Development

Everything about development on the Lix codebase, including design documents and maintenance documentation.

- Design planning
  - regexp engine investigation
  - Dreams
  - Language versioning
  - Docs rewrite plans
  - Nix lang v2
  - Flake stabilisation proposal
  - Observability and Protocol Design
  - Replacement CLI design & Profiles
  - Nix bootstrapping
  - Improving IFD
  - Flakes feature freeze
  - xattrs feasibility to supplement the SQLite database model
- Technical notes
  - Pointer equality
  - Darwin debugging for the Linux developer
- Release names

Design planning

This is the place where we plan major changes to various components of the code base, to tackle the big issues. Treat everything as an unimplemented idea unless stated otherwise. Make sure to link to implementation efforts in order to keep track of them.

Ideas and plans are usually written by individuals or groups of individuals, this is not an official roadmap. Anyone in the Lix community may add to it.

Design planning

regexp engine investigation

nix uses libstdc++'s `std::regex`. it uses whatever version of libstdc++ the host system has.

which it invokes in both `std::regex_replace` `std::regex_match` modes.

nix occasionally uses the flags `std::regex::extended` and `std::regex::icase` which determine the available features - it's always either no flags, or both of these together. there's also a couple things that use the flag `std::regex::ECMAScript`. when the constructor is called without a flags parameter, the flags default to `std::regex::ECMAScript` (see method signature in C++23 32.7.2), so really we have only two cases.

`std::cregex_iterator` and `std::sregex_iterator` are used.

there's a header `regex-combinators.hh` which defines `regex::group` and `regex::list`.... and a couple others that are unused. but those are just trivial textual things, not extensions, so we can ignore the file.

getting the C++ standard

someday when C++23 is official you will be able to pirate the PDF. otherwise, you can clone https://github.com/cplusplus/draft and check out the tag `n4950` which is the current formally adopted working draft as of 2024-03-14 and is intended to have the same technical content as the final standard. you can then invoke `make` in the `source` subdirectory which will produce `std.pdf`. you will need LaTeX installed. if you're ever not sure which working draft is the one that became a particular version of the standard, Wikipedia will probably tell you...

(personally I install `texlive.combined.scheme-full` from nixpkgs on all my machines that have room for it, but this is surely more than necessary, it just makes me feel warm and fuzzy -- Irenes)

chapter 32 is the one that documents regular expressions.

open questions that require reading the standard

- what are all the syntactic and semantic constructs we need to support?

required functionality

the `extended` flag, per the C++ standard, "Specifies that the grammar recognized by the regular expression engine shall be that used by extended regular expressions in POSIX.". it references

POSIX, Base Definitions and Headers, Section 9.4.

the `ECMAScript` flag "Specifies that the grammar recognized by the regular expression engine shall be that used by ECMAScript in ECMA-262, as modified in [section 32.12 of the C++ standard]." it references ECMA-262 15.10. the changes in 32.12 are important and probably do create real compatibility issues for us, though fortunately it's only a single page.

if we complete this chart we can use it to assess which existing engines would meet our needs, or how much of a pain in the ass it would be to make a new one

the columns are the two ways it gets invoked

	extended + icase	**ECMAScript**
Syntactic constructs	--	--
(TODO: fill in every construct here)		
Semantics	--	--
Case-insensitivity	yes	?
(TODO: fill in other behaviors here)		

Design planning

Dreams

This page documents the dreams of the Lix team. These are features which we have generally not roadmapped yet, and which may not have complete and thoroughly thought-through plans, and which we would like to think about more completely before implementing. We are writing them down publicly so that others can dream with us.

- language versioning https://wiki.lix.systems/books/lix-contributors/page/language-versioning
- split the evaluator into a separate process, interact with it only via rpc (horrors)
- bytecode evaluator with all the possible trappings (horrors)
  - allows arbitrary runtime-define breakpoints, watchpoints, program inspection and manipulation
  - interacts with rpc to allow perfect lsp hosts, better debuggers etc
- new gc for the evaluator to replace bdw, prototype/template for gc in eventual rust evaluator (horrors)
- flakes as a library of code that provides new nix subcommands (horrors, others)
- lix.conf `prelude-path =` for system-wide subcommands a la git (horrors)
  - also can make per-repo `lix *` commands (jade, janik)
- eval caching with a `memoize :: str -> any -> any` builtin that is overridden by `scopedImport` with a unique, deterministic subscope (horrors)
  - `import := f: memoize (toString f) (scopedImport builtins f)` (horrors)
- flake eval caching entire attrpaths: `mapAttrsRecursive (n: const (memoize n))` on all scopes/attrsets in the "flake" (horrors)
- lazyUpdate is a disaster waiting to happen, turns all values into even worse errors sources than simple thunks (and is deeply intrusive to the evaluator for little gain). why not special attrset ops `__members, __getMember` to simulate lazyUpdate in a library that doesn't infect all future versions of the language and can be transpiled when necessary? (horrors)
  - pureImport is too fine grained, store paths as boundaries actually make sense (and give memoize stable starting scopes), pure eval mode could be "ask thing to pack itself up, add to store, eval from there like nix flakes do" (horrors)
- all authoritative information about the store attached to store objects, not an sqlite database (eg in xattrs or similar) (horrors)
  - would make overlayfs stores for containers/vms trivial
- redo the lazy trees infra on the basis of "virtual" store paths and mountpoints (turning eg a zip file into a virtual mountpoint `/nix/store/lazy/thing.zip/...`) (horrors)
  - notably do not use fuse for this, just a pure vfs implementation
- fully decouple evaluator and store (horrors)
  - tvix has kind of done this with EvalIO, lix needs it too (otherwise the eval-process split will not be possible)
- store operations state, like "what derivations were realized in the last build" (Qyriad)

    - "what attrpath was this accessed by to build"
- profiler for nix code (jade)
- nix develop replace store path but actually good, with bind mounts (jade)
- nixos-rebuild gets unfucked perhaps with samueldr code (jade)
- we kinda wanna have inherits consistent by container type such that you can write inherit (thing) [a b c] to create a list, inherit (thing) { a b c } to create a set, or nest those in existing lists or sets to extend them in-place like current inherit (horrors)
- unbreak the io model (horrors)
    - currently nix has an async io model shoved into a sync runtime, and an async model that can't decide whether it's push or pull. this ***sucks***
- a dependency graph for builds which explains why different dependencies are being built
    - store path truncated to unique names in output...?
- native nix-output-monitor (`nom`) style (slash bazel-style) output formatting (showing a live updating list of stuff being built/fetched, with warnings stacking up above it)
    - web viewer for the build graph as it is happening with a nice live log viewer (jade)
        - relatedly: show closure graphs nicely (jade)
- make the store properly multi-tenant, with things like, e.g. authentication and maybe even certain http done via hooks on the client side (jade)
    - see e.g. https://git.lix.systems/lix-project/lix/issues/254
    - overall improve the clarity of what is actually running on the daemon vs the client (jade)
- replace `nix profile` with something not broken with a clear ramp to either have a manifest mutably in the store or operate mutably against a configuration directory. ideally out of tree. (jade)
- fix fs builtin problems (jade)
    - can't read symlinks
    - filterSource gives no metadata of interest esp on symlinks
    - can't synthesize symlinks or files into the store except by serious nar abuse
        - (Zoe) We can imagine a generalized transformSource builtin which presents an fs subtree as a nested attrSet containing the full metadata and contents of all files and links in the subtree, and expects an nested attrSet in the same format as output, allowing arbitrary transformations in pure nix code. As long any other other operations that touch touch the file system are disallowed inside the transformation function (evaluating other paths, building derivations, pathExists, etc) this should be a consistent operation. There may be performance/usability reasons to not use this precise interface, but I think it's a good abstract guide stone of what to strive for.
    - is lib.filesets made of evil? how does it work?
        - answer: it's filterSource in a trench coat with some set operations
    - what if you could take a source tree of a monorepo and rewrite cross project symlinks to refer to store paths of those other projects so you don't copy the entire giant repo to store every time and can have each subproject as its own store path?
    - what if you had a fetch git subtree primitive that was free if there's no modification?
        - (Zoe) It's a little trickier than just that because if you want a filtered git subtree you need some way to ensure that the filter hasn't changed either.

