xh

Spencer Tipping

December 20, 2014

Contents

Part I

design

Chapter 1

constraints

xh is designed to be a powerful and ergonomic interface to multiple systems, many of which are remote. As such, it's subject to programming language, shell, and distributed-systems constraints:

1. *realprog* xh will be used for real programming. (*initial assumption*)
2. *shell* xh will be used as a shell. (*initial assumption*)
3. *distributed* xh will be used to manage any machine on which you have a login, which could be hundreds or thousands. (*initial assumption*)
4. *noroot* You will not always have root access to machines you want to use, and they may have different architectures. (*initial assumption*)
5. *ergonomic* xh should approach the limit of ergonomic efficiency as it learns more about you. (*initial assumption*)
6. *security* xh should never compromise your security, provided you understand what it's doing. (*initial assumption*)
7. *webserver* It should be possible to write a "hello world" HTTP server on one line.
    - *initial assumption*
    - *realprog* 1
    - *shell* 2
8. *liveprev* It should be possible to preview the evaluation of any well-formed expression without causing side-effects.
    - *initial assumption*
    - *shell* 2
    - *ergonomic* 5

- *nodebug* 11

9. *notslow* xh should never cause an unresolvable performance problem that could be worked around by using a different language.

   - *initial assumption*
   - *realprog* 1
   - *ergonomic* 5

10. *unreliable* Connections between machines may die at any time, and remain down for arbitrarily long. xh must never become unresponsive when this happens, and any data coming from those machines should block until it is available again (i.e. xh's behavior should be invariant with connection failures).

    - *initial assumption*
    - *realprog* 1
    - *shell* 2
    - *distributed* 3

11. *nodebug* Debugging should require little or no effort; all error cases should be trivially obvious.

    - *initial assumption*
    - *realprog* 1
    - *distributed* 3
    - *ergonomic* 5

12. *database* An xh instance should trivially function as a database; there should be no distinction between data in memory and data on disk.

    - *initial assumption*
    - *realprog* 1
    - *ergonomic* 5
    - *nodebug* 11
    - *no-oome* 19
    - *notslow* 9

13. *prediction* xh should use every keystroke to build/refine a model it uses to predict future keystrokes and commands. (*ergonomic* 5)

14. *history* The likelihood that xh forgets anything from your command history should be inversely proportional to the amount of effort required to retype/recreate it. (*ergonomic* 5, *prediction* 13)

15. *anonymous* xh must provide a way to accept input and execute commands without updating its prediction model. (*security* 6)

16. *pastebin* xh should be able to submit an encrypted version of its current state to HTTP services like Github gists or pastebin.

    - *ergonomic* 5
    - *security* 6
    - *unreliable* 10
    - *selfinstall* 52
    - *wwwinit* 53

17. *likeshell* xh-script needs to feel like a regular shell for most purposes. (*shell* 2)

18. *imperative* xh-script should be fundamentally imperative.

    - *realprog* 1
    - *shell* 2
    - *likeshell* 17

19. *no-oome* xh must never run out of memory or swap pages to disk, regardless of what you tell it to do.

    - *realprog* 1
    - *shell* 2
    - *notslow* 9
    - *ergonomic* 5

20. *nonblock* xh must respond to every keystroke within 20ms; therefore, SSH must be used only for nonblocking RPC requests (i.e. the shell always runs locally).

    - *shell* 2
    - *notslow* 9
    - *ergonomic* 5

21. *remotestuff* All resources, local and remote, must be uniformly accessible; i.e. autocomplete, filename substitution, etc, must all just work (up to random access, which is impossible without FUSE or similar).

    - *shell* 2
    - *distributed* 3
    - *ergonomic* 5

22. *prefix* xh-script uses prefix notation. (*shell* 2)

23. *quasiquote* xh-script quasiquotes values by default. (*shell* 2)

24. *unquote* xh-script defines an unquote operator. (*shell* 2, *quasiquote* 23)

25. *datastruct* The xh runtime provides real, garbage-collected data structures. (*realprog* 1)

26. *quotestruct* Every xh data structure has a quoted form.

    - *datastruct* 25
    - *shell* 2
    - *nodebug* 11
    - *liveprev* 8

27. *printstruct* Every xh data structure can be losslessly serialized by quoting it. In addition, every type of list can be losslessly serialized by coercing it to a string; the result can be unquoted to coerce it back to its original form.

    - *shell* 2
    - *distributed* 3
    - *database* 12
    - *quotestruct* 26
    - *varsinrc* 55
    - *imagemerging* 58

28. *immutable* Data structures have no identity and therefore are immutable. By extension, circular references can't be created except by indirection through a mutable value. (*distributed* 3, *printstruct* 27)

29. *opaques* xh-script must have access to machine-specific opaque resources like PIDs and file handles. (*realprog* 1, *shell* 2)

30. *mutablesyms* Each xh instance should implement a mutable symbol table with weak reference support, subject to safe distributed garbage collection.

    - *immutable* 28
    - *opaques* 29
    - *no-oome* 19
    - *heap* 46

31. *stateown* Every piece of mutable state, including symbol tables, must have at most one authoritative copy (mutable state ownership within xh is managed by a CP system, and the state itself is trivially CP).

    - *unreliable* 10

- *opaques* 29
- *mutablesyms* 30
- *threadmobility* 49

32. *checkpoint* An xh instance should be able to save checkpoints of itself in case of failure. If you do this, xh becomes an AP system. (*unreliable* 10, *stateown* 31)

33. *lazy* xh's evaluator must support some kind of laziness.

    - *realprog* 1
    - *no-oome* 19
    - *remotestuff* 21
    - *notslow* 9

34. *printlazy* Lazy values must have well-defined quoted forms and be losslessly serializable.

    - *quotestruct* 26
    - *printstruct* 27
    - *lazy* 33
    - *threadmobility* 49
    - *heap* 46

35. *introspectlazy* All lazy values can be subject to introspection to identify why they haven't been realized. This introspection must fully encode xh's knowledge about a value, modulo outstanding IO or CPU requests.

    - *nodebug* 11
    - *notslow* 9
    - *unreliable* 10
    - *nonblock* 20
    - *lazy* 33
    - *threadscheduler* 48

36. *abstract* xh must be able to partially evaluate expressions that contain unknown quantities.

    - *liveprev* 8
    - *lazy* 33
    - *introspectlazy* 35
    - *printlazy* 34

37. *code=data* xh-script code should be a reasonable data storage format. (*shell* 2, *abstract* 36)

38. *selfparse* xh-script must contain a library to parse itself. (*code=data* 37)

39. *homoiconic* xh-script must be homoiconic.

    - *code=data* 37
    - *selfparse* 38
    - *selfhost* 43
    - *abstractstruct* 45

40. *xh2c* xh should be able to compile any function to C, compile it if the host has a C compiler, and transparently migrate execution into this process.

    - *realprog* 1
    - *threadmobility* 49
    - *notslow* 9

41. *xh2perl* xh should be able to compile any function to Perl rather than interpreting its execution.

    - *realprog* 1
    - *noroot* 4
    - *notslow* 9

42. *xh2js* xh should be able to compile any function to Javascript so that browser sessions can transparently become computing nodes.

    - *realprog* 1
    - *distributed* 3
    - *notslow* 9

43. *selfhost* xh should follow a bootstrapped self-hosting runtime model.

    - *xh2c* 40
    - *xh2perl* 41
    - *xh2js* 42
    - *abstractstruct* 45

44. *dynamiccompiler* xh-script should be executed by a profiling/tracing dynamic compiler that automatically compiles certain pieces of code to alternative forms like Perl or C. (*notslow* 9)

45. *abstractstruct* The xh compiler should optimize data structure representations for the backend being targeted.

- *notslow* 9
- *threadmobility* 49
- *dynamiccompiler* 44

46. *heap* xh needs to implement its own heap and memory manager, and swap values to disk without blocking.

    - *realprog* 1
    - *no-oome* 19
    - *database* 12
    - *inperl* 56

47. *threading* xh should implement its own threading model to accommodate blocked IO requests.

    - *shell* 2
    - *distributed* 3
    - *webserver* 7
    - *lazy* 33
    - *heap* 46

48. *threadscheduler* xh threads should be subject to scheduling that reflects the user's priorities.

    - *shell* 2
    - *distributed* 3
    - *lazy* 33
    - *threading* 47

49. *threadmobility* Running threads must be transparently portable between machines and compiled backends.

    - *distributed* 3
    - *threading* 47
    - *dynamiccompiler* 44
    - *abstractstruct* 45
    - *threadscheduler* 48

50. *refaffinity* All machine-specific references must encode the machine for which they are defined. (*opaques* 29, *threadmobility* 49)

51. *uniqueid* Every xh instance must have a unique ID, ideally one that can be typed easily. (*ergonomic* 5, *refaffinity* 50)

52. *selfinstall* xh needs to be able to self-install on remote machines with no intervention (assuming you have a passwordless SSH connection). (*distributed* 3, *noroot* 4)

53. *wwwinit* You should be able to upload your xh image to a website and then install it with a command like this: `curl me.com/xh | perl`. (*distributed* 3, *noroot* 4)

54. *selfmodifying* Your settings should be present as soon as you download your image, so the image must be self-modifying and contain your settings.

