X shell

Spencer Tipping

March 20, 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.

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

```
#!/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 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 })}
```

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

```
BEGIN {xh::defmodule('xh::interpreter.pl', <<'_')}
use Memoize qw/memoize/;
use List::Util qw/max/;

sub active_regions {
  # Returns a series of numbers that describes, in pre-order, regions of
  # the given string that should be interpolated. The numeric list has the
  # following format:
  #
  # (offset << 32 | len), (offset << 32 | len) ...

  my @pieces = split /(\\.|\$@?!?\w+|\$@?!?\{[^}]+\}|@?!?\(|[')])/s, $_[0];
  my $offset = 0;
  my @result;
  my @quote_offsets;

  for (@pieces) {
    if (@quote_offsets && substr($_[0], $quote_offsets[-1], 1) eq "'") {
      # We're inside a hard-quote, so ignore everything except for the next
      # hard-quote.
      pop @quote_offsets if /^'/;
    } else {
      if (/^'/ || /^@?!?\(/) {
        push @quote_offsets, $offset;
      } elsif (/^\$/) {
        push @result, $offset << 32 | length;
```

```
      } elsif (/^\)/) {
        my $start = pop @quote_offsets;
        push @result, $start << 32 | $offset + 1 - $start;
      }
    }
    $offset += length;
  }

  sort {$a <=> $b} @result;
}

memoize 'active_regions';

our $whitespace = qr/\s+/;
our $newlines   = qr/\n(?:\s*\n)*/;
our %closers    = ('(' => ')', '[' => ']', '{' => '}');

sub element_regions {
  # Returns integer-encoded regions describing the positions of list
  # elements. The list passed into this function should be unwrapped; that
  # is, it should have no braces.
  my ($is_vertical, $xs) = @_;
  my $split_on           = $is_vertical ? $newlines : $whitespace;
  my $offset             = 0;
  my @pieces             = split / ( "(?:\\.|[^"])*"
                                   | '(?:\\.|[^'])*'
                                   | \\.
                                   | [({\[\]})]
                                   | $split_on ) /xs, $_[0];
  my @paren_offsets;
  my @parens;
  my @result;
  my $item_start = -1;

  for (@pieces) {
    unless (@paren_offsets) {
      if (/$split_on/ || /^[)\]}]/) {
        # End any item if we have one.
        push @result, $item_start << 32 | $offset - $item_start
        if $item_start >= 0;
        $item_start = -1;
      } else {
        # Start an item unless we've already done so.
        $item_start = $offset if $item_start < 0;
      }
    }
```