- Better facilities for writing performant code (Zoe)
  - Builtins should document their algorithmics and when they cause files to be written to the store
  - More opt-in persistent data structures with different performance tradeoffs that can be coerced to from the standard values
    - RRB vectors or similar for lists
    - HAMT or similar for attrSets
      - should allow using arbitrary values as keys
      - will probably need an explicit distinction between strings and symbols
      - also a separate set type, so you don't have to bother faking it with null keys
    - StringView like type for strings
      - or maybe just convert in place the first time we'd need to get the length?
- Doing something about IFD being bad (raito, pennae?):

  https://pad.lix.systems/sW0nbPohTgqy2UdIJjPeUA

fixing ux

- some way of having a persistent short lived evaluator for fast completions in CLI (Dawn)
- □ fancy □ repl, a la IPython and pry (Qyriad)
- Support instance of Lix running locally off the main page to try out
  - Obviously WebAssembly schenanigans involved
- replacing nixos-option (jade)
  - CLI commands should be possible to actually deprecate (jade)
- a debug macro like rust's `dbg!` https://doc.rust-lang.org/std/macro.dbg.html
- pipe operator (Qyriad)
  - and either haskell's $ or left pipe operator
- hyperlinked sources in docs (jade)
- a VFS mirror of the Nix store that puts the names first, attaches a more descriptive label if necessary, and then the hash, literally just for convenience (Qyriad)

slaying the hydra

these are problems that make hydra sad

- make -jsem jobserver built into Lix (horrors actually wrote one years ago)
  - this would allow much better build density in Lix and eliminate most need to tune NIX_BUILD_CORES
  - see: https://github.com/NixOS/nixpkgs/pull/143820, it turns out the make jobserver protocol is actually *horrible*, and we should instead do this with a reasonable socket protocol injected into the sandbox by Lix
- externalize deciding which host to build things on (delroth, jade)

    - this is necessary because /etc/nix/machines is really stupid and doesn't have nearly enough information to decide whether a machine can admit a job.
- make the remote protocol not suck (jade)
    - latency is bad
    - a lot of stuff blocks in ways it only dubiously needs to?
- what if you could have build cost estimates on large installations, which could go into scheduling decisions? (jade)
    - galaxy brained idea: build a neural net for derivation build costs for scheduling purposes. probably take as input the `derivation show` json with the hashes removed and then a pile of historical hydra data
    - do we have the data to do this? we want cpu time, io, and (ugh these would be very fake though because measuring memory is fraught) memory stats for builds.
    - schedule on machines that have space for the expected cpu-time/memory-time/io-time of the derivation
- make the nix daemon know what is actually building (jade)

Design planning

Language versioning

This document is extremely a draft. It needs some editing and discussion before it can be made into a useful thing. It's been simply copy pasted out of the pad in its current form.

See also

- FIXME: piegames langver ideas

musings

puck: honestly, having language version as part of a scopedImport-style primop would be funny
horrors: we're shitposting about setting language version from the source accessor

- horrors: `use features...;` file head clause
  - jade: this can be combined into feature sets like editions or such. we might become ghc haskell but whatever.
- horrors: [some kind of file head clause and/or propagation is the] only real way out of this mess that doesn't require a package manager in the package manager
  jade: yeah. doing it from flakes seems initially sane until you realise you can import below the flake in the same git repo and then blegh
  horrors: an ambient minimum-language-version binding in builtins that can be scopedImport'ed for flake support on top of this

horrific writeup

basic mechanism

add a new syntactic element that is only valid at the head of a file and used only to declare language requirements. nix versions that cannot satisfy all requirements must reject this element to situations in which two nix versions parse the same file differently, or even evaluating the same file to different derivation hashes. any kind of comment as used by eg GHC is not viable for nix for this reason.

proposed syntax for the first implementation: `use $( $feature: ident )+;`

anything ahead of this directive could be either unversioned nix code or versioned nix code (see below for details), but since the directive is only valid at the *head* of a file or expression this "code"

can only be comments. this kind of locks us into supporting the current comment syntax forever, but the comment syntax is rather fine so this won't be a problem.

each feature may declare a syntactical requirement for the file, a semantic requirement, or possible both (cf rust editions, or perl `use v<something>`).

features may be global, namespaced to their implementations, or live in a reserved `experimental` namespace an implementation can add to and remove from as it wishes with ***absolutely no guarantee of future evaluatility***.

syntactic features

syntax is entirely local to the file itself and has few to no intercompatibility constraints with other code. a very useful syntax requirement would something like `no-url-literals`, which might strip the syntactic ability to parse url-like sequences of characters into strings and, rather than nix currently does the experimental feature of the same name simply throwing a parse error, parse them as eg a lambda with a sequence of divisions in its body.

(realistically `no-url-literals` would not appear in practice, instead it should be implied by `use` itself since url literals are such an obvious misfeature)

semantic features

semantic features produce evaluation changes that could be achieved any other way. examples of this are:

- the recent change that evalutes `x` in `inherit (x) names...` at most once overall rather than once per inherited name accessed
- potential extensions to the string context mechanism
- new types of values

semantic changes may escape the expression that requires them and usually some of amount of cross-compatibility with other semantic versions must be given. using the same examples as above, considerations can include:

- observable side-effects changing (if `x` includes a call to `trace`)
- `getContext` returning sets an outside use may not expect
- value types being unknown to outside users and causing failures

this is in fact a full classification of cross-compatibility issues: side-effects changing, evaluation outputs changing, and evaluation inputs changing. side-effects need not be considered very much since nixlang is supposed to be *pure* and all side-effects that are not part of the store interface must already be considered incidental. evaluation outputs changing can be handled by optional lint or runtime warnings when a versioned evaluation structure passes a semantic version boundary without being annotated as an intentional behavioral leak. evaluation inputs changing is a non-issue because nix plugins and the `ExternalValue` infrastructure already make it impossible to rely on the type system being fully specified at the time an expression is written

inter-file inter-actions

by default language features **must not** be propagated across an unadorned `import` boundary to retain compatibility with existing nix code (eg nixpkgs, which will not be able to switch for quite some time). in some circumanstances it is however *required* to propagate language features across imports to provide a consistent and meaningful interface, eg in the case of a hypothetical `requiredLanguageFeatures` attribute for a flake. to allow for both of these requirements to peacefully coexist we add a new primop:

```
scopedImportUsing
:: { features ? <current language features> :: AttrSetOf bool
     ## ^ language features as would be specified by `use ...;`.
     ## selecting a default-off feature is achieved by setting its key to `true`,
     ## deselecting a default-on feature is achieve by setting its key to `false`.
     ## nesting is not needed because features are identifiers. future changes to
     ## the use interface may extend the type of this set.
   , newGlobals ? env: env :: AttrSetOf Any -> AttrSetOf Any
     ## ^ function to produce the new global environment. it receives the default globals
     ## set for the target expression language features (as calculated form `features` and
     ## the target `use` clause) and produces a new set.
     ## `scopedImport` behavior is recovered by setting this to `const newEnv`.
   }
-> PathLike
## ^ imported path as in `scopedImport`
-> Any
## ^ import result. may be cached, most immediately using the intransparent internal
## object id of the provided features and the globals set. this mimics the beavior
## or `import` in cppnix
```

if the imported expression selects a different set of language features the features specified by `scopedImportUsing` are ignored.