    - *distributed* 3
    - *ergonomic* 5
    - *prediction* 13
    - *selfinstall* 52
    - *wwwinit* 53

55. *varsinrc* Your settings should be able to contain any value you can create from the REPL (with the caveat that some are defined only with respect to a specific machine).

    - *realprog* 1
    - *shell* 2
    - *ergonomic* 5
    - *datastruct* 25
    - *wwwinit* 53

56. *inperl* xh should probably be written in Perl 5.

    - *distributed* 3
    - *noroot* 4
    - *selfinstall* 52
    - *wwwinit* 53
    - *selfmodifying* 54

57. *perlcoreonly* xh can't have any dependencies on CPAN modules, or anything else that isn't in the core library.

    - *distributed* 3
    - *noroot* 4
    - *selfinstall* 52

58. *imagemerging* It should be possible to address variables defined within xh images (as files or network locations). (*selfmodifying* 54, *varsinrc* 55)

59. *sshrpc* xh's RPC protocol must work via stdin/out communication over an SSH channel to a remote instance of itself.

    - *distributed* 3
    - *security* 6
    - *selfinstall* 52
    - *nonblock* 20
    - *remotestuff* 21

60. *rpcmulti* xh's RPC protocol must support request multiplexing.

    - *distributed* 3
    - *notslow* 9
    - *nonblock* 20
    - *remotestuff* 21
    - *lazy* 33
    - *sshrpc* 59

61. *hostswitch* Two xh servers on the same host should automatically connect to each other. This allows a server-only machine to act as a VPN.

    - *distributed* 3
    - *noroot* 4
    - *sshrpc* 59
    - *transitive* 63

62. *domainsockets* xh should create a UNIX domain socket to listen for other same-machine instances. (*security* 6, *hostswitch* 61)

63. *transitive* xh's network topology should forward requests transitively.

    - *distributed* 3
    - *noroot* 4
    - *sshrpc* 59

64. *routing* xh should implement a network optimizer that responds to observations it makes about latency and throughput.

    - *notslow* 9
    - *sshrpc* 59
    - *transitive* 63

Chapter 2

xh-script

These constraints are based on the ones in chapter 1.

1. *xhs.datatypes* xh has two fundamental data types, lists and strings. (*initial assumption*)
2. *xhs.listtypes* Lists have three types, `list`, `array`, and `map`, corresponding to `()`, `[]`, and `{}`, respectively. (*initial assumption*, *xhs.datatypes* 1)
3. *xhs.eval-identities* Evaluation of any expression may happen at any time; the only scheduling constraint is the realization of lazy expressions, whose status is visible by looking at their quoted forms. Therefore, the evaluator is, to some degree, associative, commutative, and idempotent.
   - *initial assumption*
   - *distributed* 3 above
   - *nodebug* 11 above
   - *liveprev* 8 above
   - *nonblock* 20 above
   - *lazy* 33 above
   - *introspectlazy* 35 above
   - *abstract* 36 above
4. *xhs.relational* Relational evaluation is possible by using `amb`, which returns any of the given presumably-equivalent values. xh-script is relational and invertible, though inversion is not always lossless and may produce perpetually-unresolved unknowns representing degrees of freedom.
   - *initial assumption*
   - *nodebug* 11 above
   - *lazy* 33 above

- *introspectlazy* 35 above
- *abstract* 36 above
- *selfhost* 43 above
- *abstractstruct* 45 above
- *threadscheduler* 48 above
- *xhs.eval-identities* 3

5. *xhs.bestfirst* Due to functions like `amb`, evaluation proceeds as a best-first search through the space of values. You can influence this search by defining the `abstraction` relation for a particular class of expressions. (*notslow* 9 above, *xhs.relational* 4)

6. *xhs.nocut* Unlike Prolog, xh defines no cut primitive. You should use `abstraction` to locally grade the search space instead.

   - *nodebug* 11 above
   - *xhs.eval-identities* 3
   - *xhs.bestfirst* 5

7. *xhs.unquote-structure* Unquoting is structure-preserving with respect to parsing; that is, it will never force a reparse if its argument has already been parsed.

   - *initial assumption*
   - *realprog* 1 above
   - *unquote* 24 above
   - *notslow* 9 above
   - *abstract* 36 above

8. *xhs.stackscope* All scoping is done by passing a second argument to `unquote`; this enables variable resolution during the unquoting operation.

   - *initial assumption*
   - *unquote* 24 above
   - *mutablesyms* 30 above
   - *xhs.eval-identities* 3

9. *xhs.noshadow* Variable shadowing is impossible. (*xhs.eval-identities* 3, *xhs.stackscope* 8)

10. *xhs.unquote-parse* Unquoting and structural parsing are orthogonal operations provided by `unquote` and `read`, respectively.

    - *quotestruct* 26 above
    - *introspectlazy* 35 above

- *xhs.eval-identities* 3
- *xhs.unquote-structure* 7

11. *xhs.runtimereceiver* Whether via RPC or locally, statements issued to an xh runtime can be interpreted as messages being sent to a receiver; the reply is sent along whatever continuation is specified. The runtime doesn't differentiate between local and remote requests, including those made by functions.

    - *imperative* 18 above
    - *threading* 47 above
    - *threadmobility* 49 above
    - *stateown* 31 above

12. *xhs.namespaces* Functions and variables exist in separate namespaces.

    - *likeshell* 17 above
    - *unquote* 24 above
    - *xhs.stackscope* 8
    - *xhs.runtimereceiver* 11

13. *xhs.fnliterals* Function literals are self-invoking when used as messages. (*xhs.namespaces* 12)

14. *nocallcc* Continuations are simulated in terms of lazy evaluation, but are never first-class.

    - *dynamiccompiler* 44 above
    - *introspectlazy* 35 above
    - *abstract* 36 above
    - *xhs.runtimereceiver* 11

15. *xhs.transientdefs* Some definitions are "transient," in which case they are used to resolve blocked lazy values but then may be discarded at any point.

    - *distributed* 3 above
    - *no-oome* 19 above
    - *lazy* 33 above
    - *xhs.runtimereceiver* 11

16. *xhs.globaldefs* Global definitions can apply to values at any time, and to values on different machines (i.e. their existence is broadcast). (*lazy* 33 above, *xhs.transientdefs* 15)

17. *xhs.nomacros* Syntactic macros cannot exist because invocation commutes with expansion, but functions may operate on terms whose values are undefined. (*xhs.eval-identities* 3, *xhs.unquote-structure* 7)

18. *xhs.noerrors* Errors cannot exist, but are represented by lazy values that contain undefined quantities that will never be realized. These undefined quantities are the unevaluated backtraces to the error-causing subexpressions.

    - *nodebug* 11 above
    - *lazy* 33 above
    - *abstract* 36 above
    - *xhs.eval-identities* 3

19. *xhs.destructuring* Any value can be used as a destructuring bind pattern. (*initial assumption*, *xhs.relational* 4)

20. *xhs.ambdestructure* `(amb)` can be used to destructure values, and it behaves as a disjunction.

    - *initial assumption*
    - *xhs.relational* 4
    - *xhs.destructuring* 19

21. *xhs.dof* Degrees of freedom within an inversion are represented by abstract values that will prevent the result from being realized. They are visible as unevaluated expressions within the quoted form, usually taking the form of calls to `(amb)`.

    - *initial assumption*
    - *xhs.relational* 4
    - *xhs.ambdestructure* 20

22. *xhs.nosynbareword* Barewords are considered unique symbols; that is, xh will never try to synthesize a bareword using string-style semantics. (*initial assumption*)

23. *xhs.se-axioms* Side effects and axioms are the same thing in xh. Once it commits to a side-effect, it must always assume that it happened (since it did). In particular, this means that imperative forms like `(def)` are actually ways to assume new ground truths.

    - *initial assumption*
    - *imperative* 18
    - *xhs.relational* 4
    - *xhs.globaldefs* 16

24. *xhs.virtualization* Every side effect can be replaced by a temporary assumption that models the effect. If you do this, you're replacing an axiom with a hypothesis. (*initial assumption*, *xhs.se-axioms* 23)

25. *xhs.amb-se* `(amb)` hypothesizes all side effects until you commit to a branch using `(def)`.

26. *xhs.mapsasrelations* Maps and relations are isomorphic, which means that `(def)` is a stateful form of `(assoc)`, and that map literals can be used as functions. (*initial assumption*)

27. *xhs.stablevalues* Maps, arrays, and unquoted atoms are stable under unquoting (e.g. there is no distributive property that would unquote individual values within these structures). (*initial assumption*, *xhs.mapsasrelations* 26)

Chapter 3

xh-script syntax

Design constraints for the syntax in particular.

1. *syn.reversibleparsing* The parser for xh is losslessly reversible: comment data, whitespace, and any other aspect of valid xh code is encoded in the parsed representation. (*initial assumption*)

2. *syn.tags* Lists, vectors, and maps can each be tagged by immediately prefixing the opening brace with a word. (*initial assumption*)

3. *syn.splice* A quoted form prefixed with `@` causes list splicing to occur, just like Common Lisp's `,@` and Clojure's `~@`. Technically `@` is a distributive, right-associative prefix expansion operator (sort of like `$` in some ways), so you can layer it to expand multiple layers of lists. Any non-lists are treated as lists of a single item; `@` is well-defined for all values.

