes for an object, and then ask `serialize` to give us just those attributes. To do this, we'll add a `-u` option (and for symmetry a complement `-U`) to `select_keys`:

Listing 11.8 snippets/select-keys-final

```
meta::internal_function('select_keys', <<'EOF');
my %options   = @_;
my %inherited = map {$_ => 1} split /\n/o, join "\n",
                  retrieve(grep /^parent::/o, sort keys %data)
                  if $options{'-u'} or $options{'-U'};
my $criteria  = $options{'--criteria'} ||
                $options{'--namespace'} && "^$options{'--namespace'}::" || '.';
grep /$criteria/ && (! $options{'-u'} || ! $inherited{$_}) &&
                    (! $options{'-U'} ||   $inherited{$_}) &&
                    (! $options{'-i'} ||   $transient{inherit}{namespace($_)}) &&
                    (! $options{'-I'} || ! $transient{inherit}{namespace($_)}) &&
                    (! $options{'-S'} || ! /^state::/o) &&
                    (! $options{'-m'} ||   /^meta::/o) &&
                    (! $options{'-M'} || ! /^meta::/o), sort keys %data;
EOF
```

The extra logic here searches through all of the `parent::` attributes to find properties that the parents also contain. If any parent contains one, then the property isn't unique.

`serialize` will get this new behavior automatically, since it just forwards its options to `select_keys`. But we haven't modified `ls` in a long time; it doesn't know anything about options. There are actually quite a few enhancements that we could make to `ls`, but for now let's keep it simple and change it only as necessary:

Listing 11.9 snippets/ls-with-options

```
meta::function('ls', <<'EOF');
my ($options, @criteria) = separate_options(@_);
join "\n", select_keys('--criteria' => join '|', @criteria, %$options);
EOF
```

Now you can say things like `./script ls -iu` to get a listing of attributes that are both inheritable and unique.

11.4.4 An updated `update-from` function

`update-from` now has three new responsibilities. One is to record the fact that we updated from another script, which involves creating a new `parent::` attribute for each inheritance operation. Another is to ask each new parent which attributes it intends to define. The last thing it needs to do is clean up any attributes that some parent used to define but no longer does.[7] Here's the new implementation:

Listing 11.10 `snippets/update-from-final`

```
my ($options, @targets) = separate_options(@_);
my %options = %$options;
@targets or die 'Must specify at least one target to update from';
my $save_state = ! ($options{'-n'} || $options{'--no-save'});
my $no_parents =    $options{'-P'} || $options{'--no-parent'} || $options{'--no-parents'};
my $force      =    $options{'-f'} || $options{'--force'};

&{'save-state'}('before-update') if $save_state;
for my $target (@targets) {
  # The -a flag will become relevant once we add formatting to 'ls'
  my $attributes = join '', qx($target ls -aiu);
  warn "Skipping unreachable object $target" unless $attributes;
  if ($attributes) {
    # Remove keys that the parent used to define but doesn't anymore:
    rm(split /\n/, retrieve("parent::$target")) if $data{"parent::$target"};
    associate("parent::$target", $attributes) unless $no_parents;
    eval qx($target serialize -ipmu);
    eval qx($target serialize -ipMu);
    warn $@ if $@;
    reload();

    if (verify()) {
      print "Successfully updated from $_[0]. ",
            "Run 'load-state before-update' to undo this change.\n" if $save_state;
    } elsif ($force) {
      warn 'The object failed verification, but the failure state has been ' .
           'kept because --force was specified.';
      warn 'At this point your object will not save properly, though backup ' .
           'copies will be created.';
      print "Run 'load-state before-update' to undo the update and return to ",
            "a working state.\n" if $save_state;
    } else {
      warn 'Verification failed after the upgrade was complete.';
      print "$0 has been reverted to its pre-upgrade state.\n" if $save_state;
      print "If you want to upgrade and keep the failure state, then run ",
```

[7] Due to how `update-from` is structured, this step actually happens first.

```
36              "'update-from $target --force'." if $save_state;
37        return &{'load-state'}('before-update') if $save_state;
38      }
39    }
40  }
```

11.4.5 The `update` function

This new `update-from` function contains all of the logic to perform individual updates, but it still requires you to list the parent objects. There isn't any need to do this manually though, since if we look for `parent::` attributes we can get the list. That's what `update` does:

Listing 11.11 `snippets/update-function`

```
meta::function('update', <<'EOF');
&{'update-from'}(@_, grep s/^parent::/o, sort keys %data);
EOF
```

11.5 `clone` and `child`

As things stand creating children from an object is a bit cumbersome. We have to manually copy the object and then run `update-from` once to get the parent to work out, which seems like too much work. Let's take care of copying first by creating a `clone` function:

