Lix contributors

Information you might want if you're thinking about contributing to Lix

Lix Beta Guide

Thank you for choosing to help us in our beta!

There is a lot of work-in-progress documentation and a lot of it is work in progress or awaiting move to the wiki. Our apologies for this state, let us know if there is something you need.

If you run into any friction, please let us know. We would like to hear all your complaints, and this beta is as much about testing our processes as it is about testing the software.

Getting yourself set up with an account (if desired)

Sign in with GitHub on https://identity.lix.systems.

Note that your email will be visible on Gerrit if you use it, so change it on https://identity.lix.systems if necessary.

A brief tour of the Lix systems

See Information Organisation for where information is.

The Lix sources are developed on Gerrit, built with Buildbot, and released on a Forgejo repo.

Contributor documentation for the project is maintained on this wiki. FIXME(jade): a lot of it is awaiting migration onto the wiki from the private pad system, see tracking issue.

Status

We are confident enough to run nightly builds on the machines we care about. We expect Lix to have, generally, fewer bugs than Nix 2.18, which is what you probably already have.

Notable changes:

- REPL is much better
  - The debugger is no longer missing variables
  - `--debugger-on-trace` gives you a debugger for `builtins.trace`
  - The `nix repl` startup messages have been shortened
- Many errors now print the value in question (`cannot coerce set to string`, `expected list but got string`, etc.)

- Many bugs have been fixed, in general:
  - `nix eval nixpkgs#hello` now gives the derivation path instead of hanging
  - `nix-env -qa` lists all attribute paths leading to a package, instead of missing some
- `nix flake check -v` prints what is being checked (and now we notice how slow that command is)
- Stack overflow is now caught properly
- Performance improvements (8-20% faster than 2.18)
- Correctness (inherit-from laziness fixed)
- `nix repl` can `:doc` library functions.
- `nix repl` can accept overlays as config files, see `repl-overlays` release note in the sources.

We have an installer, but it is not easy to use for HEAD builds. We also have a binary cache but we need to do more work to make it actually hit for building HEAD.

On NixOS/nix-darwin

Use the overlay: https://git.lix.systems/lix-project/nixos-module

Please file bugs if this explodes the build of tooling you use, we can fix it in the overlay.

Flakes

Add Lix to your system configuration like so:

```
{
 inputs = {
  lix = {
   url = "https://git.lix.systems/lix-project/lix/archive/main.tar.gz";
   flake = false;
  };

  lix-module = {
   url = "https://git.lix.systems/lix-project/nixos-module/archive/main.tar.gz";
   inputs.nixpkgs.follows = "nixpkgs";
   inputs.lix.follows = "lix";
  };
 };

 outputs = {nixpkgs, lix-module, lix, ...}: {
  # or equivalent for darwin
  nixosConfigurations.your-box = nixpkgs.lib.nixosSystem {
   system = "x86_64-linux";
   modules = [
```

```
        ./machines/your-box
        lix-module.nixosModules.default
      ];
    };
  };
}
```

You can then update it with `nix flake update lix; nix flake update lix-module`.

Not flakes

Also supported.

Add inputs for `git+https://git.lix.systems/lix-project/lix` and `git+https://git.lix.systems/lix-project/nixos-module` to your preferred pinning tool.

Use in a NixOS module: e.g. `imports = [ (import "${your-pinning-thingy.lix-nixos-module}/module.nix" { lix = your-pinning-thingy.lix; }) ];`

Niv

Add the sources for the module and Lix itself, using `ssh://` after registering your keys with git.lix.systems:

```
$ niv add git -n lix-nixos-module --repo 'https://git.lix.systems/lix-project/nixos-module'
$ niv add git -n lix-lix --repo 'https://git.lix.systems/lix-project/lix'
```