    - *initial assumption*
    - *realprog* 1
    - *ergonomic* 5

4. *syn.escaping* Any character can be prefixed with `\` to cause it to be interpreted as a string. The only exception is that some escape sequences are interpreted, including `\n`, `\t`, and similar. (*initial assumption*, *likeshell* 17)

5. *syn.hashcomments* Comments begin with `#` preceded either by whitespace or the beginning of a line. Unlike in many languages, comment data is available in the parsed representation of xh source code. (*likeshell* 17, *syn.reversibleparsing* 1)

6. *syn.stringquoting* Single-quoted and double-quoted strings have exactly the semantics they do in Perl or bash; that is, single-quoted strings are oblivious to most unquoting features, whereas double-quoted strings are interpolated. (*likeshell* 17)

7. *syn.stringexpressions* Within a double-quoted string, you need to prefix any interpolating () group with a $ to make it active. (*nodebug* 11, *likeshell* 17)

8. *syn.toplevelexpressions* Outside words and quoted strings, () does not require a $ prefix to interpolate. Put differently, the $ is required if and only if you are interpolating by same-word string concatenation. (*realprog* 1, *ergonomic* 5)

9. *syn.flattoplevel* xh's toplevel grammar is based on Tcl, not Lisp; this means that you don't need to wrap each statement in parentheses. Line breaks are significant unless preceded with \ or inside a list. Unlike bash and tcl, all sub-lists are parsed as in Lisp; that is, this toplevel syntax applies only at the outermost level.

    - *initial assumption*
    - *likeshell* 17
    - *ergonomic* 5

Chapter 4

runtime

xh-script operates within a hosting environment that manages things like memory allocation and thread/evaluation scheduling. Beyond this, we also need a quoted-value format that's more efficient than doing a bunch of string manipulation (*xhr.representation* 6, *xhr.flatcontainers* 7, *xhr.deduplication* 9).

1. *xhr.priorityqueue* Evaluation always happens as a process of pulling expressions from a priority queue.

   - *initial assumption*
   - *xhs.relational* 4
   - *xhs.bestfirst* 5

2. *xhr.prioritytracing* Every expression in the queue knows its "origin" for scheduling purposes. (*xhs.bestfirst* 5, *xhr.priorityqueue* 1)

3. *xhr.staticinline* Function compositions should be added as derived definitions to minimize the number of symbol-table lookups per unit rewriting distance.

   - *initial assumption*
   - *notslow* 9
   - *xhs.relational* 4

4. *xhr.latency* The runtime should provide low enough latency that it can be used as the graph-solving backend for RPC routing.

   - *initial assumption*
   - *notslow* 9
   - *routing* 64

5. *xhr.valuecache* To guarantee low latency, the runtime should emit transient values for solutions it finds. These become cached bindings that can be kicked out under memory pressure, but reduce the load on the optimizer.

- *initial assumption*
- *notslow* 9
- *xhr.latency* 4

6. *xhr.representation* Every quasiquoted form with variant pieces should be represented as a separate instantiable class.

   - *initial assumption*
   - *quasiquote* 23
   - *notslow* 9
   - *xhs.eval-identities* 3

7. *xhr.flatcontainers* Quasiquoted structures are profiled for the distributions of their children (upon expansion); for strongly nonuniform distributions, specialized flattened container types are generated.

   - *initial assumption*
   - *quasiquote* 23
   - *notslow* 9
   - *xhs.eval-identities* 3
   - *xhr.staticinline* 3
   - *xhr.representation* 6

8. *xhr.flatlimit* Containers should be flattened until the type-encoding overhead is minimized for the given (possibly-contextful) distribution of values. In practice, this probably means using PPM and Huffman coding with an initial noise floor to prevent short-run overfitting.

   - *initial assumption*
   - *notslow* 9
   - *xhr.flatcontainers* 7

9. *xhr.deduplication* Every independent value within a quasiquoted form should be referred to by a structural signature, in our case SHA-256. This trivially causes strings, and by extension execution paths, to be deduplicated. Because we assume no hash collisions, xh string values have no defined instance affinity (apropos of *refaffinity* 50).

   - *heap* 46
   - *xhr.staticinline* 3
   - *xhr.representation* 6
   - *xhr.hinting* 11

10. *xhr.pointerentropy* 256 bits is sufficient to encode any pointer.

- *initial assumption*
- *uniqueid* 51
- *refaffinity* 50
- *xhr.deduplication* 9

11. *xhr.hinting* Expressions should be hinted with tags that track and influence their paths through the search space. The optimizer uses machine learning against these tags to predict successful search strategies.

    - *initial assumption*
    - *notslow* 9
    - *xhs.bestfirst* 5
    - *xhs.nocut* 6

12. *xhr.hashing* The runtime should use some type of masked hashing strategy (or other decision tree) to minimize the expected resolution time for each expression. (*initial assumption*, *xhr.hinting* 11)

13. *xhr.transientprediction* Many functions will end up returning lazy values, and most of the time those lazy values will eventually be realized. The runtime should have some expectation of which lazy sub-values will be realized, and with what probability; this influences its search strategy in the future.

    - *initial assumption*
    - *notslow* 9
    - *xhs.transientdefs* 15
    - *xhs.bestfirst* 5
    - *xhr.hinting* 11

14. *xhr.override* The user must be able to completely override any strategy preferences the runtime has. The runtime can be arbitrarily wrong and the user can be arbitrarily right.

    - *initial assumption*
    - *xhs.bestfirst* 5
    - *xhr.hinting* 11
    - *xhr.transientprediction* 13

15. *xhr.externalstrategy* The xh runtime does not itself define the evaluation strategy, nor does it internally observe things; this is done as part of the evaluation functions in the standard library. The only thing the xh runtime provides is a scheduled/prioritized event loop.

    - *initial assumption*

- *abstract* 36
- *code=data* 37
- *xhr.override* 14

16. *xhr.evaluatorapi* Evaluator functions are straightforward to write, and the standard library includes several designed for different use cases (e.g. local, distributed, profiling). Any significantly nontrivial aspect of it is factored off into an API.

    - *initial assumption*
    - *xhr.override* 14
    - *xhr.externalstrategy* 15

17. *xhr.evaluatorbase* The runtime is itself subject to evaluation (since it's self-hosting), and the base evaluator is implemented in Perl, C, or Javascript. This base evaluator runs locally; the distributed evaluator runs on top of it.

    - *initial assumption*
    - *xh2perl* 41
    - *xh2c* 40
    - *xh2js* 42
    - *inperl* 56
    - *selfhost* 43
    - *xhr.override* 14
    - *xhr.evaluatorapi* 16

18. *xhr.cryptographic* Any function can be modeled as a cipher and subject to forms of cryptanalysis to discover structure. The worst case is a truly random mapping that requires each permutation to be evaluated independently. (This is relevant for code synthesis, which is an inversion of the evaluator.) (*initial assumption*, *xhs.relational* 4)

19. *xhr.uniformityreduction* Entropy can be reduced by biasing otherwise uniform distributions of `(amb)` alternatives, possibly by looking at context. This can be done using real-world data, if any exists. (*xhr.cryptographic* 18)

20. *xhr.separability* Entropy can be reduced by identifying input separability or other such structure. This is done through cryptanalysis, which must ultimately be verified structurally even if observed empirically.[1] (*xhr.cryptographic* 18)

[1] **TODO:** is this really true? If so, we're in an awkward place where inference is concerned.

Chapter 5

computational abstraction

1. *ca.structured* All compilation is run through a structured programming layer that has abstractions for numeric operations and basic control flow. Shortcuts for higher-level operations are provided to leverage platform-specific optimized libraries.

    - *initial assumption*
    - *xh2c* 40
    - *xh2perl* 41
    - *xh2js* 42

2. *ca.varwidthint* Integer operations have signed and unsigned variants, and exist at any bit width. xh doesn't restrict to 32/64 bits (or other common values) because not all backends, e.g. Perl and Javascript, support all bit widths natively. (*initial assumption*, *inperl* 56)

3. *ca.float* Floating-point operations are defined for 32-bit and 64-bit floats. These are present on every sane platform. (*initial assumption*)

4. *ca.flatmemory* We can't assume that the underlying backend provides any data structures for us; we just address memory as a flat bunch of bytes. It's necessary to do this because we implement our own memory paging. (*notslow* 9, *no-oome* 19)

5. *ca.harvard* Data memory is separate from instructions; this abstraction has no homoiconicity at all. It's ok to do this here because all code at this level is backend-specific and machine generated. The only exception to this is that you can refer to function pointers, but they're assumed to be untyped and opaque.

    - *initial assumption*
    - *xh2c* 40

- *xh2perl* 41
- *xh2js* 42
- *ca.flatmemory* 4

6. *ca.usergc* All garbage collectors are implemented in xh-script and compiled into the flat memory model.

   - *initial assumption*
   - *realprog* 1
   - *imperative* 18
   - *xh2c* 40
   - *xh2perl* 41
   - *xh2js* 42
   - *ca.flatmemory* 4
   - *ca.harvard* 5

7. *ca.lazygc* Garbage collectors are lazy, since the heap is useful as a cache and is swapped to disk.