Listing 11.12 `snippets/clone-function`

```
meta::function('clone', <<'EOF');
open my $file, '>', $_[0];
print $file serialize();
close $file;
chmod 0700, $_[0];
EOF
```

Now you can clone an object as it exists at any given moment. More interesting, though, is the related `child` function, which creates an object already setup for inheritance:

Listing 11.13 `snippets/child-function`

```
meta::function('child', <<'EOF');
my ($child_name) = @_;
clone($child_name);
qx($child_name update-from $0 -n);
EOF
```

`object` implements this function; so, for example, you could inherit from it like this:

```
$ /path/to/object child ./foo
$ ./foo update -n
$ ./foo ls -a parent::
parent::/path/to/object
$
```

Chapter 12

Detecting divergence

The inheritance system built in the last chapter has a couple of problems. One is a subtle race condition that can happen when a script has grandparents and out-of-date parents. The other is the very real deficiency that objects can't have divergent attributes without those attributes then being overwritten on the next update.[1]

Neither of these problems is actually very difficult to solve. All we need to do is store not only the name of the attribute that we inherited, but also the state it was in when we last read it. We then make sure that the attribute we've got hasn't changed since we last read it before we update it. If it's in a different state, we assume that it was modified locally and preserve our local copy.

12.1 Attribute hashing

Right now, `parent::` attributes are stored as lists of attributes defined by that parent. What we really want is not only the names of those attributes, but their hashes as well. So, for example, we inherited `function::ls` from `object` and its hash was `01a23d51d5b529e03943bd57e33f92df`. If our copy isn't the same, then we preserve our local one because it's divergent.

`function::update-from` uses `ls` to get the attribute list that ends up making up the `parent::` attribute. The last implementation used `-aiu`; we'll add a new command-line option called `-h` that tells `ls` to include hashes. So the output would look like this:

```
$ ./object ls -ahiu
...
meta::type::meta       c6250056816b58a9608dd1b2614246f8
meta::type::parent     09d1d03379e4e0b262e06939f4e00464
meta::type::retriever  71a29050bf9f20f6c71afddff83addc9
```

[1] I ran into this when writing Caterwaul; its homepage is a self-modifying Perl file that needed a gray background instead of a white one when rendered as HTML. So I wrote divergence detection.

```
meta::type::state      84da7d5220471307f1f990c5057d3319
...
$
```

This is saying that `object`'s copy of `meta::type::meta` hashes to `c6250056816b58a9608dd1b2614246f8`, so if ours doesn't, then we have diverged. This divergence detection is taken care of by `update-from`, which still evaluates everything the parent script sends but restores divergent attribute values after the fact. (I considered doing it the right way and having the parent script not send divergent attributes, but this was more complicated than I wanted to implement.)

12.2 The code (lots of it)

In order to implement all of this stuff, we'll need to modify `ls` and `update-from`.

12.2.1 The new `ls`

Here's the monster `ls` that `object` defines:[2]

Listing 12.1 snippets/updated-ls

```
meta::function('ls', <<'EOF');
my ($options, @criteria) = separate_options(@_);
my ($external, $shadows, $sizes, $flags, $long, $hashes, $parent_hashes) =
    @$options{qw(-e -s -z -f -l -h -p)};

$sizes = $flags = $hashes = $parent_hashes = 1 if $long;

return table_display([grep ! exists $data{$externalized_functions{$_}},
    sort keys %externalized_functions]) if $shadows;

my $criteria    = join('|', @criteria);
my @definitions = select_keys('--criteria' => $criteria,
                              '--path' => $transient{path}, %$options);

my %inverses  = map {$externalized_functions{$_} => $_} keys %externalized_functions;
my @externals = map $inverses{$_}, grep length, @definitions;
my @internals = grep length $inverses{$_}, @definitions;
my @sizes     = map sprintf('%6d %6d', length(serialize_single($_)), length(retrieve($_))),
    @{$external ? \@internals : \@definitions} if $sizes;

my @flags     = map {my $k = $_; join '', map(is($k, "-$_") ? $_ : '-', qw(d i m u))}
    @definitions if $flags;

my @hashes    = map fast_hash(retrieve($_)), @definitions if $hashes;
```

[2] Don't worry if you don't get what's going on here. Most of it is just display logic.

```
25
26 my %inherited     = parent_attributes(grep /^parent::/o, keys %data) if $parent_hashes;
27 my @parent_hashes = map $inherited{$_} || '-', @definitions if $parent_hashes;
28
29 join "\n", map strip($_), split /\n/,
30     table_display($external ? [grep length, @externals] : [@definitions],
31                   $sizes ? ([@sizes]) : (), $flags ? ([@flags]) : (),
32                   $hashes ? ([@hashes]) : (), $parent_hashes ? ([@parent_hashes]) : ());
33 EOF
```