`scopedImportUsing` is available in the `builtins` set and crucially, *can be replaced*. this allows a hypothetical flake implementation to replace both `scopedImportUsing` *and* `import` with its own versions that provide propagation behaviors that might be expected from such a library:

- importing within the same flake simply propagates the language features as-is
- importing across flake boundaries first resolves the language versions used by the imported-from flake, then applies and propagates using these features. if the imported-from flake then imports code from elsewhere this cycle repeats and can eventually restore the language features set to its original value when importing code next to the code importing the importing code

- importing out of a flake boundary (as might be possible in an impure mode) resets the propagated language feature set as if it had never been set in the first place

additionally the current language features might be made available through a builtin value `languageFeatures` by such a replacement of `scopedImportUsing`.

builtins versioning, global versions

a language feature may add or remove elements of `builtins` or the global environment. as mentioned earlier this does not pose a large hazard since evaluation is sufficiently unespecified that this must *already* be expected to happen.

interactions with eg nixpkgs lib

nixpkgs lib (and other libraries) will have to cater to the smallest common denominator when exposing library functions/constants as they do now. if we change a function to have a different prototype and a library reexports it from builtins to its own namespace the language features used by the code importing the library do not matter. to make this problem less unbearable we may want to introduce a concept of library objects and a "use library" directive like eg python `from ... import ...` that can pass language features down to the library being imported in some way.

as a first approximation is would be sufficient to encourage libraries to version their namespaces in such a way that accessing a namespace that relies on language features not present in the current evaluator will fail to evaluate (eg by providing the library itself as a plain set and each version as an attribute that (lazily) imports the specific version of the library needed to fulfill the requested version).

bad ideas for features to remove/change in the first langver

- remove url literals
- remove `with`
- remove `rec` (including `__overrides`)
- remove `let { body = ...; ... }`
- remove `or` contextual keyword, either rework or make a real keyword
- extend `listToAttrs` prototype to also accept 2-tuples instead of name-value-attrpairs
- remove `__sub` and similar overloading

Feature detection

jade: I think we might want to be able to feature detect certain features, e.g. new builtin args, which can be done without, but we would like to know if they are there.

`builtins.nixVersion` has been defanged, which means that an alternate cross impl compatible mechanism needs to be created.

Minimally thought-through proposal

`builtins.features` is an attribute set, where individual attribute names are exposed with the value true if they are implemented by a given implementation.

Attribute names are of the format:

"domainname.feature", for example, "systems.lix.somefeature".

Docs rewrite plans

Here, for now (public edit link): https://pad.lix.systems/lix-docs-planning

Design planning

Nix lang v2

The Nix language unfortunately is full of little and big design accidents. Only so much can be fixed without breaking backwards compatibility.

Our goal is to design an improved Nix language revision, working title "Nix 2". To keep the scope manageable, the first iteration of language improvements will be restricted to be mostly backwards compatible and only require minimal migration effort. This allows us to test the process on a smaller scale, as well as allows us to get the quick and easy improvements out as soon as possible for others to use.

Join the discussion on Matrix: #nix-lang2:lix.systems

The rough action plan is:

1. Fork the grammar and gate its usage behind a feature flag.
2. Use the new grammar as a playground to experiment and implement fixes and improvements to the language, free of any constraints of backwards compatibility.
3. Figure out language versioning and prepare interoperability.
4. Provide a migration path, stabilize the new language, and make it available to users.

Initial language changes

Fixing floats

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/1979
- Confidence: High

Grammar: All floats must have a digit before the `.`. This is a hard requirement for making some of the other proposed syntax changes parse unambiguously in the first place.

Moreover, floating point semantics are currently broken in several ways. They need to strictly follow IEE754 double semantics instead.

Given that such a switch is not easy to make in a safe way, as an intermediate solution all floating point operations should be forbidden, effectively making floating point values opaque to the language.

Set slicing

Partially adapted from https://github.com/NixOS/rfcs/pull/110.

- Status: Draft implemented in https://gerrit.lix.systems/c/lix/+/1987
- Confidence: High

Sets can be sliced using `set.[key1, key2]` and `set.{key1, key2}`. The first returns a projection of the listed keys into a list, the second one a subset. All keys must be identifiers (or string identifiers), scoped to the attribute set.

[TBD: it is unclear as to whether interpolation is useful and how easy it is to implement] Identifiers may be interpolated: `set.[key1, ${key2}]` is equivalent to `[ set.key1, set.${key2} ]`, `set.{key1, ${key2}}` is equivalent to `{ key1 = set.key1; ${key2} = set.${key2}; }`.

Slicing into lists is a replacement for using `with`:

```
dependencies = python.pkgs.[
    arabic-reshaper
    babel
    beautifulsoup4
    bleach
    celery
    chardet
    cryptography
];
```

List and Set unpacking

- Status: Draft implementation in https://gerrit.lix.systems/c/lix/+/1988 and https://gerrit.lix.systems/c/lix/+/1989
- Confidence: Mid

In a list, elements which are lists themselves can be unpacked with the `*` operator. They will be concatenated in-place. `["hello", *list, "world"]` is equivalent to `["hello"] ++ list ++ ["world"]`

This can be easily combined with set slicing. The operator precedence facilitates patterns like the following:

```
  configureFlags = [
    "--without-ensurepip"
    "--with-system-expat"
    *(optionals (!(stdenv.isDarwin && pythonAtLeast "3.12")) [
      #  ./Modules/_decimal/_decimal.c:4673:6: error: "No valid combination of CONFIG_64,
CONFIG_32 and _PyHASH_BITS",
      # https://hydra.nixos.org/build/248410479/nixlog/2/tail
      "--with-system-libmpdec",
```

```
    ])
    *(optionals (openssl != null) [
      "--with-openssl=${openssl.dev}",
    ])
  ];
```

In a set, one can unpack elements like this:

`let baz = { bar = "foo"; }; in { foo = "bar"; *baz.{bar}; }`

This combines well with `optionalAttrs`:

```
{
  meta = with lib; {
    maintainers = with maintainers; [ matthewbauer qyliss ];
    platforms = platforms.unix;
    license = licenses.bsd2;
  };

  HOST_SH = stdenv'.shell;

  *lib.optionalAttrs stdenv'.hasCC {
    # TODO should CC wrapper set this?
    CPP = "${stdenv'.cc.targetPrefix}cpp";
  };

  *attrs;

  *lib.optionalAttrs (attrs.headersOnly or false) {
    installPhase = "includesPhase";
    dontBuild = true;
  };

  # Files that use NetBSD-specific macros need to have nbtool_config.h
  # included ahead of them on non-NetBSD platforms.
  postPatch = lib.optionalString (!stdenv'.hostPlatform.isNetBSD) ''
    set +e
    grep -Zlr "^__RCSID
    ^__BEGIN_DECLS" $COMPONENT_PATH | xargs -0r grep -FLZ nbtool_config.h |
        xargs -0tr sed -i '0,/^#/s//#include <nbtool_config.h>\n\0/'
    set -e
```

```
  '' + attrs.postPatch or "";

}
```

It also allows to have "local" `let` bindings for just some of the keys, without having to move them out of the entire attrset:

```
{
  key1 = "value1";
  *let
    stuff = "foo";
  in
  {
    inherit stuff;
    key2 = stuff;
  };
}
```

As with convential set declaration, duplicate keys are not allowed.

Note that the pattern of `inherit (foo) bar baz;` is equivalent to `*foo.{bar, baz};`.

Pipe operator function application: `|>`

This is being worked on in [RFC 148]

- Status: Implemented and released in Nix and Lix as an experimental feature flag `pipe-operator`
- Confidence: High

In `nixpkgs`, there is the `lib.pipe` function which will allows to write `g f a` as `pipe a [f g]`. Especially with deep nested and complicated data transformations, it makes the code flow from left to right and thus easier to read. Sadly, it is under-used because many people are not aware of it.

The fundamental problem it tries to solve though is that function calls are prefix, i.e. that a data processing chain with multiple entries is read from right to left. (Or, when adding parentheses, from the inside to the out side.)

Therefore, we introduce the `|>` operator. `a |> f |> g` is equivalent to `g(f(a))`.

List indexing

- Status: Not implemented yet
- Confidence: High

TODO link to RFC

Introduce `list.INDEX` on lists as syntax sugar for `builtins.elemAt list index`. ~~`list.${index}` interpolation for dynamic variables also works like it does for attribute sets.~~ To avoid type ambiguities at runtime, `ident.${expr}` is reserved for dynamic attribute access only, dynamic list indexing still requires using `builtins.elemAt`

~~Optional: We could even introduce `.last` `.tail` and `.length` as attributes. Need to think about that.~~ Is a bad idea because of dynamic typing.

Function list destructuring

- Status: Not implemented yet
- Confidence: Mid

The same way as function arguments can be destructured into an attrset with `{…}`, it should also work with lists. Some restrictions:

- Because order matters, arguments cannot have default values.
- Like with the attrset syntax, `...` indicates that the list may have more arguments.
- For now, the `...` must always be at the end. This restriction can easily be lifted some time in the future.
- Unlike in other languages, capturing the rest of the list (for example in head:tail patterns like in Haskell) is not possible because of performance considerations.