Then, import the Lix NixOS module:

```
  imports = [
    (import "${sources.lix-nixos-module}/module.nix"
      (let lix = sources.lix-lix.outPath;
      in {
        inherit lix;
        versionSuffix =
          "pre${builtins.substring 0 8 lix.lastModifiedDate}-${lix.shortRev}";
      }))
  ];
```

On other Linux or on macOS

Currently we are still working on the installer (see tracking project). It is possible to convert an existing Nix install to Lix.

flakey-profile

This is **experimental**. Some people have successfully used it on macOS. We have tested it on an Arch Linux system installed a long time ago with the shell-based installer, and it works fine. This method works by replacing your system profile with one that is built by simple Nix code with flakey-profile.

You can rollback if it blows up by `/nix/var/nix/profiles/default-{SECOND-HIGHEST-NUMBER}/bin/nix-env --rollback --profile /nix/var/nix/profiles/default`.

Clone `https://git.lix.systems/lix-project/nixos-module.git`, then, inside it, run `sudo nix run --extra-experimental-features 'nix-command flakes' .#system-profile.switch`.

Finally, run `sudo systemctl daemon-reload && sudo systemctl restart nix-daemon`, or, for macOS:

```
sudo launchctl stop system/org.nixos.nix-daemon
sudo launchctl enable system/org.nixos.nix-daemon
sudo launchctl kickstart -k system/org.nixos.nix-daemon
```

Restoring a broken install after a macOS update

After updating macOS, you may get error messages like these:

```
~/.nix-profile: no such file or directory
/nix/var/nix/profile/default: no such file or directory
error: cannot connect to socket at '/nix/var/nix/daemon-socket/socket': Connection refused
```

You can fix this by opening "Disk Utility" and manually mounting the `Nix` Volume again. Then, run these commands to re-install the lix daemon:

```
sudo launchctl load /nix/var/nix/profiles/default/Library/LaunchDaemons/org.nixos.nix-daemon.plist
sudo launchctl kickstart -k system/org.nixos.nix-daemon
```

Manually, with `nix profile`

We::Qyriad have used these steps **on macOS** has it has **seemed** to work, but we would recommend flakey-profile over it.

```
$ sudo -H --preserve-env=SSH_AUTH_SOCK nix --experimental-features 'nix-command flakes' profile install --profile /nix/var/nix/profiles/default git+ssh://git@git.lix.systems/lix-project/lix --priority 3
```

- `--preserve-env=SSH_AUTH_SOCK` assumes that your SSH agent is important to access the Lix repo
- `--priority 3` makes it symlink Lix over your existing Nix install

If you then run `sudo nix --experimental-features 'nix-command flakes' profile list --profile /nix/var/nix/profiles/default`, you should get output similar to this:

```
Index:              0
Store paths:        /nix/store/8ma7xas2nb0i3lq8mm7fpgalv94s8pzh-nss-cacert-3.92

Index:              1
Store paths:        /nix/store/53r8ay20mygy2sifn7j2p8wjqlx2kxik-nix-2.19.2

Index:              2
Flake attribute:    packages.aarch64-darwin.default
Original flake URL: git+ssh://git@git.lix.systems/lix-project/lix
Locked flake URL:   git+ssh://git@git.lix.systems/lix-
project/lix?ref=refs/heads/main&rev=98b497a1a43a4ff39263ed5259f12c5d00b4d8c0
Store paths:        /nix/store/8040hxr4rr8bpb5yp4b48709x3qs4bwb-nix-2.90.0
```

You may then use `sudo nix --experimental-features 'nix-command flakes' profile remove --profile /nix/var/nix/profiles/default 1` to remove your original installation of Nix. This is (probably) optional.

Verification

You should now get something like the following:

```
~ » nix --version
nix (Nix) 2.90.0-lixpre20240324-f86b965
```

Information organisation

Lix has a lot of information as a project, and we want to make it accessible in a way that it can be found later if necessary.

