xh

Spencer Tipping

March 23, 2014

Contents

Part I

Language reference

Chapter 1

Similarities to TCL

Every xh value is a string. This includes lists, 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)
(identity $baz)
"hi there \$bar!"
(echo $baz)
hi there $bar!                      # $bar unevaluated by (1)
(identity $foo)
"hi there \$bar!"                   # $foo == $baz, of course
(echo $foo)
hi there $bar!
()
```

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

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

```
(def bar bif)
(def foo "hi there \$bar!")
(echo $!foo)                          # allow re-interpolation
hi there bif!
(count [$foo])                        # single element
1
(count [$@foo])                       # multiple elements
3
(nth [$@!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)
["[[ [["]
()
```

Chapter 2

Similarities to Lisp

xh is strongly based on the Lisp family of languages, most visibly in its homoiconicity. 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

Like Haskell, 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 (fn [$total +$x] (+ $total $x)
            [$total *$x] (* $total $x)) \
        0 \
        [+1 +2 *5 +1])
16
()
```

4.1 Type hints

The string form of a value conveys its type. xh syntax supports the following structures:

```
[x y z ...]                     # array/vector
{x y z ...}                     # map
(x y z ...)                     # interpolated (active!) list
$x                              # interpolated (active!) variable
bareword                        # string with interpolation
-42.0                           # string with interpolation
"..."                           # string with interpolation
'...'                           # string with no interpolation
\x                              # single-character string, no interp
```

When you ask xh about the type of a value, xh looks at the first byte and figures it out.[1] Because of this, not all strings are convertible to values despite all values being convertible to strings. You can easily convert between types by interpolating:

```
(def list-form [1 2 3 4])
(def string-form "$@list-form")
(identity $list-form)
[1 2 3 4]
(identity $string-form)
"1 2 3 4"
(def map-form {$@list-form})
(identity $map-form)
{1 2 3 4}
()
```

Therefore the meaning of `$@x` could be interpreted as, "the untyped version of `x`," and `$!@x` could be, "eval the untyped version of `x`."

I bring this up here because most modern languages provide some facility for multimethods (e.g. OOP). In xh you do this by writing partial relations and destructuring:

```
(def (custom-count [$@xs]) (count $xs))
(def (custom-count {$@xs}) (%m (count $xs) / 2))
```

4.2 Laziness and localization

xh is a distributed runtime with serializable lazy values, which is a potential problem if you want to avoid proxying all over the place. Fortunately, a more elegant solution exists in most cases. Rather than using POSIX calls directly, xh programs access system resources like files through a slight indirection:

```
(def some-bytes (subs /etc/passwd 0 4096))
(echo $some-bytes)                # $some-bytes is lazy
```

This is clearly trivial if the `def` and `echo` execute on the same machine. But the `echo` can also be moved trivially by adding a hostname component to the file:

```
(def some-bytes (subs /etc/passwd 0 4096))
(identity $some-bytes)
(subs @host1/etc/passwd 0 4096)
```

This `@host1` namespace allows any remote xh runtime to negotiate with the original host, making lazy values fully mobile (albeit possibly slower).

[1] Note that xh is in no way required to represent these values as strings internally. It just lies so convincingly that you would never know the difference.

4.3 Argument evaluation

Functions are always defined using destructuring. So even trivial functions like `(def (f $x) ...)` are interpreted by xh as patterns. Technically, a pattern is a reverse expansion; the rule is that if you're pattern-matching against something, expanding the filled-in values should produce the original expression, possibly modulo string differences.

I mention this in such detail because it impacts how lazy arguments work. Suppose you have this:

```
(def (f $x) (count $x))          # pattern is strict
(f $nonexistent)                 # $nonexistent is lazy
```

When you try to evaluate `(f $nonexistent)`, nothing happens; this expression isn't expanded into `(count $nonexistent)` because *f's definition doesn't allow for lazy variables*. Remember 1 from way earlier: because the pattern for `f` was written using a regular `$` for interpolation, no value of `x` could have resulted in `$x` generating a lazy value.[2]

[2]TODO: is this remotely true? This seems like it totally kills lazy evaluation in general. Really think carefully about this before committing to it.

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 %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 ($xs)   = @_;
  my $offset = 0;
  my @pieces = split / ( "(?:\\.|[^"\\])*"
                       | '(?:\\.|[^'\\])*'
                       | \\.
                       | [({\[\]})]
                       | \s+ ) /xs, $_[0];
  my @parens;
  my @result;
  my $item_start = -1;

  for (@pieces) {
    unless (@parens) {
      if (/^\s+/ || /^[)\]}]/) {
        # 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;
      }
    }

    # Update bracket tracking.
    if ($_ eq $closers{$parens[-1]}) {
      if (@parens) {
        pop @parens;
```

```
      } else {
        die 'illegal closing brace: ... '
          . substr($xs, max(0, $offset - 10), 20)
          . ' ...'
          . "\n(whole string is $xs)";
      }
    } elsif (/^[(\[{]/) {
      push @parens, $_;
    }

    $offset += length;
  }

  push @result, $item_start << 32 | $offset if $item_start >= 0;
  @result;
}

memoize 'element_regions';

sub xh_list_unbox {
  return $1 if $_[0] =~ /^\[(.*)\]$/
            || $_[0] =~ /^\((.*)\)$/
            || $_[0] =~ /^\{(.*)\}$/;
  $_[0];
}

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

sub into_list {'(' . join(' ', map xh_list_box($_), @_) . ')'}
sub into_vec  {'[' . join(' ', map xh_list_box($_), @_) . ']'}
sub into_map  {'{' . join(' ', map xh_list_box($_), @_) . '}'}

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]]}
```

```
119
120 sub xh_nth_eq {
121   my (undef, $i, $v) = @_;
122   my $unboxed        = xh_list_unbox $_[0];
123   my @regions        = element_regions 0, $unboxed;
124   my $r              = $regions[$i];
125
126   substr($_[0], 0, 1 + ($r >> 32)) . $v .
127   substr($_[0], 1 + ($r >> 32) + ($r & 0xffffffff));
128 }
129 _
```