X shell

Spencer Tipping

March 22, 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 ho-moiconicity. 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.

Chapter 4

Functions

xh supports two equivalent ways to write function-like relations:

```
(def (foo $x) {echo hi there, $x!})
(foo spencer)
hi there, spencer!
()
```

This is named definition by destructuring, which works great for most cases. When you're writing an anonymous function, however, you'll need to describe the mappings individually:

```
(reduce {[$total +$x] (+ $total $x)
         [$total *$x] (* $total $x)} \
        0 \
        [+1 +2 *5 +1])
16
()
```

Part II

Bootstrap implementation

Chapter 5

Self-replication

Listing 5.1 boot/xh-header

```
 1  #!/usr/bin/env perl
 2  BEGIN {
 3  print STDERR q{
 4  NOTE: Development image
 5
 6  If you see this note after installing the shell, it's probably because
 7  you're running a version that has not yet rebuilt itself (maybe you got the
 8  wrong file from the Git repo?). You can do this, but it will be really
 9  slow and may use a lot of memory. There are two ways to fix this:
10
11  1. Download the standard image from http://spencertipping.com/xh
12  2. Have this image recompile itself by running xh.recompile-in-place (this
13     will take some time because it stress-tests your Perl runtime)
14
15  Note also that bootstrapping requires Perl 5.14 or later, whereas running a
16  compiled image just requires Perl 5.10.
17
18  };
19  }
20
21  BEGIN {eval(our $xh_bootstrap = q{
22  # xh: the X shell | https://github.com/spencertipping/xh
23  # Copyright (C) 2014, Spencer Tipping
24  # Licensed under the terms of the MIT source code license
25
26  # For the benefit of HTML viewers (long story):
27  # <body style='display:none'>
28  # <script src='http://spencertipping.com/xh/page.js'></script>
29  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 5.2 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  })}
```

11

Chapter 6

Perl-hosted evaluator

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

Listing 6.1 modules/interpreter.pl

```perl
1   BEGIN {xh::defmodule('xh::interpreter.pl', <<'_')}
2   use Memoize qw/memoize/;
3   use List::Util qw/max/;
4
5   sub active_regions {
6     # Returns a series of numbers that describes, in pre-order, regions of
7     # the given string that should be interpolated. The numeric list has the
8     # following format:
9     #
10    # (offset << 32 | len), (offset << 32 | len) ...
11
12    my @pieces = split /(\\.|\$@?!?\w+|\$@?!?\{[^}]+\}|@?!?\(|['])])/s, $_[0];
13    my $offset = 0;
14    my @result;
15    my @quote_offsets;
16
17    for (@pieces) {
18      if (@quote_offsets && substr($_[0], $quote_offsets[-1], 1) eq "'") {
19        # We're inside a hard-quote, so ignore everything except for the next
20        # hard-quote.
21        pop @quote_offsets if /^'/;
22      } else {
23        if (/^'/ || /^@?!?\(/) {
24          push @quote_offsets, $offset;
25        } elsif (/^\$/) {
26          push @result, $offset << 32 | length;
```

```perl
27          } elsif (/^\)/) {
28            my $start = pop @quote_offsets;
29            push @result, $start << 32 | $offset + 1 - $start;
30          }
31        }
32        $offset += length;
33      }
34
35      sort {$a <=> $b} @result;
36    }
37
38    memoize 'active_regions';
39
40    our %closers = ('(' => ')', '[' => ']', '{' => '}');
41    sub element_regions {
42      # Returns integer-encoded regions describing the positions of list
43      # elements. The list passed into this function should be unwrapped; that
44      # is, it should have no braces.
45      my ($xs)   = @_;
46      my $offset = 0;
47      my @pieces = split / ( "(?:\\.|[^"])*"
48                           | '(?:\\.|[^'])*'
49                           | \\.
50                           | [({\[\]})]
51                           | \s+ ) /xs, $_[0];
52      my @paren_offsets;
53      my @parens;
54      my @result;
55      my $item_start = -1;
56
57      for (@pieces) {
58        unless (@paren_offsets) {
59          if (/\s+/ || /^[)\]}]/) {
60            # End any item if we have one.
61            push @result, $item_start << 32 | $offset - $item_start
62            if $item_start >= 0;
63            $item_start = -1;
64          } else {
65            # Start an item unless we've already done so.
66            $item_start = $offset if $item_start < 0;
67          }
68        }
69
70        # Update bracket tracking.
71        if ($_ eq $closers{$parens[-1]}) {
72          if (@parens) {
```