There are various tools for keeping information in the project, and they have different purposes

Chat

Chat is good for:

- Information that will be meaningless in hours
- Ephemeral discussions, in general

The chat is expected to move way too fast to follow. As such:

- Don't write things in chat that you expect to be found later
- If discussions of design happen, write them down, at least by copy pasting into a pad and adding the pad to the index
- If tips and tricks are discussed, please write them down
- Please do reviews on Gerrit so they are archived, rather than in chat
- Write things down in the log if they are expected to be found

The log pad (https://pad.lix.systems/lix-event-log [private])

The log pad is intended as a tool to communicate what is going on in general, without having to have everyone pay attention to chat too much.

It should be used for:

- Updates on what we are working on, in addition to chat

It should not be used for:

- Actually notifying people, necessarily

Pad (https://pad.lix.systems [private])

We anticipate that the pad service will be semi-permanently private by default, since it doesn't support ACLs.

The pad is good for:

- Sketching out drafts of documents that aren't ready yet
- Planning private things
- Generally getting people on the same page about things in active design, making what might be meeting notes, or similar.

The pad is not good for:

- Information that should be available to users (unless it is planned to move)
- Information that is not actively changing

N.B. For users who aren't in the Lix core team, the service returns 500 when you attempt to login. This is a known issue that can't be fixed.

Wiki (https://wiki.lix.systems)

The wiki is good for:

- Development process information, like you would find on https://rustc-dev-guide.rust-lang.org/ for the Rust compiler, for instance.
- Design documents

The wiki is not good for:

- User facing documentation
- Documentation that deserves to be reviewed
- Actively writing a document in real time with others

Markdown files in the Lix repo

Markdown files in the Lix repo are good for:

- Maintaining things that are tied directly to the code
- Documentation that needs to be reviewed
- User facing documentation

Markdown files in the Lix repo are bad for:

- Quickly iterating on things
- Design documents

Forgejo issues (https://git.lix.systems)

Our primary issue tracker is Forgejo issues.

We are currently attempting to use the Forgejo project boards feature to communicate what people are working on; it may be replaced with better Kanban software in the future. When making project boards on Forgejo, make them on the `lix-project` organisation unless they are strictly contained within one project.

The issue tracker is good for:

- Actionable work
- Bugs

The issue tracker is not good for:

- Dreams or otherwise not actionable information that is a long term goal
- Private information
- Information that needs to be found later, design documentation

Where to put an issue

- lix-project/lix, if it is contained within Lix (but is not more appropriate to put in the installer e.g.)
  - If it is an upstream bug, tag its equivalent `lix-import` on https://git.lix.systems/nixos/nix, and get someone with the bot token to run the issue import script in `maintainers/issue_import.py`. (FIXME: someone ought to put that on a cron job)
  - Please never file issues on our Nix mirror.
- lix-project/installer, if it is the installer
- lix-project/web-services, if it is infrastructure related
- lix-project/meta, if it does not fit anywhere obvious and you just need it to put it on a board
- lix-project/nixos-module, if it is a packaging bug in that specifically

Gerrit (https://gerrit.lix.systems)

Gerrit is good for:

- Reviewing code
- Maintaining a record of code reviews

Gerrit is not good for:

- Persisting information in a discoverable way to anyone in the future
- Documentation

Why Lix?

(page under construction. editor's note: parts of https://pad.lix.systems/lix-manifesto (PRIVATE) are ported, parts need review before posting here)

(editor's note (ii): this page wants to be a contributor facing page, as opposed to the website page that maybe will have more general info?)

We should introduce ourselves! We are the Lix team, and we are working on a fork of CppNix focused on stability and user experience over features.