This, together with list indexing syntax, will make tuple-style code constructs a first-class citizen of the language. Replacing `nameValuePair` alone is expected to give significant performance gains (short lists are heavily optimized in the evaluator).

Disallow inner-attribute merging

- Status: Not implemented yet
- Confidence: Mid

Nix has syntax sugar for merging attrsets within attrset declarations: `{ a = {}; a.b = "hello"; }` will be fused into `{ a = { b = "hello"; }; }` at parse time.

This feature, only rarely used, does not compose well with other features like `rec` attrsets, leading to unintuitive semantics and potential foot guns: https://git.lix.systems/lix-project/lix/issues/350, https://github.com/NixOS/nix/issues/6251, https://github.com/NixOS/nix/issues/9020, https://github.com/NixOS/nix/issues/11268, https://md.darmstadt.ccc.de/xtNP7JuIQ5iNW1FjuhUccw#inherit-from-scopes-differently-than-inherit

Since these problems would be deeply aggravated by the new set unpacking syntax (defined below), it is best to completely remove this feature altogether. Since it only is convenience syntax sugar, no replacement syntax is necessary.

Expand `inherit` syntax

- Status: Not implemented yet
- Confidence: Low

The `inherit` syntax is adapted to be both more powerful and more consisten with the slicing syntax. The `inherit (from)` is made redundant and deprecated for removal in a future language revision. Inherit can also be used outside of attrsets and let bindings now, and will behave as if it was in a let binding.

```
inherit lib.{mkIf, types};
inherit {
  lib.mkif,
  types.{attrsOf, listOf, string}
};
# Mixing old with new style syntax: Do we want to allow this?
inherit
  lib.mkif
  types.{attrsOf, listOf, string}
;
# This only makes sense within attrsets really
inherit foo;
```

Proper keywords for `null`, `true` and `false`

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/1986
- Confidence: High

I don't know why these are builtins instead of keywords but at this point I assume it's because it was faster to implement.

Proper syntax nodes for all arithmetic expressions

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/1981
- Confidence: High

No more `__sub` and `__lessThan`. These had no reason but laziness to exist in the first place.

`?` and `or` operator

- Status: Draft implementation in https://gerrit.lix.systems/c/lix/+/1990
- Confidence: Mid
- Write `pkgs.foo.bar or default` as `pkgs ? foo.bar : default`, remove the `or` pseudo-keyword
- Unlike with `or`, no attribute access is needed: `value ?: default`

    - `?:` is more powerful than `or`, since it also works outside of `.`
    - [Optional] For consistency, function default arguments use `?:` instead of `?`
- `?:` has a lower priority than function application, which solves a lot of the confusion
- `?` operator for testing attribute set keys becomes a special case of `?:` without default value.
    - This does not change any of the semantics of `?`, but fixes the weird operator precedence as well
- [Optional] Introduce a new operator `.?`, also inspired by Kotlin. `foo.?bar` is equivalent to `if foo != null then foo.bar else null`.
    - C# uses `?.` instead

All line endings must be `\n`

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/1992
- Confidence: High

The current handling of `\r` is so jank that we'd better do without.

CRLF line endings are allowed within the file for Windows compat, but in strings the line endings get consistently normalized to LF only.

All files must be valid UTF-8 text

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/1991
- Confidence: High

The world runs on UTF-8, and most tools these days expect UTF-8 encoded input by default. There's no reason to allow other encodings or invalid byte sequences.

Sane escape sequences for strings

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/2089 and https://gerrit.lix.systems/c/lix/+/2104
- Confidence: Mid
- Escape sequences are restricted from *anything* to `\t`, `\r`, `\n`, `\"`, `\$`, `\\`, `\x…`, `\u{…}`
- `\` followed by a line break escapes it, a.k.a. string continuation escape (Rust)
- `$$` does not escape `$$` anymore, so `$${}` is now a dollar with an interpolation

Indented strings

Don't strip indentation of first line

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/2104
- Confidence: High

The current behavior is just weird, both for single-line strings (commonly used for unquoted `"`) and multi-line strings. The new behavior is also what Haskell does (in its new multiline strings proposal).

Indented strings work with tabs

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/2105
- Confidence: High

Programming languages may be opinionated, but making some features work only with space indentation is crossing a line.

Tabs and spaces can be mixed as part of the string's content, but not for the string's indentation. Indentation is calculated based on the longest common prefix.

Old cruft to remove

https://wiki.lix.systems/link/21#bkmrk-bad-ideas-for-featur

Remove unquoted URLs

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/1982
- Confidence: High

DSL or not, you'll survive typing those two additional extra characters.

Remove `let {}` syntax

- Status: Implemented in https://gerrit.lix.systems/c/lix/+/1980
- Confidence: High

And also the special `body` attribute.

`__override` special attribute

- Status: Implemented in TODO
- Confidence: High

No more magic attributes please. `__functor` is already bad enough.

Fix tokenization rules

https://md.darmstadt.ccc.de/xtNP7JuIQ5iNW1FjuhUccw?view=#token-boundaries-aren%E2%80%99t-real-and-will-hurt-you-cf-nix-iceberg

Status: Partially implemented in https://gerrit.lix.systems/c/lix/+/1984

Autocaller must die

Status: Not implemented

wtf?

```
❯ nix-build --expr '[[[ ({a}: [a]) ]]]' --arg a 'with import <nixpkgs> {}; hello' --no-link
fetching path input 'path:/nix/store/nyysli8lhjf03jgyvrf7mlxrlgnqn9qp-source'
/nix/store/kwmqk7ygvhypxadsdaai27gl6qfxv7za-hello-2.12.1
```

Future language changes

Some changes to the syntax would make large chunks of existing code invalid. These need to be postponed until proper versioning and migration tooling have been figured out.

Comma separated lists (confidence: high)

Currently list items are space separated. This has two major drawbacks:

1. This is inconsistent with most other language syntax features, which use `,` or `;` as item separator.
2. Not having them requires using parentheses around function calls in lists. Those are currently easy to forget, causing confusing type issues for beginners. (This would be less of an issue if we had a type system that could catch the mistake early on …)

Function declaration (confidence: low)

- `args@` now always also contains the default values (are there use cases where one strictly needs this not to be the case? Regardless, that behavior could be manually emulated if necessary)
- The `?` for defaults becomes `?:`
- Functions can also destructure list arguments: `[name, value]: _` as a replacement for `nameValuePair` and and to make tuples a first-class citizen (together with list indexing).
  - Note however that this change conflicts with comma separated lists because having both would cause too much lookahead in the parser.

NUL bytes must be supported within strings

Status: Blocked on rewriting the garbage collector to be compatible

0-terminated strings were a mistake, and we should not make any concessions in the language to implementations who use them. Especially when they're buggy.

Paths

Comments

While we are touching the syntax, let's leave some space here to discuss code comments.

- I like having the distinction between commented out code (syntax highlighting: unobtrusive) and commenting code (syntax highlighting: vivid).

- We should leave some room for semantic doc comments, should they ever come in Nix (TODO link respective discussions)
- There is this concept of "semantic comment" that comments out entire AST nodes. This is immensely useful even though few languages have it. (Caveat: the commented out code must at least be syntactically correct.)

TODO

Design planning

Flake stabilisation proposal

Preface

STATUS: The core team has discussed this proposal, and decided that Lix will be moving in a different direction instead, see Flakes feature freeze. However, if someone wanted to stabilize Flakes in their current form in a minimally invasive way, then this proposal would present one valid way to achieve that goal.

Problem Statement

Flakes are a mess. They are extremely popular (so it's very painful to discard them), but they are also deeply flawed in so many ways, and their compat story is non-existent. Let's go through a few things that are *traditionally* associated with flakes, but they don't need to be.