There isn't as much going on here as it looks like. Most of the logic is just figuring out what to display, as this `ls` implementation supports many different display modes. A few things I would like to point out, though, are:

1. On line 8, I reference a function called `table_display`. This just puts things into aligned columns; here's how `object` defines it:

Listing 12.2 snippets/table-display

```
1  meta::internal_function('table_display', <<'EOF');
2  # Displays an array of arrays as a table; that is, with alignment. Arrays are
3  # expected to be in column-major order.
4
5  sub maximum_length_in {
6    my $maximum = 0;
7    length > $maximum and $maximum = length for @_;
8    $maximum;
9  }
10
11 my @arrays    = @_;
12 my @lengths   = map maximum_length_in(@$_), @arrays;
13 my @row_major = map {my $i = $_; [map $$_[$i], @arrays]} 0 .. $#{$arrays[0]};
14 my $format    = join '  ', map "%-${_}s", @lengths;
15
16 join "\n", map strip(sprintf($format, @$_)), @row_major;
17 EOF
```

2. On line 13, I reference `$transient{path}`. This is a really useful shell feature that implicitly filters attributes when you type `ls`, but otherwise has nothing to do with what's going on here.

3. On line 21, I reference a function called `is()`. Remember the gnarly `grep` conditional in `select_keys`? That's now factored into `function::is`. Here's `object`'s implementation:

Listing 12.3 snippets/function-is

```
1  meta::function('is', <<'EOF');
2  my ($attribute, @criteria) = @_;
```

```
my ($options, @stuff) = separate_options(@criteria);
exists $data{$attribute} and attribute_is($attribute, %$options);
EOF
```

Listing 12.4 snippets/internal-function-attribute-is

```
meta::internal_function('attribute_is', <<'EOF');
my ($a, %options) = @_;
my %inherited = parent_attributes(grep /^parent::/o, sort keys %data) if
    grep exists $options{$_}, qw/-u -U -d -D/;

my $criteria = $options{'--criteria'} ||
               $options{'--namespace'} && "^$options{'--namespace'}::" || '.';

my %tests = ('-u' => sub {! $inherited{$a}},
             '-d' => sub {$inherited{$a} && fast_hash(retrieve($a)) ne $inherited{$a}},
             '-i' => sub {$transient{inherit}{namespace($a)}},
             '-s' => sub {$a =~ /^state::/o},
             '-m' => sub {$a =~ /^meta::/o});

# These checks make sure all tests are satisfied (a test is just a
# command-line option). I'm sure there's a clearer way to do it...
return 0 unless scalar keys %tests == scalar
    grep ! exists $options{$_}    ||   &{$tests{$_}}(), keys %tests;

return 0 unless scalar keys %tests == scalar
    grep ! exists $options{uc $_} || ! &{$tests{$_}}(), keys %tests;

$a =~ /$_/ || return 0 for @{$options{'--path'}};       # I'll explain this later
$a =~ /$criteria/;
EOF
```

12.2.2 The new `update-from`

Chapter 13

Virtual attributes

In the last chapter, `retrieve` was defined to grab either an attribute or a file. But there are a lot more things you might want to do with it, including creating views of existing data. This is exactly what virtual attributes are about.

13.1 Core implementation

Here's an updated `retrieve`:

Listing 13.1 `snippets/updated-retrieve`

```
meta::internal_function('retrieve', <<'EOF');
my @results = map defined $data{$_} ? $data{$_} : retrieve_with_hooks($_), @_;
wantarray ? @results : $results[0];
EOF
```

This is exactly the same as the one before except that the hard-coded `file::read` call has been replaced by a call to `retrieve_with_hooks`. This provides the indirection necessary to allow the user to introduce arbitrary retrievers. Before the user can do this, though, we'll need to make a new namespace:

Listing 13.2 `snippets/meta-retriever`

```
meta::meta('type::retriever', <<'EOF');
meta::configure 'retriever', extension => '.pl', inherit => 1;
meta::define_form 'retriever', sub {
  my ($name, $value) = @_;
  $transient{retrievers}{$name} = meta::eval_in("sub {\n$value\n}", "retriever::$name");
};
EOF
```

Nothing particularly interesting is going on here. We're just maintaining a name → implementation mapping in `%transient`. This mapping is then used

by `retrieve_with_hooks`, which asks `retriever::` functions for their values until one of them returns something besides `undef`:

Listing 13.3 `snippets/retrieve-with-hooks`

```
meta::internal_function('retrieve_with_hooks', <<'EOF');
# Uses the hooks defined in $transient{retrievers}, and returns undef if none work.
my ($attribute) = @_;
my $result      = undef;

defined($result = &$_($attribute)) and return $result for
    map $transient{retrievers}{$_}, sort keys %{$transient{retrievers}};

return undef;
EOF
```

13.2 Retrievers in `object`

`object` comes with four retrievers:

`retriever::file` Retrieves contents from a filename. This lets you say things like `cat foo`, where `foo` is a file. By extension, you can also copy files into named attributes using `function::cp`.