   - *no-oome* 19
   - *xhs.relational* 4
   - *xhs.bestfirst* 5
   - *xhs.transientdefs* 15
   - *xhs.globaldefs* 16
   - *ca.flatmemory* 4

8. *ca.gclocality* GC is a strictly local process; all values sent over RPCs are quoted. The only exception is for mutable resources, which can be referred to remotely by acquiring a unique reference to it. When those references are no longer referred to, the remote instance notifies the owner. If the remote instance drops offline, any references it holds are invalidated.

   - *distributed* 3
   - *unreliable* 10
   - *remotestuff* 21
   - *printstruct* 27
   - *immutable* 28
   - *stateown* 31
   - *ca.usergc* 6

9. *ca.userprofiling* Profiling is implemented as an xh-script library and is compiled into each backend automatically.

- *initial assumption*
- *xh2c* 40
- *xh2perl* 41
- *xh2js* 42
- *notslow* 9
- *xhs.relational* 4

10. *ca.userrelational* Relational evaluation is implemented as an xh-script library that is then compiled into each backend automatically. Because of this self reference, the xh image contains two implementations of the relational evaluator.

    - *initial assumption*
    - *xh2c* 40
    - *xh2perl* 41
    - *xh2js* 42
    - *xhs.relational* 4

11. *ca.backendrelational* Because the computational abstraction is xh-script hosted, compiler backends assume a relational evaluator. (*initial assumption*, *ca.userrelational* 10)

12. *ca.compiledinstances* Every image compiled into a backend becomes a connected xh instance with an independently-managed heap, symbol table, etc. Communication is done via the usual RPC protocol. In a sense, the default xh image is one that has been precompiled into Perl.

    - *initial assumption*
    - *distributed* 3
    - *notslow* 9

13. *ca.compiledvisibility* Compiled images are visible in the global xh network topology. (*ca.compiledinstances* 12)

14. *ca.selfmanagement* Compiled images don't have managing instances; that is, they are expected to recompile themselves in response to any profile-guided optimization. (*ca.compiledinstances* 12, *ca.compiledvisibility* 13)

15. *ca.nomultiplicity* Images can't spontaneously multiply for the purpose of exploring the space of possible optimizations. This would require some kind of instance GC process, which is beyond the scope of xh. The only exception is that every compiler backend can create a new instance, obviously, since runtimes in different languages don't tend to work together trivially. (*ca.selfmanagement* 14)

16. *ca.mipermachine* Even if xh is careful about the number of instances it creates, there will be multiple instances per physical machine.

    - *distributed* 3
    - *ca.compiledinstances* 12
    - *ca.compiledvisibility* 13

17. *ca.machineid* xh instances need a way to unambiguously identify a machine, even when the topology spans multiple networks (so there may be hostname collisions).

    - *remotestuff* 21
    - *opaques* 29
    - *ca.mipermachine* 16

18. *ca.machineiduuid* Hostnames as aliases for machine UUIDs is an acceptable strategy for dealing with machine identification. It's important to make the names as human-friendly as possible. (*ergonomic* 5, *ca.machineid* 17)

Chapter 6

feasibility of relational evaluation

Writing a compiler in a relational framework is slightly insane because there's a fine line between judiciously combining known strategies for things and synthesizing algorithms. The only way for the problem to be remotely tractable is for us to either use heuristics, or to cache solutions somewhere. xh does the latter.

The idea of a "solution" deserves some discussion. xh doesn't need to know answers to questions, but it does need to have something that decreases the entropy of the search space. Specifically, xh most likely has a synthesis rate of 20 bits per minute if we're lucky, and that number goes up exponentially with additional bits (though not if the bits are separable, which xh can figure out using differential cryptanalysis).

Using techniques like cryptanalysis is ideal because it allows the core relational evaluator to be unbiased; any optimizations it makes are empirically verified first. Verification is itself not quite trivial, since xh won't always have a predictive model to prove things (and proving things is hard in any case). To get around this, xh is allowed to assume that correlations it observes are reproducible.[1]

More specifically, xh needs to deal with:

1. Black-box systems (so no analytical solutions or proofs)
2. Time-variant systems
3. Noisy systems

All of these can be mitigated to some extent by repeated observation. In particular, $H(\text{model}) \leq H(\text{observations})$ obviously applies. In practice, this is

[1] This may be suggested by Occam's Razor, depending on how you look at it, though it's still a weak form of the causation-from-correlation inference fallacy.

unlikely to be a problem; it's fine if xh never fully understands the systems it's dealing with as long as it observes the most visible/important aspects.

Fitting a model to observation data is itself subject to optimization; not all models are equally probable. xh is more about the expected than the worst case, so biasing the space of approximators to reduce modeling entropy is fair game.

Modeling solutions is related to the representational optimization implied by *xhr.flatlimit* 8, which gives us a convenient way to quantify optimization: an optimal program has the highest computational throughput per unit time. In practical terms, this means that (1) the representations of data tend to be small/efficient, and (2) they are moved through components that can process them quickly (i.e. no significant bottlenecks). Because this system is modeled as a throughput problem, the network routing logic from *routing* 64 applies to algorithm optimization.

Success/failure prediction is nontrivial because values don't have to be fully realized to be useful. For example, suppose we have two ways to generate the first 4KB of a string, one of which also produces the next 4KB quickly and one of which never realizes it. If all we need is the first 1KB, then the second 4KB of the string doesn't matter. So the question isn't what realizes the value as a whole, it's what ends up causing the value to block evaluation later on. We want to predict and minimize blocking.

Another way to put it is that we want to minimize the time until a value can be garbage-collected. (**TODO:** is this true? What are the implications?)

Chapter 7

feasibility of representational abstraction

Given *immutable* 28, representational abstraction is just a question of whole-value encoding; we don't have to worry about things like updating a value in place. The goal of representational abstraction is to generate value encodings that minimize the expected heap size, which can happen easily during garbage collection (since the heap gets copied in any case). There will end up being several such encodings for any given type of value, and the optimizer will choose different ones depending on the use case. The presence of such alternatives implies the existence of transcoding functions, which means that n alternatives require $O(n^2)$ code space (and possibly more because representations are sometimes mutually dependent). We can mitigate this slightly by generating these functions lazily.

Concretely speaking, representational abstraction applies to strings and lists, which are the only two data types in xh (*xhs.datatypes* 1). This makes analysis interesting because there isn't much of a distinction between types and values; for example, `3.141592` is a bare string that can be interpreted as a number. Most of the typeful semantics of xh are built around structured transformations of quoted values, so any type inference involves predicting which transformations will apply (chapter 8).

We also need to enable recursive abstraction to handle things like Church-encoded numbers. Generated representations are subject to exactly the same optimization strategies that apply to primitives.

Chapter 8

quoted value relations

xh functions are implemented as string→string relations whose operands bind in ways consistent with the fundamental structure of the language. That is, list forms are always fully matched; it isn't possible to match an unescaped open bracket alone, for example. Fundamental structure includes the following constructs:

```
(x1 x2 ... xN)      # paren-list
[x1 x2 ... xN]      # bracket-list
{x1 x2 ... xN}      # brace-list
"stuff"             # double-quoted string atom
'stuff'             # single-quoted string atom
word                # unquoted, untyped atom
```

Of these, lists, double-quoted atoms, and unquoted atoms are subject to interpolation:

```
$x                  # variable value as a single element
!x                  # quoted variable value
(f x y ...)         # function result interpolation
!(f x y ...)        # quoted function result
```

1. *qvr.unwrap* `@` is a right-associative prefix operator that unwraps one layer of lists. For example, if `$x = [[1 2] [3 4]]`, then `@@$x` would be `1 2 3 4`. Any scalars are treated as single-element lists.

2. *qvr.quoted* `!` and `$` are two ways to dereference something; `$` (implied if you use `()`) may block until a complete value is available, whereas `!` immediately quotes the value in whatever state of evaluation it happens to be in (*quotestruct* 26, *printlazy* 34). You can use quoted-value introspection and evaluation functions to inspect and progress the state of such a value.

3. *qvr.unquote* `$` is a prefix operator that unquotes things until they converge to their asymptotic value limit. `()` is a special form that calls a

function, returning its unquoted result (*xhs.mapsasrelations* 26); function calls are different from general unquoting in that unquoting is a strictly static operation, whereas function calls cause values to be rerun through relations. Within a double-quoted string, `()` must be written as `$()` (*syn.stringquoting* 6).