- 2.4 CLI is obvious. There's no reason why it ever had to be tied so much to flakes. It should be stabilized independently (and probably before flakes)
- Pure eval. Again, never should've been flake-gated
- Installables/runnables abstraction
- Git awareness
- Output schemas (vanilla Nix only has `default.nix` and `shell.nix`, but flakes define more things that are CLI-integrated like formatters, checks, nixosConfigurations etc.)
- builtins.fetchTree deprecation/refactoring/stabilization (TODO: research this more)
- Channels deprecation
- NIX_PATH deprecation

On the last 2 points, see this: https://samuel.dionne-riel.com/blog/2024/05/07/its-not-flakes-vs-channels.html

Overall, flakes did too much at once. We can sort those out one by one. Deprecating NIX_PATH and channels would be a bit tricky, but we can try to re-use flake registries for the same functionality.

Also, flakes have a very bad backwards-compatibility story. Worse than that, we are a CppNix fork, so we want to provide a migration path for a reasonable amount of time. CppNix also completely doesn't have forward compatibility. This means that doing any changes to the `flake.nix` or to `flake.lock` will break flakes for CppNix users. This is really bad, it essentially means we're removing flakes outright, so this isn't something that we want to do.

With those preparations out of the way, we can now get to the flakes.

Flake Components

Flakes themselves have many moving parts.

- `flake.nix` schema: `description`, `nixConfig`, `inputs` and `outputs`
  - `inputs` are super static. Changing anything about them will break a lot of stuff
  - `outputs` is extensible. Changing the predefined attributes isn't great and can break things
  - `description` and `nixConfig` are arbitrary, and can contain bogus info (FIXME: is this true?) We can use this to introduce new functionality without changing other fields, but this is a crime, so let's try to avoid that
- `inputs` URL parser
- `flake.lock` format

The most cursed part is how tightly connected all of that is. `flake.lock` records the inputs to `builtins.fetchTree`. These inputs are parsed from `flake.nix`. The real abstraction here is `inputs` URL parser. Everything else is implementation details that leak out into public interfaces.

So the situation is tricky. Code changes leak out, there are no useful versioning mechanisms, we need to make changes in such a way as to not break upstream, and the adoption is large enough that we don't want to break things. But thankfully, there is a way to deal with it, largely inspired but Opentofu's approach.

The Plan

Stage 0: Fork the Interfaces

First, we must fork the interfaces. Instead of having ossified `flake.nix` and `flake.lock` interfaces that we have no control over - we fork them into different files. Naming is TBD, but let's use `flake.lix` and `flake.lick` in this discussion. More specifically, the procedure looks like this:

- We change all of the flake-related code to use `flake.lix` and `flake.lick` files instead
- We add new internal structures for `flake.lix` and `flake.lick`. For starters, we can have the same structure, but fix the versioning story: `flake.lick` should have SemVer versioning instead of monotonic uint (that would make experimenting and/or forking the format so much easier, because SemVer allows "metadata" info added to the actual version), and `flake.lix` should have the `version` top-level element, too. `flake.lix` is computible, and so it's very non-trivial and depends on many factors: we **must** version it. Also SemVer. The versions have to be managed separately
- We add the migration code. It would look at `flake.nix` and `flake.lock` and create corresponding `flake.lix` and `flake.lick`

This completely changes the compatibility story, because we no longer have to think about upstream usage: we only read, never modify the files the upstream uses. Together with adding sane versioning, we can isolate the versioning to just our project, and make changes (including

backwards-incompatible ones) in a sane manner.

Stage 1: Eating Spaghetti

Next, we need to decouple implementation, `flake.lix` and `flake.lick` from each other. For the latter two, we already have separate version on "manifest" file and the lockfile; it's a good start. Let's discuss what needs to be done to unveil this spaghetti:

- Implementation and `flake.lix`
  - TODO: does it make sense to use `builtins.fetchTree` for inputs, or do we need a separate interface?
  - Parametrized URLs are similar to Terraform, but they have an extreme amount of edge cases to cover. The actual parameters should be separate arguments; no need to try to embed them into a URL
  - `follows` mechanism is horrible. It is extremely rare that you want to respect downstream lockfile in practice. Let's just not do that
  - `inputs` is a special case among special cases; it can't contain any logic, and it also uses C++ code for trust on first use. There's no reason to be so locked into C++: it may be reasonable to expose the toggle to do trust on first use to the user, and have `inputs` be regular NixLang, and possibly even its interpretation be in NixLang (TBD about that)
- Implementation and `flake.lick`
  - The implementation completely bleeds through to the lockfile: it saves all of the arguments for `builtins.fetchTree` and uses that for reproducibility
  - To verify that the contents are actually the same, we need a checksum; `narHash` is the checksum. TBD if we want checksum to have more complex structure (algo/version/w.e. as well as value) or if lockfile versioning is enough
  - Instead of saving all of the arguments for the particular fetcher, we need to have an abstract `version` that we can compute from fetcher contents (TBD if in NixLang or in C++)
- `flake.lix` and `flake.lick`
  - As pointed out above, parametrisation of URLs is a blatant abstraction violation; the interface for parameters in `flake.lix` and `flake.lick` should match
  - `flake.lix` should contain "version range", and `flake.lick` should contain the "resolved version". The entire specifics are tricky for e.g. git

Stage 2: Improving the Interfaces

There's a lot that can be done here. Cross-compilation, version resolution, newer fields, and more - all of that belongs to this stage.

Stage 3: Maintenance

This path is backwards-compatible throughout, so we can maintain an upgrade path without much issue. We can have a directory with subdirectories for each major version. Those subdirectories will also handle upgrading the lockfile; then, we'll always have a path to upgrade from CppNix flakes to

Lix flakes: you just execute all of the existing upgrades in order.

Truly backwards-incompatible changes would be adding absolutely necessary metadata, without which the previous version is useless. `npm` has this: their oldest lockfile (you can call it "v0") didn't have a `version` field, and it also didn't record checksums. It simply doesn't contain any metadata that better lockfiles do, so the only way to move forward is to extract whatever you can from it, and generate a newer-version lockfile from scratch with that data.

As long as we only need the `version` and `checksum` (which seems to be the case), the only source of breaking changes I see is security vulnerabilities. If e.g. NAR hashing is proven to be vulnerable - it's probably for the best to not rely on the already existing hashes at all.

Notes

This plan doesn't have to be executed as sequentially as it's described. Really, we can have something like a from-scratch rewrite for flakes and include it in the first `flake.lix`+`flake.lick` versions. Or we could only add the versioning code. Or we could add versioning and version resolution, or versioning and cross-compilation, or literally anything else, as long as versioning is definitely present.

Appendix A: Flakes are a broken abstraction

Some parts of this were already mentioned, but flakes are pretty broken on fundamental levels. The lockfile essentially containing arguments for a C++ function are an example of that. This isn't an abstraction pretty much by definition - it does not abstract away the details. A good example of a lockfile is `version = 3` for Cargo:

```
[[package]]
name = "anstyle-wincon"
version = "3.0.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "61a38449feb7068f52bb06c12759005cf459ee52bb4adc1d5a7c4322d716fb19"
dependencies = [
 "anstyle",
 "windows-sys 0.52.0",
]

[[package]]
name = "anstyle"
version = "1.0.7"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "038dfcf04a5feb68e9c60b21c9625a54c2c0616e79b72b0fd87075a056ae1d1b"
```

```
[[package]]
name = "windows-sys"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "282be5f36a8ce781fad8c8ae18fa3f9beff57ec1b52cb3de0789201425d9a33d"
dependencies = [
 "windows-targets 0.52.5",
]

...
```

We more or less have the `source` (as bad as it is implemented in flakes), and we have `checksum` (which is NAR hash) and `name` - but we are missing the `version` abstraction. There's some complexity to unpack here (for example, it isn't trivial to say what "version" means for a tarball or a filesystem path), but flakes don't even try - they just completely ignore the need for this abstraction, and use C++ implementation details instead.

Another issue is the general confusion about what flakes are supposed to be, and how Nix plays into that. Nix is a lot of things, but the way it ended up working out is that Nix is a builder abstraction: you use Nix to build packages, and the packages may have dependencies, and yadda yadda. But because Nix is so general, it can be used to build a "meta-package" of all "installed packages", and you can also use it to build OS configs, so you can essentially build a system meta-package. The whole NixOS system is just a big meta-package that consists of other packages.