Listing 13.4 `snippets/retriever-file`

```
meta::retriever('file', <<'EOF');
-f $_[0] ? file::read($_[0]) : undef;
EOF
```

`retriever::id` Retrieves the string you are retrieving. This is primarily useful when you want a quick test value for something. For instance `cat id::foobar` will print `foobar`. Note that `id::` is not defined as a namespace, it's just a name pattern detected by `retriever::id`.

Listing 13.5 `snippets/retriever-id`

```
meta::retriever('id', <<'EOF');
$_[0] =~ /^id::/ ? substr($_[0], 4) : undef;
EOF
```

`retriever::object` This is a fun one. It lets you ask other Perl objects for their attributes. For example, `cat object::./foo::function::ls` fetches `function::ls` from `./foo`. If `./foo` is unreachable or doesn't contain `function::ls`, then `undef` is returned (which allows other retrievers to try to resolve your virtual attribute).

Listing 13.6 snippets/retriever-object

```
meta::retriever('object', <<'EOF');
# Fetch a property from another Perl object. This uses the 'cat' function.
return undef unless $_[0] =~ /^object::(.*?)::(.*)$/ && -x $1 && qx|$1 is '$2'|;
join '', qx|$1 cat '$2'|;
EOF
```

`retriever::perl` This returns the result of evaluating a given Perl expression. For example, `cat perl::3+4` prints 7.

Listing 13.7 snippets/retriever-perl

```
meta::retriever('perl', <<'EOF');
# Lets you use the result of evaluating some Perl expression
return undef unless $_[0] =~ /^perl::(.*)$/;
eval $1;
EOF
```

As you can see, there's not much involved in writing a retriever. The only particularly counterintuitive thing about it is returning `undef` instead of `0` or the empty string.

- Fixing up `edit` Right now, `function::edit` will happily let you edit a virtual attribute and then attempt to `associate` it, causing a failure. This is obviously lame; here's a better `edit` function that takes care of this:

Listing 13.8 snippets/updated-edit

```
meta::function('edit', <<'EOF');
my ($name, %options) = @_;
my $extension = extension_for($name);

die "$name is virtual or does not exist" unless exists $data{$name};

associate($name, invoke_editor_on($data{$name} // '', %options,
    attribute => $name, extension => $extension), execute => 1)});

save() unless $data{'data::edit::no-save'};
'';
EOF
```

- More interesting retrievers I'm writing this document in a self-modifying Perl file that includes a simple preprocessor.[1] In order to make this work, there are a bunch of attributes in the `section::` namespace, and each one includes some preprocessor directives to either include other attributes or to generate TEX constructs (in some cases both).

[1] Unsurprisingly, it's called `preprocessor` in the `perl-objects` repository.

The easiest way to render the document as TEX is to just have a virtual attribute I can retrieve that will contain the fully preprocessed source. Fortunately, `preprocessor` provides `retriever::pp` that does exactly this. Here's what that looks like:

Listing 13.9 `snippets/retriever-pp`

```
meta::retriever('pp', <<'EOF');
return undef unless namespace($_[0]) eq 'pp';
my $attr = retrieve(attribute($_[0]));
defined $attr ? preprocess($attr) : undef;
EOF
```

Notice that this retriever itself calls `retrieve`! This is totally allowed as long as you don't create an infinite loop. Here, we're trimming the `pp::` from the beginning of the attribute and are just retrieving the rest of it. This is good practice, as it lets you combine virtual retrievers. For example, you could preprocess an external file by saying `cat pp::my-file`.

Another interesting retriever is the `http` retriever defined in `repository`. This lets you retrieve arbitrary web content through Perl's LWP library:

Listing 13.10 `snippets/retriever-http`

```
meta::retriever('http', <<'EOF');
use LWP::Simple ();
return undef unless $_[0] =~ /^(?:http:)?\/\/(\w+.*)$/;
LWP::Simple::get("http://$1");
EOF
```

If you have this retriever, you can say things like `cat http://google.com` (or shorter, `cat //google.com`) and get back the content sent by the web server. This is especially useful when combined with the preprocessor, as it allows you to compile web-based resources into the final output.