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

```
  my $unboxed = xh_list_unbox $_[0];
  map xh_list_box(substr $unboxed, $_ >> 32, $_ & 0xffffffff),
      element_regions 1, $unboxed;
}

sub into_list  {'(' . join(' ',  map xh_list_box($_), @_) . ')'}
sub into_vec   {'[' . join(' ',  map xh_list_box($_), @_) . ']'}
sub into_block {'{' . join("\n",                      @_) . '}'}

sub xh_vecp   {$_[0] =~ /^\[.*\]$/}
sub xh_listp  {$_[0] =~ /^\(.*\)$/}
sub xh_blockp {$_[0] =~ /^\{.*\}$/}
sub xh_varp   {$_[0] =~ /^\$/}

sub xh_count {
  scalar element_regions 0, xh_list_unbox $_[0];
}

sub xh_nth {(parse_list $_[0])[$_[1]]}

sub xh_nth_eq {
  # FIXME
  my ($copy, $i, $v) = @_;
  my @regions        = element_regions 0, $copy;
  my $r              = $regions[$i];
  substr($copy, $r >> 32, $r & 0xffffffff) = $v;
  $copy;
}

sub xh_vcount {
  scalar element_regions 1, xh_list_unbox $_[0];
}

sub xh_vnth {
  my @regions = element_regions 1, $_[0];
  my $r       = $regions[$_[1]];
  xh_list_box substr $_[0], $r >> 32, $r & 0xffffffff;
}

sub xh_vnth_eq {
  my ($copy, $i, $v) = @_;
  my @regions        = element_regions 1, $copy;
  my $r              = $regions[$i];
  substr($copy, $r >> 32, $r & 0xffffffff) = $v;
  $copy;
}
```

```
165
166 sub destructuring_bind;
167 sub destructuring_bind {
168   # Both $pattern and $v should be quoted; that is, the string character [
169   # should be encoded as \[.
170   my ($pattern, $v) = @_;
171   my @pattern_elements = element_regions 0, $pattern;
172   my @v_elements       = element_regions 0, $v;
173   my %bindings;
174
175   # NOTE: no $@ matching
176   return undef unless @v_elements == @pattern_elements;
177
178   # NOTE: no foo$bar matching (partial constants)
179   for (my $i = 0; $i < @pattern_elements; ++$i) {
180     my $pi = xh_nth $pattern, $i;
181     my $vi = xh_nth $v,       $i;
182
183     return undef if $pi !~ /^\$/ && $pi ne $vi;
184
185     my @pattern_regions = element_regions 0, $pi;
186     my @v_regions       = element_regions 0, $vi;
187     return undef unless @pattern_regions == 1 && $pi =~ /^\$/
188                      || @pattern_regions == @v_regions;
189
190     if (xh_vecp $pi) {
191       my $sub_bind = destructuring_bind $pi, $vi;
192       return undef unless ref $sub_bind;
193       my %sub_bindings = %$sub_bind;
194       for (keys %sub_bindings) {
195         return undef if exists $bindings{$_}
196                      && $bindings{$_} ne $sub_bindings{$_};
197         $bindings{$_} = $sub_bindings{$_};
198       }
199     } elsif (xh_listp $pi) {
200       die "TODO: implement list binding for $pi";
201     } elsif ($pi =~ /^\$\{?(\w+)\}?$/) {
202       return undef if exists $bindings{$1} && $bindings{$1} ne $vi;
203       $bindings{$1} = $vi;
204     } elsif ($pi =~ /^\$/) {
205       die "illegal binding form: $pi";
206     } else {
207       return undef unless $pi eq $vi;
208     }
209   }
210
```

```
  {%bindings};
}

sub invoke;
sub interpolate;
sub interpolate {
  # Takes a string and a compiled binding hash and interpolates all
  # applicable substrings outside-in. This process may involve full
  # evaluation if () subexpressions are present, and is in general
  # quadratic or worse in the length of the string.
  my $bindings              = $_[0];
  my @interpolation_regions = active_regions $_[1];
  my @result_pieces;

  for (@interpolation_regions) {
    my $slice = substr $_[0], $_ >> 32, $_ & 0xffffffff;

    # NOTE: no support for complex ${} expressions
    if ($slice =~ /^\$(@?!?)\{?(\w+)\}?$/) {
      # Expand a named variable that may or may not be defined yet.
      push @result_pieces,
           exists ${$bindings}{$2} ?
               $1 eq ''  ? xh_listquote(xh_deactivate $bindings->{$2})
             : $1 eq '@' ? xh_deactivate($bindings->{$2})
             : $1 eq '!' ? xh_listquote($bindings->{$2})
             :             $bindings->{$2}
           : "\$$slice";
    } elsif ($slice =~ /^\((.*)\)$/s) {
      push @result_pieces, invoke $bindings, parse_list interpolate $1;
    } else {
      push @result_pieces, $slice;
    }
  }

  join '', @result_pieces;
}

sub xh_function_cases {
  my @result;
  my @so_far;
  for (parse_vlist $_[0]) {
    my ($command, @args) = parse_list $_;
    if (xh_vecp $command) {
      push @result, into_block @so_far if @so_far;
      @so_far = ($command, into_list @args);
    }
```

```
257   }
258   push @result, into_block @so_far if @so_far;
259   @result;
260 }
261
262 sub evaluate;
263 sub invoke {
264   # NOTE: no support for (foo bar $x)-style conditional destructuring;
265   # these are all rewritten into lambda forms
266   my ($bindings, $f, @args) = @_;
267   my $args = into_vec @args;
268
269   # Resolve f into a lambda form if it's still in word form.
270   $f = $bindings->{$f} if exists $bindings->{$f};
271
272   # Escape into perl
273   return $f->($bindings, @args) if ref $f eq 'CODE';
274
275   my %nested_bindings = %$bindings;
276   for (xh_function_cases $f) {
277     my ($formals, @body) = parse_block $_;
278     if (my $maybe_bindings = destructuring_bind $formals, $args) {
279       $nested_bindings{$_} = $$maybe_bindings{$_}
280       for keys %$maybe_bindings;
281       return evaluate {%nested_bindings}, into_block @body;
282     }
283   }
284
285   return into_list $f, @args;
286 }
287
288 sub evaluate {
289   my ($bindings, $block) = @_;
290   my @statements         = parse_block $block;
291   my $result;
292
293   # NOTE: this function updates $bindings in place.
294   for (@statements) {
295     # Each statement is an invocation, which for now we assume all to be
296     # functions.
297     #
298     # NOTE: this is semantically incomplete as we don't consider
299     # macro-bindings.
300     $result = invoke $bindings, parse_list interpolate $bindings, $_;
301   }
302   $result;
```

```
303 }
304 _
```