Destructuring constructs provide some degree of type selection. For example, lists are typeful:

```
[@$xs]              # matches [1 2 3], but not (1 2 3) or {1 2 3}
(amb [@$xs] {@$xs}) # matches [1 2 3] and {1 2 3}, but not (1 2 3)
```

Strings are structure-preserving, which means you can write parsers using destructuring notation. For example, the following parses $a^n b^n c$:

```
def (rep 0 $x) ''
def (rep $n $x) "$x$(rep (dec $n) $x)"
def (parse "$(rep $n a)$(rep $n b)c") ...
```

The same kind of logic applies to lists by the following isomorphism:

```
def (list->string []) ''
def (list->string [$x @$xs]) "$x$(list->string [@$xs])"
```

Part II

base implementation

Chapter 9

self-replication

Listing 9.1 boot/xh-header

```
#!/usr/bin/env perl
#<body style='display:none'><script id='self' type='xh'>
BEGIN {eval($::xh_bootstrap = q{
# xh | https://github.com/spencertipping/xh
# Copyright (C) 2014, Spencer Tipping
# Licensed under the terms of the MIT source code license
use 5.014;
package xh;
our %modules;
our @module_ordering;
our %eval_numbers = (1 => '$xh_bootstrap');

sub with_eval_rewriting(&) {
  my @result = eval {$_[0]->(@_[1..$#_])};
  die $@ =~ s/\(eval (\d+)\)/$eval_numbers{$1}/egr if $@;
  @result;
}

sub named_eval {
  my ($name, $code) = @_;
  $eval_numbers{$1 + 1} = $name if eval('__FILE__') =~ /\(eval (\d+)\)/;
  with_eval_rewriting {eval $code; die $@ if $@};
}

our %compilers = (
  pl => sub {
    my $package = $_[0] =~ s/\./::/gr;
    eval {named_eval $_[0], "{package ::$package;\n$_[1]\n}"};
    die "error compiling module $_[0]: $@" if $@;
```

```
  },
  html => sub {});

sub defmodule {
  my ($name, $code, @args) = @_;
  chomp($modules{$name} = $code);
  push @module_ordering, $name;
  my ($base, $extension) = split /\.(\w+$)/, $name;
  die "undefined module extension '$extension' for $name"
    unless exists $compilers{$extension};
  $compilers{$extension}->($base, $code, @args);
}

chomp($modules{bootstrap} = $::xh_bootstrap);
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 9.2 boot/xh-header (continued)

```
sub serialize_module {
  my ($module)   = @_;
  my $contents   = $modules{$module};
  my $terminator = '_';
  $terminator .= '_' while $contents =~ /^$terminator$/m;
  join "\n", "BEGIN {xh::defmodule('$module', <<'$terminator')}",
             $contents,
             $terminator;
}

sub image {
  join "\n", "#!/usr/bin/env perl",
             "#<body style='display:none'><script type='xh'>",
             "BEGIN {eval(\$::xh_bootstrap = <<'_')}",
             $modules{bootstrap},
             '_',
             map(serialize_module($_), grep !/\.html$/, @module_ordering),
             "#</" . "script>",
             map(serialize_module($_), grep  /\.html$/, @module_ordering),
             "xh::main::main;\n__DATA__";
}
})}
```

Here's a quick test implementation of `xh::main::main`; its purpose is to make sure replication works. This won't be present in real images:

Listing 9.3 src/test/main.pl

```
BEGIN {xh::defmodule('xh::main.pl', <<'_')}
sub main {
  # TESTCODE (FIXME if in a real image)
  print ::xh::image if grep /^--recompile/, @ARGV;

  if (grep /^--repl/, @ARGV) {
    print STDERR "> ";
    while (<STDIN>) {
      chomp;
      if (length) {
        eval {
          my ($parsed) = xh::corescript::parse $_;
          my $result =
            eval {xh::corescript::evaluate($parsed,
                                           $xh::corescript::global_bindings,
                                           2)};
          print $@ ? "! $@\n"
                   : "= " . $result->str . "\n";
        };
        print "! $@\n" if $@;
      }
      print STDERR "> ";
    }
  }
}
_
```

Chapter 10

xh-core script

xh-script is complicated, and it would be overkill to write a complete parser in Perl considering that we'll end up needing a self-hosting parser later on. Instead, we define a sub-language called *xh-core script* that consists of assembly-level commands. This language is a syntactic subset of xh-script whose execution semantics are linear and imperative, and as such can be trivially compiled to Perl or Javascript (not C because we'll need a GC first).

This sublanguage uses a simplified form of `def` that produces traditional functions. As a result, defined things cannot use destructuring binds and are monomorphic under the symbol name. xh-core script also doesn't support any string interpolation or other magic syntax.

Listing 10.1 src/corescript.pl

```
BEGIN {xh::defmodule('xh::corescript.pl', <<'_')}
sub parse;
sub evaluate;

sub into_hash {
  my %result;
  for (my $i = 0; $i < @_; $i += 2) {
    my $k = ref($_[$i]) ? $_[$i]->str : $_[$i];
    $result{$k} = $_[$i + 1];
  }
  \%result;
}

sub xh::corescript::literal::new  {my ($c, $x) = @_; bless \$x, $c}
sub xh::corescript::var::new      {my ($c, $x) = @_; bless \$x, $c}
sub xh::corescript::list::new     {bless [@_[1..$#_]], $_[0]}
sub xh::corescript::array::new    {bless [@_[1..$#_]], $_[0]}
sub xh::corescript::hash::new     {bless into_hash(@_[1..$#_]), $_[0]}
sub xh::corescript::bindings::new {bless [$_[1], $_[2]], $_[0]}
```

```
sub xh::corescript::fn::new       {bless [$_[1], $_[2]], $_[0]}
sub xh::corescript::native::new   {bless [$_[1], $_[2]], $_[0]}
sub xh::corescript::delay::new    {bless [@_[1..$#_]], $_[0]}

sub xh::corescript::literal::concrete  {1}
sub xh::corescript::var::concrete      {0}
sub xh::corescript::list::concrete     {0}
sub xh::corescript::array::concrete    {1}
sub xh::corescript::hash::concrete     {1}
sub xh::corescript::bindings::concrete {1}
sub xh::corescript::fn::concrete       {1}
sub xh::corescript::native::concrete   {1}
sub xh::corescript::delay::concrete    {0}

sub xh::corescript::literal::true  { length ${$_[0]} && ${$_[0]} ne '0' }
sub xh::corescript::var::true      {1}
sub xh::corescript::list::true     {1}
sub xh::corescript::array::true    {1}
sub xh::corescript::hash::true     {1}
sub xh::corescript::bindings::true {1}
sub xh::corescript::fn::true       {1}
sub xh::corescript::native::true   {1}
sub xh::corescript::delay::true    {1}

sub strhash {
  my $h = 0;
  $h = $h * 33 ^ ord for split //, $_[0];
  $h;
}

sub arrayhash {
  my $h = 0;
  $h = $h * 65 ^ (ref($_) ? $_->hashcode : strhash($_)) for @_;
  $h;
}

sub hashhash {
  my @ks = sort keys %{$_[0]};
  arrayhash(@ks) ^ arrayhash(@{$_[0]}{@ks});
}

sub xh::corescript::literal::hashcode  { strhash ${$_[0]} }
sub xh::corescript::var::hashcode      { ~strhash ${$_[0]} }
sub xh::corescript::list::hashcode     { arrayhash @{$_[0]} }
sub xh::corescript::array::hashcode    { 1 + arrayhash @{$_[0]} }
sub xh::corescript::hash::hashcode     { hashhash $_[0] }
```

```
66
67  sub xh::corescript::bindings::hashcode {
68    (defined ${$_[0]}[0] ? $$_[0]->hashcode : 0) ^ hashhash ${$_[0]}[1];
69  }
70
71  sub xh::corescript::fn::hashcode     { strhash(${$_[0]}[0])
72                                       ^ ${$_[0]}[1]->hashcode }
73  sub xh::corescript::native::hashcode { 65 * strhash(${$_[0]}[0]) }
74  sub xh::corescript::delay::hashcode  { ${$_[0]}[0]->hashcode
75                                       ^ arrayhash @{$_[0]}[1..$#{$_[0]}] }
76
77  # shorthands
78  use constant literal  => 'xh::corescript::literal';
79  use constant var      => 'xh::corescript::var';
80  use constant list     => 'xh::corescript::list';
81  use constant hash     => 'xh::corescript::hash';
82  use constant array    => 'xh::corescript::array';
83  use constant bindings => 'xh::corescript::bindings';
84  use constant fn       => 'xh::corescript::fn';
85  use constant native   => 'xh::corescript::native';
86  use constant delay    => 'xh::corescript::delay';
87
88  use constant REF_IDS => 0;
89
90  our $deadline = 0;
91  our %globals;
92  our $global_bindings = bindings->new(undef, \%globals);
93
94  our $apply = literal->new('apply');
95
96  sub quote_call {
97    my ($f, $xs) = @_;
98    ref($xs) eq array ? list->new($f, @$xs)
99                      : list->new($apply, $f, $xs);
100 }
101
102 sub ref_id {
103   return '' unless REF_IDS and time > $deadline;
104   ($_[0] =~ s/^.*\(0x..(...).*$/$1/r) . ':';
105 }
106
107 sub xh::corescript::literal::str {
108   my $s = ${$_[0]};
109   ref_id($_[0]) .
110   ($s =~ /['()[]{},\s]|^$/ ? "'" . (${$_[0]} =~ s/'/\\'/gr) . "'"
111                            : $s);
```

```
}

sub xh::corescript::var::str   { ref_id($_[0]) . "\$${$_[0]}" }
sub xh::corescript::list::str  { ref_id($_[0]) .
                                   '(' . join(' ', map $_->str, @{$_[0]})
                                       . ')' }

sub xh::corescript::array::str { ref_id($_[0]) .
                                   '[' . join(' ', map $_->str, @{$_[0]})
                                       . ']' }

sub xh::corescript::hash::str {
  ref_id($_[0]) .
  '{' . join(', ', map $_ . ' ' . ${$_[0]}{$_}->str, sort keys %{$_[0]})
      . '}';
}

sub xh::corescript::bindings::str {
  my ($self) = @_;
  my ($parent, $h) = @$self;
  ref_id($_[0]) .
  '(bindings ' . ($parent ? $parent->str : "''")
         . ' ' . join(', ', map "$_ " . $$h{$_}->str, sort keys %$h) . ')';
}

sub xh::corescript::fn::str {
  my ($self) = @_;
  my ($formal, $body) = @$self;
  ref_id($_[0]) . "(fn* $formal " . $body->str . ')';
}

sub xh::corescript::native::str {
  my ($self) = @_;
  ref_id($_[0]) . "\$$$self[0]";
}

sub xh::corescript::delay::str {
  my ($self) = @_;
  ref_id($_[0]) . '(delay ' . join(' ', map $_->str, @$self) . ')';
}

sub xh::corescript::bindings::get {
  my ($self, $x) = @_;
  my ($parent, $h) = @$self;
  return undef unless ref($x) eq literal or ref($x) eq var;
  my $binding = $$h{$x->name} // ($parent && $parent->get($x));
```