Core team members

- puck (@puckipedia), she/her
- hexchen (@hexchen), she/her
  hexchen is working primarily on mantaining and extending the Lix project infrastructure.
- Qyriad (@Qyriad)
  Build system experts who delve way, way too deep into tooling
- eldritch horrors (FIXME(horrors): github if desired?), they/them
- wiggles (@9999years), she/her
- Irenes (@IreneKnapp), they/them
- jade (@lf-), they/them
  jade is working on packaging, testing, infrastructure, tooling, review, stability, and a large amount of the writing in Lix. They are currently studying Computer Engineering at UBC in Canada.
- raito (@RaitoBezarius). he/they
  Raito is working on nixpkgs packaging, infrastructure, review in Lix.
  They are a Tvix developer focusing on the store and the evaluator.
- Kate Temkin (@ktemkin)
  A performance art piece written live by a collective of hardware hackers & low-level engineers. Kate works on Lix as part of a commitment to helping you do cool things, and is seriously considering rewriting every bit of documentation ever to cross paths with Nix.
- Lunaphied (@lunaphied), she/her (singular), they/them (plural)
  Lunaphied spend a disproportionate amount of their time considering how to get FPGAs as far from Earth as possible. When they’re not working on Space Stuff, they consider doing the same for Nix regressions.

FAQ

What is Lix anyway?

Lix is a fork of CppNix 2.18, focused on stability and the user experience of both users and contributors. We want to create a safe platform to move Nix technology forward, as a piece of critical infrastructure.

To this end, we have instituted a freeze on the core, where we apply high standards to changes to the core of the system and pursue testing and stability as our first priority on the core. Our long term vision is to shrink and decouple parts of the core, and move features like Flakes to the periphery of the system.

To achieve our goals in user experience, we are allowing significantly more contributions, still with tests, to the user facing surface of the system where there are fewer stability guarantees, and explicitly define what is expected to be stable and what can change.

Part of our work on the interface of Lix is in Qyriad's project Xil, which is an experiment in an alternate CLI for Nix implementations, which will potentially slowly merge with the Lix CLI.

Technical differences from CppNix

- Lix is built with Meson, so language servers will just work on it
- Lix does not include lazy trees, and does not intend to use the upstream implementation of lazy trees; something like lazy trees is planned (FIXME: publish the planning document for that).
- Lix does not use libgit2 and does not intend to use it
- Lix is entirely self-hosted in terms of infrastructure and uses Gerrit/Forgejo instead of GitHub
- `nix repl` can `:doc` library functions
- `nix repl` can accept overlays as config files; see `repl-overlays` release note in the sources
- Performance improvements (8-20% faster than 2.18)

Views on flakes

The Lix project acknowledges that flakes are the way that the majority of people use Nix today, and does not intend to remove support for them. However, as part of our overall focus on stability and dependability, some features of Flakes will be changed to be stricter.

Flakes are not the only way to write Nix language code in Lix, and we intend to provide a good experience to flakes users, while also improving the experience for those not using flakes, by evolving a compatible but more flexible flake-like abstraction in the periphery of the Lix system.

Why is Lix different from tvix?

tvix is a Nix implementation from the ground up in Rust, aiming to be compatible with CppNix, by building a system from the ground up. It is developed by some of the same people. tvix also aims to improve the stability of Nix technology, but with the approach of starting from the beginning.

Lix is intended to evolve CppNix into a stable foundation for future evolution, without breaking clients along the way. Its goals are to aggressively pursue technical debt and remove the skeletons from the closet, while remaining deliberate about behavioural changes through testing. Lix will contain Rust components in the near future.

The two projects have similar goals but different approaches, and there will likely be cross-pollination between them; though cross-pollination of code is difficult due to licensing.

Style Guide

Not just about code, a style guide is a list of decisions we've made, that we want to be consistent about going forwards. It does not need to be comprehensive of all possible issues, nor does it need to confine itself to trivial topics such as formatting. It's a tool for ourselves so we don't forget where we've been, and can avoid solving the same problems again.

Don't be shy about adding to it. Things written here do impose some burden, but the hope is that they lessen other burdens in the long run. Use your judgement about what's worth it.