13

```
73        pop @paren_offsets;
74        pop @parens;
75      } else {
76        die 'illegal closing brace: ... '
77          . substr($xs, max(0, $offset - 10), 20)
78          . ' ...'
79          . "\n(whole string is $xs)";
80      }
81    } elsif (/^[(\[{]/) {
82      push @paren_offsets, $offset;
83      push @parens, $_;
84    }
85
86    $offset += length;
87  }
88
89  push @result, $item_start << 32 | $offset if $item_start >= 0;
90  @result;
91 }
92
93 memoize 'element_regions';
94
95 sub xh_list_box {
96   $_[0] !~ /^[({\[]/ && element_regions(0, $_[0]) > 1
97     ? "[$_[0]]"
98     : $_[0];
99 }
100
101 sub xh_list_unbox {
102   return $1 if $_[0] =~ /^\[(.*)\]$/
103             || $_[0] =~ /^\((.*)\)$/
104             || $_[0] =~ /^\{(.*)\}$/;
105   $_[0];
106 }
107
108 sub parse_list {
109   my $unboxed = xh_list_unbox $_[0];
110   map xh_list_box(substr $unboxed, $_ >> 32, $_ & 0xffffffff),
111       element_regions 0, $unboxed;
112 }
113
114 sub into_list  {'(' . join(' ', map xh_list_box($_), @_) . ')'}
115 sub into_vec   {'[' . join(' ', map xh_list_box($_), @_) . ']'}
116 sub into_block {'{' . join(' ', map xh_list_box($_), @_) . '}'}
117
118 sub xh_vecp    {$_[0] =~ /^\[.*\]$/}
```