```
  ref($binding) ? $binding : undef;
}

sub xh::corescript::bindings::contains {
  my ($self, $x) = @_;
  my ($parent, $h) = @$self;
  exists($$h{$x->name}) || $parent && $parent->contains($x);
}

sub xh::corescript::hash::get {
  my ($self, $x) = @_;
  $$self{$x->str};
}

sub xh::corescript::hash::contains {
  my ($self, $x) = @_;
  exists $$self{$x->str};
}

sub xh::corescript::array::get {
  my ($self, $x) = @_;
  return undef unless ref($x) eq literal;
  my $i = $x->name;
  die "array index must be number (got $i instead)" unless $i eq $i + 0;
  $$self[$i];
}

*xh::corescript::list::get = *xh::corescript::array::get;

sub xh::corescript::literal::get {
  my ($self, $x) = @_;
  return undef unless ref($x) eq literal;
  my $i = $x->name;
  die "literal index must be number (got $i instead)" unless $i eq $i + 0;
  literal->new(ord substr $$self, $i, 1);
}

sub xh::corescript::var::name     { ${$_[0]} }
sub xh::corescript::literal::name { ${$_[0]} }
sub xh::corescript::native::name  { ${$_[0]}[0] }

sub xh::corescript::literal::eval  { $_[0] }
sub xh::corescript::native::eval   { $_[0] }
sub xh::corescript::bindings::eval { $_[0] }

sub xh::corescript::var::eval { my ($self, $bindings) = @_;
```

```
                                  $bindings->get($self) // $self }

sub xh::corescript::list::eval {
  my ($self, $bindings) = @_;
  my ($f, @xs) = @$self;
  my $r = $f->invoke($bindings, array->new(@xs));
  return $r if defined $r;
  my @e = map $_->eval($bindings), @$self;
  $e[0] = $bindings->get($e[0]) // $e[0]
    if ref($e[0]) eq literal or ref($e[0]) eq var;
  $$self[$_] eq $e[$_] or return list->new(@e) for 0..$#e;
  $self;
}

sub xh::corescript::array::eval {
  my ($self, $bindings) = @_;
  my @e = map $_->eval($bindings), @$self;
  $$self[$_] eq $e[$_] or return array->new(@e) for 0..$#e;
  $self;
}

sub xh::corescript::hash::eval {
  my ($self, $bindings) = @_;
  my %new;
  my $changed = 0;
  for (keys %$self) {
    my $v0 = $$self{$_};
    my $v  = $new{$_} = $v0->eval($bindings);
    last if $changed = $v ne $v0;
  }
  if ($changed) {
    $new{$_} //= $$self{$_}->eval($bindings) for keys %$self;
    hash->new(%new);
  } else {
    $self;
  }
}

sub xh::corescript::fn::eval {
  my ($self, $bindings) = @_;
  my ($formal, $body) = @$self;
  my $newbody = $body->eval(
    bindings->new($bindings, {$formal => 0}));
  $body eq $newbody ? $self
                    : fn->new($formal, $newbody);
}
```

```
250
251 sub xh::corescript::delay::eval {
252   my ($self, $bindings) = @_;
253   my ($values, @body) = @$self;
254   for my $x (@$values) {
255     if (!$x->concrete && $x eq $x->eval($bindings)) {
256       my @e = map $_->eval($bindings), @body;
257       $body[$_] eq $e[$_] or return delay->new($values, @e) for 0..$#e;
258       return $self;
259     }
260   }
261   list->new(@body)->eval($bindings);
262 }
263
264 sub xh::corescript::var::invoke {
265   my ($self, $bindings, $args) = @_;
266   my $e = $self->eval($bindings);
267   return undef if $e eq $self;
268   $e->invoke($bindings, $args);
269 }
270
271 sub xh::corescript::literal::invoke {
272   my ($self, $bindings, $args) = @_;
273   my $f = $bindings->get($self);
274   return undef unless defined $f;
275   $f->invoke($bindings, $args);
276 }
277
278 sub xh::corescript::list::invoke {
279   my ($self, $bindings, $args) = @_;
280   die quote_call($self, $args)->str . ': timeout expired'
281     if $deadline and time > $deadline;
282
283   my $evaled = $self->eval($bindings);
284   return undef if $self eq $evaled;
285
286   my $result = eval {$evaled->invoke($bindings, $args)};
287   my $error = $@;
288   if ($error) {
289     print STDERR quote_call($self, $args)->str, ': ', $error, "\n";
290     die $error;
291   } else {
292     $result;
293   }
294 }
295
```

```
sub xh::corescript::delay::invoke {
  my ($self, $bindings, $args) = @_;
  die quote_call($self, $args)->str . ': timeout expired'
    if $deadline and time > $deadline;

  my $e = $self->eval($bindings);
  return undef if $e eq $self;
  my $result = eval {$e->invoke($bindings, $args)};
  my $error = $@;
  if ($error) {
    print STDERR quote_call($self, $args)->str, ': ', $error, "\n";
    die $error;
  } else {
    $result;
  }
}

sub xh::corescript::fn::invoke {
  my ($self, $bindings, $args) = @_;
  die quote_call($self, $args)->str . ': timeout expired'
    if $deadline and time > $deadline;

  my ($formal, $body) = @$self;
  my $result = eval {
    $body->eval(bindings->new($global_bindings,
                              {$formal => $args->eval($bindings)}));
  };
  my $error = $@;
  if ($error) {
    print STDERR quote_call($self, $args)->str, ': ', $error, "\n";
    die $error;
  } else {
    $result;
  }
}

sub xh::corescript::native::invoke {
  my ($self, $bindings, $args) = @_;
  die quote_call($self, $args)->str . ': timeout expired'
    if $deadline and time > $deadline;

  my (undef, $f) = @$self;
  $args = $args->eval($bindings);
  my $result = eval {$f->($bindings, @$args)};
  my $error  = $@;
  if ($error) {
```

```
342     print STDERR quote_call($self, $args)->str, ': ', $error, "\n";
343     die $error;
344   } else {
345     $result;
346   }
347 }
348 
349 our $y     = literal->new('y');
350 our $nil   = literal->new('');
351 our $quote = literal->new('quote');
352 our $fn    = literal->new('fn*');
353 our $if    = literal->new('if');
354 
355 sub bool  { $_[0] ? $y : $nil }
356 sub quote { list->new($quote, $_[0]) }
357 
358 our %brackets = (')' => list, ']' => array, '}' => hash);
359 
360 # defines global natives that may or may not force their arguments
361 sub defglobal {
362   my %bindings = @_;
363   $globals{$_} = native->new($_, $bindings{$_}) for keys %bindings;
364 }
365 
366 defglobal '==', sub {
367   return undef unless $_[1]->concrete and $_[2]->concrete;
368   bool($_[1]->str eq $_[2]->str);
369 };
370 
371 defglobal apply => sub {
372   my ($bindings, $f, @args) = @_;
373   my $last = pop @args;
374   return undef unless ref($last) eq array;
375   $f->invoke($bindings, array->new(@args, @$last))
376     // list->new($f, @args, @$last);
377 };
378 
379 defglobal array => sub { array->new(@_[1..$#_]) };
380 
381 defglobal assoc => sub {
382   my ($bindings, $h, $k, $v) = @_;
383   if (ref($h) eq hash) {
384     return undef unless $k->concrete;
385     my $result = hash->new(%$h);
386     $$result{$k->str} = $v;
387     $result;
```