Please fix style issues in existing code as you encounter them. Style is aspirational, a journey not a destination. :)

Language and terminology

Language

Most existing Lix documentation is written in British English. We intend to continue with that.

Terminology

(FIXME: unsure if this should be in the style guide but ... it kinda should be -jade)

- **Nix language** - Use this to refer to the language which haunts us all.
- **Nix** - Nix refers to the *technology*. Used when referring to the Nix store, for example. Or, to a Nix derivation. Lix is a *Nix* implementation.
- **CppNix** - This is the preferred term for when it is necessary to refer to Nix, the software that Lix is forked from, rather than the technology.
- **Lix** - Use *Lix* when referring to the implementation. For example, "Install Lix".
- `nix`, `nix-build`, etc - Use lowercase `nix` when referring to the `nix` command, which is still supported by Lix.

Code

Code changes

Tests

If at all practicable, all new code should be tested to some extent. If writing a test is hard, we need to prioritize making it easier, and potentially block features if that is the case.

Documentation

Reference documentation should be added, in addition to release notes (`doc/manual/rl-next-dev`), for user visible changes.

For notable dev facing changes, consider adding release notes in `doc/manual/rl-next-dev`. This is not critical for all changes; in some cases it may make more sense to write it up in dev documentation instead, and indeed it may be ok to defer writing that dev documentation (it's helpful to create an issue to not forget).

Benchmarking

Changes that touch the core of the evaluator or other performance critical code in Lix should be benchmarked.

See bench/README.md for instructions.

Changelist size

If a CL is too long to review, it should be split up into smaller pieces with tests. The exact length varies but passing the 1000 line mark should give significant thought to splitting.

- When a CL is split, each commit should still be a valid state (tests passing, etc). If you must, you can gate in-progress changes with a flag or similar until the final commit. (Qyriad)

Commit messages

Include at least a sentence or two as to why you are making a commit. For example, it can be nice to have the reproduction of a bug in the commit message. The commit message *is* the message for your review.

There's no particular format or specific style for commit messages; just make sure they're descriptive and informative.

C++

While we hope to migrate the lix interpreter from C++ to Rust eventually, C++ is a language that is likely to exist for a long time, and we may end up having to use it in other contexts.

Lix is a C++20 codebase. Features of C++20 that compile on all supported platforms can be used.

NULL vs nullptr

`nullptr` where at all possible.

Static vs anonymous namespace

Prefer anonymous namespace, both currently exist in the codebase (jade: any other opinions?).

Type Aliases with `typedef` vs `using`

Prefer `using` declarations, as they can be used in more places, can be templatized, and have clearer syntax. Both currently exist in the codebase. (Qyriad)

TODO/FIXME/XXX Comments

jade: this is not consistent with the conventions I use, needs further discussion imo (TODO: block in pre-commit hook, used in local tree but should never pass code review, FIXME(name||feature): its busted, someone should go fix it later, XXX: this is bad, we are writing down that it is ugly but leaving it as-is as we didn't figure out a better way)

Something along the lines of:

- **TODO**: acknowledgement that something is acceptably-for-now incomplete, especially if the scope of fixing it is high or unknown
- **FIXME**: this should be fixed before the feature or major change that it's a part of is considered "ready"
- **XXX**: this should not pass code review and should be considered a left-in mistake

Header files

Filenames

Headers should end with `.hh`. This reduces the likelihood anyone will try to include them from C files, which would require following the rules of both languages and is easy to get wrong.

The implementation of the functions declared in a `.hh` file should be in a `.cc` file of the same name, absent reasons to do otherwise.

Order-independence

Headers should not care what order they're loaded in.

The exception, for now, is `config.h` in the `lix` repo. This must always come before all other headers. This observation should not be taken to imply it must always be that way, but at the moment it's helpful to be aware of.

Idempotence

Use `#pragma once`, it helps. You can see this in most existing header files.