This is a blessing and a curse: expressing the entire system as one package is cool and has its advantages, but this is also a very hacky way to use the build system that is Nix. It's like using the Makefile to configure your system. CppNix developed a lot of stuff to keep this approach going: channels, NIX_PATH, nixos-rebuild scripts, nix-env and other things are all used to make the experience more tolerable. So it's a lot of hacks on top of a rather quirky way to use the build system. The biggest example on how it manifests is NixOS configuration: we use it to create different build manifests for the resulting system, and we don't have other ways to interact with the system, like a package manager. This is a tough place to be in: the NixOS approach has a lot of really good properties, but it's also inherently limited because the build system is used as a configuration engine and a package manager.

Flakes are confused and stupid because they try to be a package manager for Nix, but they are a shitty package manager, and they also don't even try to resolve many of the hard questions that arise from using Nix itself as a package manager. They don't have a concept of "libraries", so everyone still uses Nixpkgs lib. They don't have version resolution, or a concept of versions. They don't really integrate with Nix profiles, they don't integrate with NixOS, they don't draw good boundaries between what different units of NixLang code do: provide library functions, create packages, create configuration, or whatever else.

There are only three package manager things that flakes actually tried to do: it's installables/runnables abstraction (just barely counts), manifest+lockfile usage (the idea itself is good but impl is awful), and defining a schema. Everything else doesn't address the issue at hand in the slightest: some of the ideas are good and should be decoupled from flakes, and some of the ideas are awful.

Regarding installables/runnables: it's a step in the right direction for drawing boundaries between packages, libraries and configs. But the way it's implemented is also bad. The definition for installables is a huge nothing-burger: basically, an "installable" is a store path or a thing that resolves into a store path (this is more or less what's said in the glossary). This definition gives you exactly nothing, and reminds of a horribly ill-defined "derivation" stuff ([1], [2], [3], [4], [5], [6]). The actually useful thing here is "runnables", which are things you can `nix run`. It's also barely defined (mostly just using the store path and appending `/bin/<name>` to it lol) and absolutely isolated from all of the Nixpkgs and NixOS work, so it ends up being completely useless in practice.

This document mainly goes over how to unbreak the flakes and make them work at all, but creating coherent abstractions on top of the unbroken flakes is a whole other dimension of pain and integration work. In practice, integrating flakes into Nix properly will end up requiring "owning the stack" or close to it: being very free to refactor and unbreak many hacks in Nixpkgs and NixOS.

Appendix B: Some thoughts on post-stabilisation world

Something that would make a lot of sense is drawing further boundaries between units of NixLang code. Flakes would be a NixLang package manager, and like there is a distinction between "binaries" and "libraries" in proper programming languages like Rust, there would be "flake types" for NixLang. Some easy examples include: "NixLang library", "Nix plugin", "configuration", "package manifest", "binary/runnable", "generic package". Just using those "flake types" for manifests doesn't do much good: there needs to be tight integration with Nixpkgs. In fact, Nixpkgs might start composing flakes instead of just NixLang code directories: this might be a great change for the better.

To give some examples on how integrating flakes would look like, we can take inspiration from dreams page. Let's discuss flake-related items:

- Using flakes as code libraries
  - This is just one possible usage! We want to draw boundaries between NixLang units. "Flake as NixLang library" is perfect: if flakes are units for package management, distributing NixLang libraries as flakes makes perfect sense
    - This would go really hard with bytecode compilation/WASM/etc., because now we'd be able to distribute high-performance library functions written in languages that aren't NixLang
- Creating subcommands - it's a little orthogonal to the flakes discussion, but some custom subcommands could be comething like "Nix plugins" and distributed as flakes

- `nix profile` working on a mutable manifest is a perfect integration example: the "installed packages" manifest would be a unit of NixLang code, and so it makes sense as a "flake type". The coolest thing here would be to use flake resolution to have transparent interaction with remote flakes
- fs builtins are very relevant for the discussion, too: Nixpkgs and NixOS are full of filesystem manipulation evil, and much of it should use a dedicated "flake type"

So basically, flakes subsystem needs to be an actual package manager with actual units (flakes). Then, flakes will actually make sense and be good, and we'll finally be able to have nice things, like not having Nixpkgs be a gigantic fs tree with dubious abstractions. I mean, pointing to Rust again (because it's good): Cargo doesn't just operate on fs trees and let you handle the rest like an old-school thing like Nix forces you to do, Cargo has many abstractions to decouple fs tree from things you care about: workspaces, crates, modules, etc. When flakes become Cargo and give us proper composition - we'll know we've done a good job.

Design planning

Observability and Protocol Design

jade: I think that we should start protocol design by thinking about who needs what information, which is most cleanly hit by looking at how observability architecture looks. Let's get cracking on what observability we need/want in Lix.

Context

Old profiling pad for the Nix language: https://pad.lix.systems/lix-profiling. This might want/need to be a different system than the overall observability architecture since it affects the evaluator primarily and has specialized needs (e.g. high performance).

Old protocol investigation pad: https://pad.lix.systems/lix-protocol-investigation

Discussion

Let's have this discussion in a pad here so we can have good live editing:

https://pad.lix.systems/lix-observability

Design planning

Replacement CLI design & Profiles

Draft pads:

- https://pad.lix.systems/lix-cli-design
- https://pad.lix.systems/lix-profiles

Design planning

Nix bootstrapping

Pad:

https://pad.lix.systems/VjA-WMSQS42dh-ghL98Uow

Design planning

Improving IFD

- Discussion pad: https://pad.lix.systems/Xd5Xyt5cToyYxIv-lNdehA
- Further reading: https://jade.fyi/blog/the-postmodern-build-system/

pennae's idea:

```
magic_ifd_primop = \f i -> import f >>= i
# nix:
importAndThen (res: <stuff>) some-ifd.outPath
```

Observation: a magic IFD primop that immediately does something to the result of an import can be "thunked" (thus processed in the background) and offer an "interesting" fix to IFDs. We cannot just make `import` be lazy because of the evaluator internals.

Subsequent observations:

- Explicit monads in Nix may be interesting
- Explicit monads are not necessary if everything "monadic" is related to continuations
- Thunked continuation "entities" / "objects" are sufficient

Why is this interesting?

The capability of forking / parallelizing the work.

Questions:

- jade: how do you generalize this over all the ways that one can consume things via IFD (readFile, etc)?

Design planning

Flakes feature freeze

The core team has decided to freeze the Flake feature set and semantics at its current point, excluding bugfixes. No new features will be added to Flakes for the time being, including (but not limited to) new input types, new features for existing fetchers and input types, or flake-centered changes to the evaluator.

Our long-term plan is to move Flakes out of the Lix code base into a separate plugin, all while cleaning up the structural foundations and building a new platform for any third party to build flake-equivalent functionality upon. Until then, we intend to continue our support of Flakes in their current form. They will remain an experimental feature and explicitly *not* be stabilized.

The main reason for the freeze is that the Flakes implementation code is *extremely* brittle, causing significant maintenance overhead. With our stated goal of refactoring the Lix code base and moving Flakes out of it, any changes to Flakes themselves would make them a moving target, and greatly increase the risk of breakage or regressions. Moreover, Flakes provide virtually no versioning mechanism to safely introduce major changes, and while Flakes are *stated* to be "experimental", their current wide adoption thwarts our ability to rapidly iterate.

Design issues of Flakes

As many other people have exhaustively written about, Flakes are sprinkled with design deficiencies. Many of those issues are related to versioning and future-proofing, making it significantly harder to fix Flakes in an iterative way without causing disruption. The list presented here is not meant to be exhaustive and represents the main issues motivating the freeze of Flakes in Lix.