```
  } elsif (ref($h) eq array) {
    return undef unless ref($k) eq literal;
    my $result = array->new(@$h);
    $$result[$k->name] = $v;
    $result;
  } else {
    return undef;
  }
};

defglobal bindings => sub {
  my ($bindings, $parent, @xs) = @_;
  my %h;
  for (my $i = 0; $i < @xs; $i += 2) {
    $h{$xs[$i]->name} = $xs[$i + 1];
  }
  bindings->new($parent, \%h);
};

defglobal concrete => sub { bool $_[1]->concrete };
defglobal contains => sub { bool $_[1]->contains($_[2]) };
defglobal count    => sub {
  my ($bindings, $x) = @_;
  ref($x) eq array   ? literal->new(scalar @{$_[1]})
: ref($x) eq literal ? literal->new(length $x->name)
                     : undef;
};

defglobal def => sub {
  my ($bindings, $var, $x) = @_;
  return undef unless ref($var) eq literal;
  $globals{$var->name} = $x;
  $var;
};

defglobal defs => sub { $_[0] };

defglobal delay => sub {
  my ($bindings, $exprs, @body) = @_;
  return undef unless ref($exprs) eq array;
  delay->new($exprs, @body);
};

defglobal do => sub {
  my ($bindings, @body) = @_;
  my $result;
```

```
  $result = $_->eval($bindings) for @body;
  $result;
};

defglobal empty => sub {
  return undef unless $_[1]->concrete;
  ref($_[1])->new;
};

defglobal 'fn*' => sub {
  my ($bindings, $formal, $body) = @_;
  $formal = $formal->eval($bindings);
  return undef unless ref($formal) eq literal;
  fn->new($formal->name, $body->eval($bindings));
};

defglobal get => sub {
  return undef unless $_[1]->concrete and $_[2]->concrete;
  $_[1]->get($_[2]);
};

defglobal globals => sub { $global_bindings };
defglobal hash    => sub { hash->new(@_[1..$#_]) };
defglobal hashcode => sub {
  my ($bindings, $x) = @_;
  my $chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
            . 'abcdefghijklmnopqrstuvwxyz'
            . '0123456789'
            . '+-*&^%@!~`|=_<>,.:;';
  my $n = length $chars;

  my $h = $x->hashcode;
  my $s = '';
  $s .= substr($chars, $h % $n, 1), $h /= $n while $h = int $h;
  literal->new($s);
};

defglobal if => sub {
  my ($bindings, $cond, $then, $else) = @_;
  $cond = $cond->eval($bindings);
  return undef unless $cond->concrete;
  ($cond->true ? $then : $else)->eval($bindings);
};

defglobal into => sub {
  my ($bindings, $to, $from) = @_;
```

```
480   return undef unless ref($to)   eq array
481                   and ref($from) eq array;
482   ref($to)->new(@$to, @$from);
483 };
484
485 defglobal iota => sub {
486   my ($bindings, $i) = @_;
487   return undef unless ref($i) eq literal;
488   array->new(map literal->new($_), 0..$i->name-1);
489 };
490
491 defglobal keys => sub {
492   ref($_[1]) eq hash  ? array->new(map parse($_), sort keys %{$_[1]})
493 : ref($_[1]) eq array ? array->new(map literal->new($_), 0..$#{$_[1]})
494                       : undef;
495 };
496
497 defglobal len     => sub { literal->new(length $_[1]->name) };
498 defglobal list    => sub { list->new(@_[1..$#_]) };
499 defglobal literal => sub { literal->new($_[1]->name) };
500
501 defglobal 'list->array' => sub {
502   my ($bindings, $l) = @_;
503   array->new(@$l);
504 };
505
506 defglobal module => sub {
507   return undef unless ref($_[1]) eq literal;
508   literal->new($xh::modules{$_[1]->name});
509 };
510
511 defglobal modules => sub {
512   array->new(map literal->new($_), @xh::module_ordering);
513 };
514
515 defglobal parse => sub {
516   return undef unless ref($_[1]) eq literal;
517   array->new(parse $_[1]->name);
518 };
519
520 defglobal quote => sub { $_[1] };
521
522 defglobal realtype => sub {
523   return undef unless $_[1]->concrete;
524   literal->new(ref($_[1]) =~ s/^.*::/r);
525 };
```

```
526
527 defglobal reduce => sub {
528   my ($bindings, $f, $init, $xs) = @_;
529   return undef unless ref($xs) eq array;
530   for my $x (@$xs) {
531     my $args = array->new($init, $x);
532     $init = $f->invoke($bindings, $args) // quote_call($f, $args);
533   }
534   $init;
535 };
536
537 defglobal scope => sub {
538   my ($bindings, $b, $v) = @_;
539   return undef unless ref($b) eq hash;
540   $v->eval(bindings->new($bindings, $b));
541 };
542
543 defglobal slice => sub {
544   my ($bindings, $xs, $lower, $upper) = @_;
545   $lower //= literal->new(0);
546   $upper //= literal->new(ref($xs) eq array   ? scalar @$xs
547                           : ref($xs) eq literal ? length($xs) - 1
548                                                 : return undef);
549   return undef unless ref($lower) eq literal
550                   and ref($upper) eq literal;
551   ref($xs) eq array   ? array->new($upper->name - 1 >= $lower->name
552                                    ? @$xs[$lower->name .. $upper->name - 1]
553                                    : ())
554 : ref($xs) eq literal ? literal->new(substr $xs->name, $lower,
555                                                 $upper - $lower)
556                       : undef;
557 };
558
559 defglobal str => sub {
560   ref($_) eq literal or return undef for @_[1..$#_];
561   literal->new(join '', map $_->name, @_[1..$#_]);
562 };
563
564 defglobal type => sub { literal->new(ref($_[1]) =~ s/^.*::/r) };
565
566 defglobal undef => sub {
567   my ($bindings, $x) = @_;
568   return undef unless ref($x) eq literal;
569   delete $globals{$x->name};
570   $x;
571 };
```

```
572
573 defglobal unquote => sub {
574   my ($bindings, $x, $b) = @_;
575   $x->eval($bindings)->eval($b // $bindings);
576 };
577
578 defglobal vals => sub { array->new(map ${$_[1]}{$_}, sort keys %{$_[1]}) };
579 defglobal var  => sub {
580   return undef unless ref($_[1]) eq literal;
581   var->new($_[1]->name);
582 };
583
584 # Primitive arithmetic
585 BEGIN {
586   my @float_binops = qw( + - * / % ** < > <= >= == != );
587   my @float_unops  = qw( - );
588   my @int_binops   = (@float_binops, qw( << >> >>> & | ^ ));
589   my @int_unops    = (@float_unops,  qw( ~ ! ));
590
591   eval qq{
592     defglobal 'f$_', sub {
593       my (\$bindings, \$x, \$y) = \@_;
594       return undef unless ref(\$x) eq literal
595                       and ref(\$y) eq literal;
596       literal->new(\$x->name $_ \$y->name);
597     };
598   } for @float_binops;
599
600   eval qq{
601     defglobal 'fu$_', sub {
602       my (\$bindings, \$x) = \@_;
603       return undef unless ref(\$x) eq literal;
604       literal->new($_ \$x);
605     };
606   } for @float_unops;
607
608   eval qq{
609     defglobal 'i$_', sub {
610       my (\$bindings, \$x, \$y) = \@_;
611       return undef unless ref(\$x) eq literal
612                       and ref(\$y) eq literal;
613       literal->new(int(int(\$x->name) $_ int(\$y->name)));
614     };
615   } for @int_binops;
616
617   eval qq{
```

```
    defglobal 'iu$_', sub {
      my (\$bindings, \$x) = \@_;
      return undef unless ref(\$x) eq literal;
      literal->new(int($_ int(\$x->name)));
    };
  } for @int_unops;
}

our %escapes = (n => "\n", r => "\r", t => "\t");
sub parse {
  my @stack = [];
  local $_;
  while ($_[0] =~ / \G (?: (?<comment> \#.*)
                         | (?<ws>      [\s,]+)
                         | '(?<qstr>   (?:[^\\']|\\.)*)'
                         | (?<str>     [^\$()\[\]{}\s,]+)
                         | (?<var>     \$[^\$\s()\[\]{},]+)
                         | (?<opener>  [(\[{])
                         | (?<closer>  [)\]}])) /gx) {
    next if $+{comment} || $+{ws};
    my $s = $+{str};
    if (defined $s)      {push @{$stack[-1]}, literal->new($s)}
    elsif ($s = $+{var}) {push @{$stack[-1]}, var->new(substr $s, 1)}
    elsif ($+{opener})   {push @stack, []}
    elsif ($s = $+{closer}) {
      my $last = pop @stack;
      die "too many closers" unless @stack;
      push @{$stack[-1]}, $brackets{$s}->new(@$last);
    } elsif (defined($s = $+{qstr})) {
      push @{$stack[-1]},
           literal->new($s =~ s|\\(.)|$escapes{$1} // $1|egr);
    } else {
      die "unrecognized token: $_";
    }
  }
  die "unbalanced brackets: " . scalar(@stack) . " != 1"
    unless @stack == 1;
  @{$stack[0]};
}

sub evaluate {
  my ($expr, $bindings, $timeout) = @_;
  local $_;
  $deadline = $timeout ? time + $timeout : 0;
  $expr->eval($bindings // $global_bindings);
}
```

```
664
665 $::xh::compilers{xh} = sub {
666   for my $x (parse $_[1]) {
667     print STDERR '> ', $x->str, "\n";
668     my $e = evaluate($x, $global_bindings, 30);
669     print STDERR '= ', $e->str, "\n";
670   }
671 };
672 _
```

Chapter 11

html introspection

xh images can be opened as self-inspecting HTML files. This strategy of embedding xh module definitions in comments isn't very elegant, but it makes the Javascript parser easier to write. (A better system would be to have the Javascript parse everything in a single <script> tag, then build all of the HTML that way; but due to browser security restrictions, this would break local viewing.)

Listing 11.1 src/introspect/dependencies.html

```
BEGIN {xh::defmodule('js-dependencies.html', <<'_')}
<script>
-- include deps/jquery.min.js
-- include deps/caterwaul.min.js
-- include deps/caterwaul.std.min.js
-- include deps/caterwaul.ui.min.js
</script>
_
```

Listing 11.2 src/introspect/css.html