14

```perl
119  sub xh_listp  {$_[0] =~ /^\(.*\)$/}
120  sub xh_blockp {$_[0] =~ /^\{.*\}$/}
121  sub xh_varp    {$_[0] =~ /^\$/}
122
123  sub xh_count {
124    scalar element_regions 0, xh_list_unbox $_[0];
125  }
126
127  sub xh_nth {(parse_list $_[0])[$_[1]]}
128
129  sub xh_nth_eq {
130    # FIXME
131    my ($copy, $i, $v) = @_;
132    my @regions        = element_regions 0, $copy;
133    my $r              = $regions[$i];
134    substr($copy, $r >> 32, $r & 0xffffffff) = $v;
135    $copy;
136  }
137
138  sub destructuring_bind;
139  sub destructuring_bind {
140    # Both $pattern and $v should be quoted; that is, the string character [
141    # should be encoded as \[.
142    my ($pattern, $v) = @_;
143    my @pattern_elements = element_regions 0, $pattern;
144    my @v_elements       = element_regions 0, $v;
145    my %bindings;
146
147    # NOTE: no $@ matching
148    return undef unless @v_elements == @pattern_elements;
149
150    # NOTE: no foo$bar matching (partial constants)
151    for (my $i = 0; $i < @pattern_elements; ++$i) {
152      my $pi = xh_nth $pattern, $i;
153      my $vi = xh_nth $v,       $i;
154
155      return undef if $pi !~ /^\$/ && $pi ne $vi;
156
157      my @pattern_regions = element_regions 0, $pi;
158      my @v_regions       = element_regions 0, $vi;
159      return undef unless @pattern_regions == 1 && $pi =~ /^\$/
160                       || @pattern_regions == @v_regions;
161
162      if (xh_vecp $pi) {
163        my $sub_bind = destructuring_bind $pi, $vi;
164        return undef unless ref $sub_bind;
```

```
165        my %sub_bindings = %$sub_bind;
166        for (keys %sub_bindings) {
167          return undef if exists $bindings{$_}
168                       && $bindings{$_} ne $sub_bindings{$_};
169          $bindings{$_} = $sub_bindings{$_};
170        }
171      } elsif (xh_listp $pi) {
172        die "TODO: implement list binding for $pi";
173      } elsif ($pi =~ /^\$\{?(\w+)\}?$/) {
174        return undef if exists $bindings{$1} && $bindings{$1} ne $vi;
175        $bindings{$1} = $vi;
176      } elsif ($pi =~ /^\$/) {
177        die "illegal binding form: $pi";
178      } else {
179        return undef unless $pi eq $vi;
180      }
181    }
182
183    {%bindings};
184  }
185
186  sub invoke;
187  sub interpolate;
188  sub interpolate {
189    # Takes a string and a compiled binding hash and interpolates all
190    # applicable substrings outside-in. This process may involve full
191    # evaluation if () subexpressions are present, and is in general
192    # quadratic or worse in the length of the string.
193    my $bindings            = $_[0];
194    my @interpolation_regions = active_regions $_[1];
195    my @result_pieces;
196
197    for (@interpolation_regions) {
198      my $slice = substr $_[0], $_ >> 32, $_ & 0xffffffff;
199
200      # NOTE: no support for complex ${} expressions
201      if ($slice =~ /^\$(@?!?)\{?(\w+)\}?$/) {
202        # Expand a named variable that may or may not be defined yet.
203        push @result_pieces,
204            exists ${$bindings}{$2} ?
205                $1 eq ''  ? xh_listquote(xh_deactivate $bindings->{$2})
206              : $1 eq '@' ? xh_deactivate($bindings->{$2})
207              : $1 eq '!' ? xh_listquote($bindings->{$2})
208              :                 $bindings->{$2}
209            : "\$$slice";
210      } elsif ($slice =~ /^\((.*)\)$/s) {
```

```perl
211        push @result_pieces, invoke $bindings, parse_list interpolate $1;
212      } else {
213        push @result_pieces, $slice;
214      }
215    }
216
217    join '', @result_pieces;
218  }
219
220  sub xh_function_cases {
221    # FIXME
222    my @result;
223    my @so_far;
224    for (parse_vlist $_[0]) {
225      my ($command, @args) = parse_list $_;
226      if (xh_vecp $command) {
227        push @result, into_block @so_far if @so_far;
228        @so_far = ($command, into_list @args);
229      }
230    }
231    push @result, into_block @so_far if @so_far;
232    @result;
233  }
234
235  sub evaluate;
236  sub invoke {
237    # NOTE: no support for (foo bar $x)-style conditional destructuring;
238    # these are all rewritten into lambda forms
239    my ($bindings, $f, @args) = @_;
240    my $args = into_vec @args;
241
242    # Resolve f into a lambda form if it's still in word form.
243    $f = $bindings->{$f} if exists $bindings->{$f};
244
245    # Escape into perl
246    return $f->($bindings, @args) if ref $f eq 'CODE';
247
248    my %nested_bindings = %$bindings;
249    for (xh_function_cases $f) {
250      my ($formals, @body) = parse_block $_;
251      if (my $maybe_bindings = destructuring_bind $formals, $args) {
252        $nested_bindings{$_} = $$maybe_bindings{$_}
253        for keys %$maybe_bindings;
254        return evaluate {%nested_bindings}, into_block @body;
255      }
256    }
```

```perl
257
258     return into_list $f, @args;
259   }
260
261   sub evaluate {
262     my ($bindings, $block) = @_;
263     my @statements        = parse_block $block;
264     my $result;
265
266     # NOTE: this function updates $bindings in place.
267     for (@statements) {
268       # Each statement is an invocation, which for now we assume all to be
269       # functions.
270       #
271       # NOTE: this is semantically incomplete as we don't consider
272       # macro-bindings.
273       $result = invoke $bindings, parse_list interpolate $bindings, $_;
274     }
275     $result;
276   }
277   _
```