X shell

Spencer Tipping

March 16, 2014

Contents

Part I

Language reference

Chapter 1

Similarities to TCL

Every xh value is a string. This includes functions, closures, lazy expressions, scope chains, call stacks, and heaps. Asserting string equivalence makes it possible to serialize any value losslessly, including a running xh process.[1]

Although the string equivalence is available, most operations have higher-level structure. For example, the `$` operator, which performs string interpolation, interpolates values in such a way that two things are true:

1. No interpolated value will be further interpolated (idempotence).
2. The interpolated value will be read as a single list element.

For example:

```
$ def bar bif
$ def foo "hi there \$bar!"
$ def baz $foo                      # no quoting necessary here by (2)
$ echo $baz
hi there $bar!                      # $bar unevaluated by (1)
$
```

This interpolation structure can be overridden by using one of three alternative forms of `$`:

```
$ def bar bif
$ def foo "hi there \$bar!"
$ echo $!foo                        # allow re-interpolation
hi there bif!
$ count [$foo]                      # single element
1
$ count [$@foo]                     # multiple elements
```

[1] Note that things like active socket connections and external processes will be proxied, however; xh can't migrate system-native things.

```
3
$ nth [$@!foo] 2                    # multiple and re-interpolation
bif!
$
```

All string values in xh programs are lifted into reader-safe quotations. This causes any "active" characters such as `$` to be prefixed with backslashes, a transformation you can mostly undo by using `$@!`. The only thing you can't undo is bracket balancing, which if undone would wreak havoc on your programs. You can see the effect of balancing by doing something like this:

```
$ def foo "[[[["
$ def bar [$@!foo]
$ echo $bar
[\[\[\[\[]
$
```

We can't get xh to create an unbalanced list through any series of rewriting operations, since the contract is that any active list characters are either positive and balanced, or escaped.

Chapter 2

Similarities to Lisp

xh is strongly based on the Lisp family of languages, most visibly in its homoiconicity. Any string wrapped in curly braces is a list of lines with the following contract:

```
$ def foo {bar bif
>          baz
>          bok quux}
$ count $foo
3
$ nth $foo 0
bar bif
$ nth $foo 2
bok quux
$ def foo {
>   bar bif
>   baz
>   bok quux
> }
$ count $foo
3
$ nth $foo 0
[bar bif]
$
```

Any string wrapped in `[]` or `()` is interpreted as a list of words, just as it is in Clojure. Also as in Lisp in general, `()` interpolates its result into the surrounding context:

```
$ def foo 'hi there'
$ echo $foo
hi there
$ echo (echo $foo)                  # similar to bash's $()
```

```
hi there
$
```

Any `()` list can be prefixed with `@` and/or `!` with effects analogous to `$`; e.g. `echo !@(echo hi there)`.

Chapter 3

Dissimilarities from everything else I know of

xh evaluates expressions outside-in:

1. Variable shadowing is not generally possible.
2. Expansion is idempotent for any set of bindings.
3. Unbound variables expand to active versions of themselves (a corollary of 2).
4. Laziness is implemented by referring to unbound quantities.
5. Bindings can be arbitrary list expressions, not just names (a partial corollary of 4).
6. No errors are ever thrown; all expressions that cannot be evaluated become `(error)` clauses that most functions consider to be opaque.
7. xh has no support for syntax macros.

Unbound names are treated as though they might at some point exist. For example:

```
$ echo $x
$x
$ def x $y
$ echo $x
$y
$ def y 10
$ echo $x
10
$
```

You can also bind expressions of things to express partial knowledge:

```
$ echo (count $str)
(count $str)
$ def (count $str) 10
$ echo $str
$str
$ echo (count $str)
10
$
```

This is the mechanism by which xh implements lazy evaluation, and it's also the reason you can serialize partially-computed lazy values.

Part II

Bootstrap implementation

Chapter 4

Self-replication

Listing 4.1 boot/xh-header

```
#!/usr/bin/env perl
BEGIN {
print STDERR q{
NOTE: Development image

If you see this note after installing the shell, it's probably because
you're running a version that has not yet rebuilt itself (maybe you got the
wrong file from the Git repo?). You can do this, but it will be really
slow and may use a lot of memory. There are two ways to fix this:

1. Download the standard image from http://spencertipping.com/xh
2. Have this image recompile itself by running xh.recompile-in-place (this
   will take some time because it stress-tests your Perl runtime)

Note also that bootstrapping requires Perl 5.14 or later, whereas running a
compiled image just requires Perl 5.10.

};
}

BEGIN {eval(our $xh_bootstrap = q{
# xh: the X shell | https://github.com/spencertipping/xh
# Copyright (C) 2014, Spencer Tipping
# Licensed under the terms of the MIT source code license

# For the benefit of HTML viewers (long story):
# <body style='display:none'>
# <script src='http://spencertipping.com/xh/page.js'></script>
use 5.014;
```

```
30 package xh;
31 our %modules;
32 our @module_ordering;
33
34 our %compilers = (pl => sub {
35   my $package = $_[0] =~ s/\./::/gr;
36   eval "{package ::$package;\n$_[1]\n}";
37   die "error compiling module $_[0]: $@" if $@;
38 });
39
40 sub defmodule {
41   my ($name, $code, @args) = @_;
42   chomp($modules{$name} = $code);
43   push @module_ordering, $name;
44   my ($base, $extension) = split /\.(\w+$)/, $name;
45   die "undefined module extension '$extension' for $name"
46     unless exists $compilers{$extension};
47   $compilers{$extension}->($base, $code, @args);
48 }
49
50 chomp($modules{bootstrap} = $::xh_bootstrap);
51 undef $::xh_bootstrap;
```

At this point we need a way to reproduce the image. Since the bootstrap code is already stored, we can just wrap it and each defined module into an appropriate `BEGIN` block.

Listing 4.2 boot/xh-header (continued)

```
1  sub image {
2    my @pieces = "#!/usr/bin/env perl";
3    push @pieces, "BEGIN {eval(our \$xh_bootstrap = <<'_')}",
4                  $modules{bootstrap},
5                  '_';
6    push @pieces, "BEGIN {xh::defmodule('$_', <<'_')}",
7                  $modules{$_},
8                  '_' for @module_ordering;
9    push @pieces, "xh::main::main;\n__DATA__";
10   join "\n", @pieces;
11 }
12 })}
```

Chapter 5

Perl-hosted evaluator

xh is self-hosting, but to get there we need to implement an interpreter in Perl. This interpreter is semantically correct but slow and shouldn't be used for anything besides bootstrapping the real compiler.

Listing 5.1 modules/interpreter.pl

```
BEGIN {xh::defmodule('xh::interpreter.pl', <<'_')}
# TODO
_
```