```
BEGIN {xh::defmodule('css.html', <<'_')}
<style>
/*BEGIN {xh::defmodule('introspection.css', <<'_')}*/
@import url(http://fonts.googleapis.com/css?family=Abel|Fira+Mono);
body {background:   #080808;
      color:        #eae8e4;
      margin:       auto;
      max-width:    600px;
      overflow-y:   scroll;
      padding-left: 14px;
      border-left:  solid 1px #383736}

```

```
h1 {font-family: 'Abel', monospace;
    font-weight: normal;
    font-size:   16px;
    color:       #878177;
    margin:      0}

h1:hover, h1.active {color: #eae8e4}
h1 .suffix {color: #878177}
h1 .suffix:before {content: '.'; color: #878177}

pre {font-family: 'Fira Mono', monospace;
     font-size:  10px}

.module {border-top: solid 1px #383736;
         overflow:   hidden}
.module pre {margin: 0}

#dom   {margin: 20px 0}
#dom, #dom a {font-family:     'Abel', sans-serif;
              font-size:       16px;
              text-decoration: none;
              color:           #878177}
#dom a:hover {color: #eae8e4}
.title {color: #f89421}
/*_*/
</style>
_
```

Listing 11.3 src/introspect/dom.html

```
BEGIN {xh::defmodule('dom.html', <<'_')}
<div id='dom'>
<a href='https://github.com/spencertipping/xh' target='_blank'>
  <span class='title'>xh</span></a>
<div>$ curl http://xh.spencertipping.com | perl</div>
</div>
_
```

Listing 11.4 src/introspect/js.html

```
BEGIN {xh::defmodule('introspection.html', <<'_')}
<script>
/*BEGIN {xh::defmodule('introspection.js', <<'_')}*/
// TESTCODE (should contain a functioning repl)
$(caterwaul(':all')(function () {
  $.fn.toggle_vertically(v) = $(this).each(toggle)
  -where [toggle(t = $(this).stop()) =
```

```
            cs /~animate/ {top:    v ? 0 : -0.3 * h}
     -then- t  /~animate/ {height: v ? h : 0}
/{opacity: +v} /~animate/ {queue: false, duration: 300}
     -where [cs = t.children().stop() /~css/ {position: 'relative'},
             h  = cs.first().height()]],

  $('body').empty() /~before/ jquery[head /append(css)]
                    /~append/ dom
                    /~append/ ui_for(parsed_modules)
                    /~css/    {display: 'block'},
  where [
  css               = $('style'),
  dom               = $('#dom'),
  self              = +$('script, style') *[$(x).html()] /seq /~join/ '\n',
  parse_modules(ls) = xs -se [ls *!process_line -seq] -where [
    xs              = {__ordering: []},
    name            = null,
    text            = '',
    process_line(s) = /^(?:\/\*)?BEGIN.*defmodule\('([^']+)'/.exec(s)
            -re [it ? name /eq[it[1]] -then- text /eq['']
                    : /^(?:\/\*)?_+(?:\*\/)?$/.test(s)
                      ? name -ocq- 'bootstrap.pl'
                        -then- xs[name] /eq [text /~substr/ 1]
                        -then- xs.__ordering /~push/ name
                    : text += '\n#{s}']],

  parsed_modules    = self.split(/\n/) /!parse_modules,
  ui_for(modules)   = ui -se [sections *![ui /~append/ x] -seq] -where [
    ui              = jquery in div.modules,
    toggle()        = $(this).toggleClass('active').next().stop()
                      /~toggle_vertically/ $(this).hasClass('active'),
    module_name(x)  = jquery [span.prefix /text(pieces[0])
                                + span.suffix /text(pieces[1])]
                      -where [pieces = x.split(/\./, 2)],
    sections        = seq in
                      modules.__ordering
                      *[jquery in h1 /append(x /!module_name)
                                     /css({cursor: 'pointer'})
                                     /click(toggle)
                               + div.module(pre /text(modules[x]))
                                 /toggle_vertically(false)]]]}));
/*_*/
</script>
_
```

Part III

self-hosting implementation

Chapter 12

xh-core interpreter

We defined an xh-core script interpreter in Perl in chapter 10, but we need one written in xh-core so we can compile it to other backends. This forms the foundation for xh-script, which is written in xh-core.

Some backends will provide convenient stuff like hashtables and garbage collection, but for those that don't (like C), we'll need to implement them from scratch. All we can assume in that case is support for arrays of known length. We figure this stuff out inside the xh-core script by looking for definitions of native functions and defining our own alternatives otherwise.

Another discrepancy between backends is the amount of runtime type information that's supported. By default all values are tagged unions that store their types, but GCs and such will need access to machine words.

Listing 12.1 src/corescript/boot.xh

```
(def destructuring-bind1
  (fn* outer        # outer = [template value bindings]
    (if (== literal (realtype (get $outer 0)))
      (assoc (get $outer 2) (get $outer 0) (get $outer 1))
      (if (if (== array (realtype (get $outer 0)))
            y
            (== hash (realtype (get $outer 0))))
        (reduce (fn* k
                  (delay [$k] destructuring-bind1
                              (get (get $outer 0) (get $k 1))
                              (get (get $outer 1) (get $k 1))
                              (get $k 0)))
                (get $outer 2)
                (keys (get $outer 0)))
        (get $outer 2)))))

(def fn1
  (fn* args
```

```
    (fn* (get $args 0)
      (scope (destructuring-bind1 (get $args 1)
                                  (var (get $args 0))
                                  {})
        (get $args 2)))))

(def fn
  (fn1 fn~ [formals body]
    ((fn1 gs~ [gs]
       (fn* $gs
         (scope (delay [$gs]
                 delay [$gs]
                 destructuring-bind1 $formals (var $gs) {})
           $body)))
     (str : (delay [$formals $body]
             delay [$formals $body]
             hashcode [$formals $body])))))

(def defn
  (fn [name formals body]
    (def $name (fn $formals $body))))

(defn and2 [x y] (if $x $y ''))
(defn or2  [x y] (if $x $x $y))
(defn not  [x]   (if $x '' y))
(defn xor2 [x y] (and2 (or2 $x $y) (not (and2 $x $y))))

(defn map1     [f xs] (flatmap1 (fn [x] [($f $x)]) $xs))
(defn filter1  [f xs] (flatmap1 (fn [x] (if (f $x) [$x] [])) $xs))
(defn flatmap1 [f xs] (reduce (fn [ys x] (into $ys ($f $x)))
                              (empty $xs)
                              $xs))

(defn reverse [xs] (reduce (fn [ys x] (into [$x] $ys))
                           []
                           $xs))

(defn partial fs (let1 [f] $fs (fn xs (apply $f (into (slice $fs 1) $xs)))))
(defn comp    fs (fn xs (get (reduce (fn [x f] [(apply $f $x)])
                                     $xs
                                     (reverse $fs))
                             0)))

(defn inc [x] (i+ $x 1))
(defn dec [x] (i- $x 1))
(defn let1 [k v body] ((delay [$k] fn [$k] $body) $v))
```

```
65
66  (defn first [[x]] $x)
67  (defn rest  [xs]  (slice $xs 1))
68  (defn last  [xs]  (get $xs (dec (count $xs))))
69
70  (defn .. args
71    (let1 [x] $args
72      (delay [$x]
73       delay [$x]
74       apply list delay [$x] (rest $args))))
75
76  (defn ->> forms
77    (reduce (fn [x l] (.. $l let1 _ $x $l))
78            (first $forms)
79            (rest $forms)))
80
81  (defn let [kvs body]
82    (->> (count $kvs)
83         (iota (i>> $_ 1))
84         (map1 (partial i* 2) $_)
85         (map1 (fn [i] [(get $kvs $i) (get $kvs (inc $i))]) $_)
86         (reduce (fn [code [k v]] (.. $k let1 $k $v $code))
87                 $body
88                 $_)))
89
90  (defn cond clauses
91    (->> (count $clauses)
92         (iota (i>> $_ 1))
93         (map1 (partial i* 2) $_)
94         (reduce (fn [code i]
95                   (let [[c v] (map1 (partial get $clauses) [$i (inc $i)])]
96                     (if $c $v $code)))
97                 ''
98                 (reverse $_))))
99
100 # TODO: generalize (delay) into some kind of quoting/unquoting mechanism.
101 # TODO: figure out some way to get variable shadowing
102 # TODO: generalize evaluation? (probably not necessary if we have
103 #   progressively-specified bindings and term updating)
104 # TODO: spec out state chaining mechanism
105 # TODO: multiple-value returns and/or channels? (to make it easier to work
106 #   with state) -- though maybe this can be built using quoting and
107 #   unquoting.
```

Chapter 13

data structures

Listing 13.1 src/cs/datameta.xh

```
(defn defstruct args
  (let [[name structure] $args]
    (def $name
      (apply tagged-struct $name $structure (slice $args 2)))))
```

Listing 13.2 src/cs/data.xh

```
(defstruct linear-array-container
  {length 0}
  (get [this i]
    [(== literal (realtype $i))]
    (mget (i+ $this (i* $word-size (i+ $i 1))))))
```

Chapter 14

algorithms

Listing 14.1 src/cs/algorithms.xh

```
1 # TODO
```

Chapter 15

garbage collection

Listing 15.1 src/cs/gc.xh

```
1 # TODO
```