- The Flake schema is effectively unversioned and thus impossible to evolve without breaking existing users. This necessitates extensive compatibility checks in the implementation, most of which very quickly becomes effectively untested and thus a regression hazard.
- The Flake lock file schema is versioned, but its version never changes even when its semantics change. Adding new input types or how existing input types are interpreted should imply a change to the lock file schema to ensure that older version do not misinterpret a lock file or fail with cryptic errors.
  - The design of flake locking is unsound: It compares the lock entry to the current lock entry, even if the thing in the lock file is fetchable but just not what *this* nix implementation would have generated for the same original input.
  - This means that *any* divergence in libfetchers input format or lock file format causes incompatibility between nix implementations. There are already divergence bugs, e.g. https://git.lix.systems/lix-project/lix/issues/841, https://git.lix.systems/lix-

project/lix/issues/520.
    - In the current system, the only way to fully prevent all compatibility bugs is to have all implementation agree to a common specification. For a multitude of reasons, this is not going to happen. There is a (stalled) effort of standardizing at least `fetchTree` here: https://github.com/nix-community/fetchTree-spec.
- Using the Nix language to specify Flake inputs requires very invasive changes to the evaluator to ensure that Flake input definitions are "trivial", i.e. pure attribute sets that do not contain computations. Instead, inputs should be defined in a format that does not require evaluating code, much like the lock file.
- The Flake interface, especially the devshells, are tied very strongly to nixpkgs. while `nix-shell` has always been tied to nixpkgs to an extent (`nix-shell -p thing` internally runs `{...}@args: with import <nixpkgs> args; (pkgs.runCommandCC or pkgs.runCommand) "shell" { buildInputs = [ (thing) ]; } ""`), this binding was not so strong that it could not be used freely by non-nixpkgs package sources. `nix develop` on the other hand hard-codes nixpkgs-specific internals in many places and is all but useless on derivations that do not use nixpkgs' `stdenv`.
- Cross-compilation support is completely absent and cannot be added without changing the Flake schema, which is (as noted above) all but impossible without introducing new attributes (which would be admonished by older `nix flake check`) or outright breakage.
- Flakes do not compose well. Reusing an expensive nixpkgs instantiation can be impossible because flakes do not allow passing parameters to the implicit instantiation created by `legacyPackages`, and even when reusing an instantiation is possible the final graph of a flake-built package may be inconsistent if multiple versions of nixpkgs were used in the flake input graph.
- Using the same source in multiple versions is not only easy to do by accident, it is hard to *not* do even with intent. The mechanism for this, flake input follows, are spooky action at a distance that allow a flake to override inputs of its dependencies even if those dependencies do not expect this, possibly causing silent breakage of packages.
    - c.f. https://github.com/mercuryTechnologies/unused-flake-input

Design planning

xattrs feasibility to supplement the SQLite database model

xattrs were imagined as a way to supplement/replace the SQLite database model.

Unfortunately, xattrs are not useable on all platforms we may care about and not by all filesystem our users.

Additionally, we need to plan for the workload size we have, i.e. large dependency chains that can spans over MB of store paths.

We collect information about this idea here.

OS support matrix

Linux filesystem support matrix and limitations

- bcachefs: 1994 bytes (@raito)
- ext4: 4040 bytes (@pennae)

Current consensus is that xattrs are not useable.

Technical notes

Those are a collection of technical notes about a specific topic in Lix.

Pointer equality

Introduction

This page dives into the concept of pointer equality, its role in Lix, and provides an outlook on how it is implemented and utilized.

For more in-depth information, refer to the resource on https://snix.dev/docs/reference/nix-language/value-pointer-equality/ from the Snix project.

Some background on pointer equality

This section provides a foundational overview of pointer equality, specifically in the context of nonstrict languages like Nix.

Why pointer equality is used?

Pointer equality is a "low-level" operation that checks whether two values are represented by the same memory reference, that is, whether they point to the same object in memory. In a lazy functional language implementation (such as Lix, Haskell or Lean), pointer equality is often much cheaper than structural equality, which may require traversing the entire data structure. Traversing terms can be slow (exponential) when doing structural comparisons, (recursive) pointer equality offer a linear time approach to structural comparison.

In Nixpkgs, `lib.systems.equals` implements the expensive structural comparison. In Lix, `==` relies on pointer equality in underspecified fashions.

The relation of pointer equality and call-by-need evaluation strategy

Pointer equality is closely tied to the evaluation strategy employed by an interpreter.

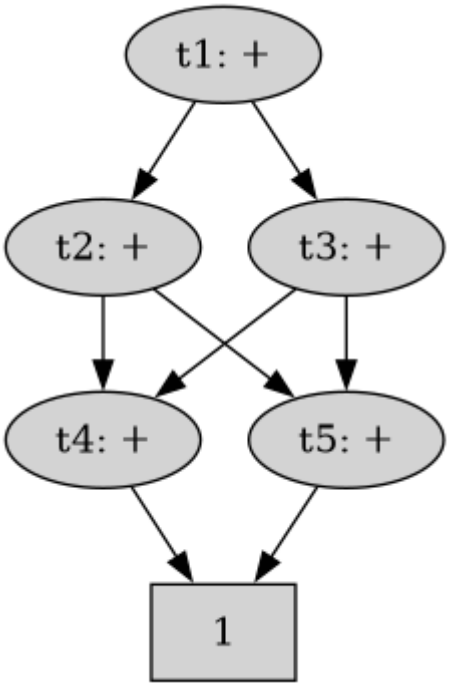
There are usually three main evaluation strategies:

- Call-by-value
- Call-by-name
- Call-by-need

A **call-by-need** strategy (also known as **lazy** or **nonstrict** evaluation) evaluates values **only when needed**, unlike **call-by-value**, which evaluates values eagerly, or **call-by-name**, where values are re-evaluated each time they are used.

In call-by-need, values are evaluated **at most once**, and the result is reused whenever the value is required again.

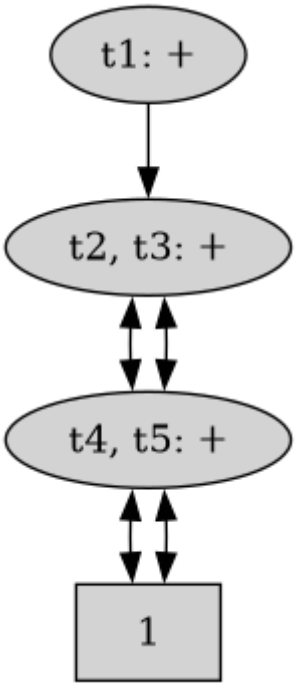
This has important implications for **pointer equality**. To have effective pointer equality, it's crucial to optimize when sharing occurs between structurally equal expressions. For example:



In this diagram, the terms represent a call to the binary addition operation `+` and one operand `1` is at the bottom. Notice that terms at the same level evaluate to the same result.

If we ask whether `t_2` is equal to `t_3` without considering sharing, pointer equality would initially return `false`, as it compares only the "tip" of the structure. However, through recursive pointer equality, the comparison can descend into the structure, testing further until it finds that the two terms ultimately share the same operand (`1`).

That being said, an interpreter can optimize this by merging shared substructures:



This maximally shared structure leads to more efficient pointer equality checks between terms that are actually equal, reducing redundant evaluations.

But, these representations are internal to the evaluator and should not lead to a language evaluating in a way or another, they are purely about optimization.

Unfortunately, pointer equality can be a primitive for detecting shared structures in structurally equal objects, causing the internal details of the interpreter to leak into the language.

The situation in Lix

Lix is a **call-by-need** evaluator of a nebulous — underspecified — language, often called Nix or Nixlang.

General comparison theory in Lix

If we ignore pointer equality and focus on semantic equality (i.e. whether the contents of the values are equal), we can draw this summary table since Lix 2.90:

Type (`v1.type()`)	Structural comparison with same type?	Comparison semantics	Notes
`nInt`	Yes	Compare integer values (`==`)	Exact numeric equality
`nBool`	Yes	Compare boolean values (`==`)	True/false match
`nString`	Yes	Compare string contents (`==`)	Structural string equality
`nPath`	Yes	Compare underlying string contents (`content->str()`)	Path equality is based on string value

Type (`v1.type()`)	Structural comparison with same type?	Comparison semantics	Notes
`nNull`	Yes	Always `true`	All null values are equal
`nList`	Yes	Compare element-wise recursively	Compare elements one by one
`nAttrs`	Yes	Compare attribute names and values recursively	Twist: Derivations are compared by their `outPath` attribute
`nFunction`	**No**	Always `false`	Functions always compare not-equal
`nExternal`	Yes	Delegate to `*external == *external`	Uses external type's own equality definition
`nFloat`	Yes	Compare float values (`==`)	Standard floating-point equality (e.g., `NaN != NaN`)

The case of functions

As shown in the previous table, functions always compare as **not equal**. However, when using pointer equality, `f == g` can evaluate to **true**.

In Lix, pointer equality introduces an **irreflexive** binary relation for functions: that is, `(x: x) == (x: x)` is always **false**.

There are several reasons why functions consistently return **not equal**:

- When defining a function like `(x: x)`, it creates a **new** identity each time. Binding this function and reusing it elsewhere preserves this identity. Thus, pointer equality can cause comparisons to return **equal**.
- Comparing functions in a lambda calculus-inspired language can be complex and is not a feature we intend to offer to users (for a deep dive into the topic, see this paper if you're still interested). It has limited practical value.

Some members of the Lix core team currently believe that allowing function comparisons at all was a **grave mistake**, as it encouraged users to mix **code** with **data** by placing functions into data structures.

The team envisions a future where function comparisons are entirely removed from the language, making any function comparison an error. The next section explores why achieving this goal will be challenging.

Where is pointer equality used in the Lix ecosystem?

The Lix codebase refers to a mysterious `builderDefs` to explain why pointer equality between attribute sets (including those with functions) is supported. This is illustrated in the following code:

```
    /* !!! Hack to support some old broken code that relies on pointer
       equality tests between sets.  (Specifically, builderDefs calls
       uniqList on a list of sets.)  Will remove this eventually. */
    auto pointerEq = [&] { return v1.pointerEqProxy() == v2.pointerEqProxy(); };
```

In reality, `builderDefs` has long since disappeared, but this comment was never updated to reflect that.

The primary use case for this specific form of pointer equality comes from the cross-compilation and platform machinery in nixpkgs. For example, the expression `pkgs.stdenv.hostPlatform == pkgs.stdenv.buildPlatform` performs a **function** pointer equality comparison.

To elaborate: `pkgs.stdenv.hostPlatform.emulator` is a function, and there are other functions within the set.

As a result, any changes to this semantics could cause issues within the Nixpkgs platform machinery, potentially leading to unexpected behavior, such as entering cross-compilation mode when it shouldn't. More details on this will follow in the next sections.

The semantics of pointer equality in Nix 2.18

As of Nix 2.18, semantics for pointer equality were to apply it for any pair of values, no matter their types, i.e. in pseudo C++:

```
bool EvalState::eqValues(Value * v1, Value * v2, const PosIdx pos, std::string_view errorCtx)
{
    forceValue(v1, pos);
    forceValue(v2, pos);

    // This is where the pointer equality intervened by comparing the addresses of the Value
pointers.
    if (v1 == v2) return true;

    // Special case type-compatibility between float and int
    if (v1.type() == nInt && v2.type() == nFloat) {
        return v1.integer().value == v2.fpoint();
    }
    if (v1.type() == nFloat && v2.type() == nInt) {
        return v1.fpoint() == v2.integer().value;
    }

    // All other types are not compatible with each other.
    if (v1.type() != v2.type()) return false;
```

```
    // [snip]
}
```

The semantics of pointer equality in Lix `80654b84b610f4c0622dd10f0af78a8a2ce97048`

Commit link: `80654b84b610f4c0622dd10f0af78a8a2ce97048`

This commit introduced a higher level of sharing of the expressions in our internal implementation, while we were careful not to induce a change in visible semantics, we decided to reduce the eligibility of the pointer equality check above to only: list, attributes and external (plugin) objects.

This resulted in various evaluation regressions, e.g. `nix eval "github:nixos/nixpkgs?rev=a999c1cc0c9eb2095729d5aa03e0d8f7ed256780#pkgsCross.gnu64.bitwarden" --no-eval-cache`.

The root cause lies in the following and is related to function comparisons:

```
with rec {
  a = {
    f = x: x;
    meow = true;
  };
  b = a // {
    meow = true;
  };
};
a == b
```

returned false.

This pattern occurs more generally in the Nixpkgs module system when deciding whether a package set is in cross mode or not:

```
let
  inherit (import <nixpkgs> { }) lib;
in
(lib.evalModules {
  modules = [
    (
      { config, ... }:
      {
        options = {
```

```
          buildPlatform = lib.mkOption {
            type = lib.types.either lib.types.str lib.types.attrs;
            apply = lib.systems.elaborate;
            default = config.hostPlatform;
          };
          hostPlatform = lib.mkOption {
            type = lib.types.either lib.types.str lib.types.attrs;
            apply = lib.systems.elaborate;
            default = "x86_64-linux";
          };
          isCross = lib.mkOption { type = lib.types.bool; };
        };
        config = {
          isCross = config.buildPlatform == config.hostPlatform;
        };
      }
    )
  ];
}).config.isCross
```

This can be tested this way as well:

```
nix-repl> nixosConfigurations.nixos.config.nixpkgs.hostPlatform ==
nixosConfigurations.nixos.config.nixpkgs.buildPlatform
```

(Credits to aloisw for the code snippets.)

Circling back to "where is pointer equality used?", the answer is that pointer equality is very used with functions in attribute sets.

The semantics of pointer equality in Lix 2.94.0

Our plan is to implement https://gerrit.lix.systems/c/lix/+/4556.

This makes again many types eligible to pointer equality checks, repairing the previous evaluation issue. A new behavior occurs now due to sharing changes:

```
let a = { f = x: x; }; in a.f == a.f
```

is now true.

At the time of writing, we believe this is the right thing to do as this object `f` has the correct identity now.

Open questions

- How does making more objects have an identity is going to hold in the future of the Nixlang?
- What about the goal of evaluating old Nixpkgs and providing stability?

Technical notes

Darwin debugging for the Linux developer

These technical notes are meant to explain to a Linux developer what are the equivalents to debug certain things over Darwin.

Key differences from Linux

Linux tool	macOS equivalent	Purpose
`strace`	`dtruss` (1), `strace-macos`	System call tracing
`perf`	`sample`, `spindump`, `Instruments` (2)	Performance profiling
`ss`	`netstat -nv`	Socket statistics
`iostat`	`iostat -Id`	Disk I/O statistics
`lsof`	`lsof`	Open file listings
`/proc`	`vmmap`, `heap`, `lsof`	Process/memory information
`blktrace`	`fs_usage -wf filesys`	Trace real-time IO activity

(1): `dtruss` is unavailable on systems with System Integrity Protection enabled (SIP). Lix CI machines have SIP enabled.

(2): `Instruments` is a graphical application that requires a remote desktop and cannot be used over SSH.

Getting flamegraphs out of `sample`

Combine https://github.com/brc7/flamegraph-sample/blob/master/stackcollapse-sample.py with https://speedscope.app/.

Inspecting memory allocations

`heap` can give a list of allocations sorted by sizes when it can recognize the type in-memory (doesn't always work for everything).

`iostat` tips

`iostat -Id` on macOS will return as the first row the cumulative data *SINCE* the first boot.

`xfrs` := IO operations per second.

Listing file descriptors

`/proc` doesn't exist on macOS, therefore, `lsof` is the only thing that can list file descriptors and let you understand the relationships.

Tracing network activity

`fs_usage -wf network` can trace real-time network activity on file descriptors.

Checking Internet quality

`networkQuality -v` will dump an Apple-infrastructure Internet connection check for bandwidth and latency.

Release names

Release names are the names of frozen desserts. There's a list on Wikipedia of frozen desserts, but of course, others can be added. The purpose of release names is that they are cute, and they are not necessarily picked for any reason.

Used release names:

- 2.90 "Vanilla Ice Cream"
- 2.91 "Dragon's Breath"
- 2.92 "Bombe glacée"
- 2.93 "Bici Bici"
- 2.94 "Açaí na tigela"

Here are some ideas of release names:

- Kulfi
- Tartufo
- Soft serve (with some flavour?)
  - Cherry Dip Soft Serve Vanilla
- Italian ice
- Gelato
- Granita
- Frozen yogurt
- Frozen custard
- Sorbet
- Frozen chocolate banana
- Watermelon slush
- Freeze pop
- Baked Alaska
- raketijsje (but only if it comes with a Dutch translation)
- Spaghettieis