`///@file` and header documentation

`///@file` should be at the top of all nix headers - Doxygen and other tools use it to decide whether a header should have documentation generated for definitions in it. See the relevant Doxygen documentation for more details.

Strongly consider adding a description of the purpose of a header file at the top of it in with `@brief A sentence saying what it is for`.

Examples:

```
/**
 * @file
 * @brief This header is for meow meow cat noises.
 */
```

```
/// @file
/// @brief meow meow meow
```

Source files

Filenames

Source files should end with `.cc`.

Nix language

Unsurprisingly Nix contains Nix code. Some amount is tests and a lot is packaging.

We use the nixfmt formatter on files outside the test suite. It's run through `treefmt` with `pre-commit` hooks. Nix code outside the test suite is expected to be formatted.

Test suite files need not be formatted with the formatter at this time, but please consider doing so with new tests that don't rely on formatting.

with

Prefer not to use `with` to bring things into scope as it obscures the source of variables and degrades language server diagnostics.

Use `let inherit (attrset) attrs` instead.

Meson

Generally based on the style in Meson's docs made consistent and with a couple tweaks; notably multiline function calls are done in "block style" (think like rustfmt does it), rather than aligned, e.g.:

```
executable('sdlprog', 'sdlprog.c',
  win_subsystem : 'windows',
  dependencies : sdl2_dep,
)
```

rather than:

```
executable('sdlprog', 'sdlprog.c',
        win_subsystem : 'windows',
        dependencies : sdl2_dep)
```

Meson's docs go back and forth on this, but we also put a space before and after the colon for keyword arguments (so `win_subsystem : 'windows'`, rather than `win_subsystem: 'windows'`).

Operations

Operational Conventions

Code Review

Self Stamping and Merging

On our Gerrit, core members have permissions to +2 any arbitrary CL, because sometimes we should be able to get something in quickly, and talk about it afterwards. In almost every case, the author of a change should not +2 their own CL, however Lix members may use their best judgement so long as they talk about it with the team when they can. Some cases where skipping synchronous review is a good idea:

- Reverting commits that accidentally broke main
- Fixing typos in other peoples' CLs that you would have +2'd, then +2'ing the edited CL
- Maybe typo fixes in `main`, though those can probably wait to be reviewed

Just make sure to talk about what you do :)

Working in the Lix codebase

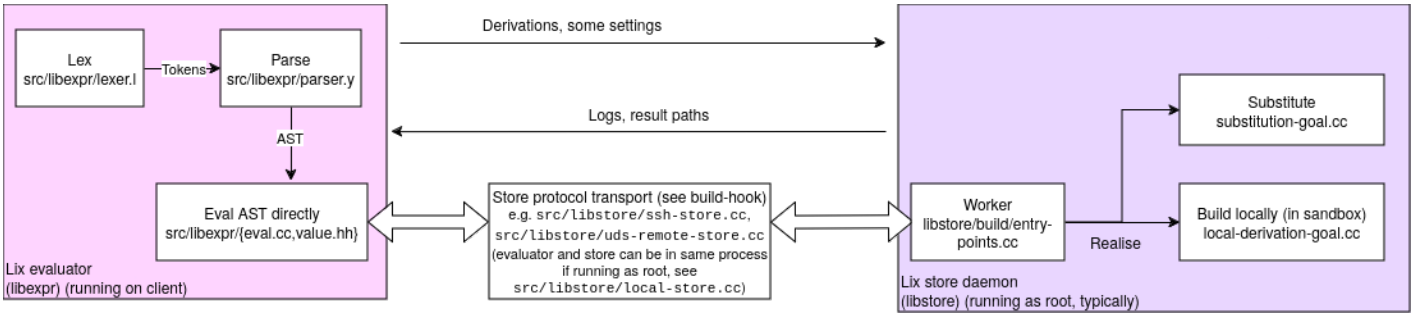
See also: See also: https://git.lix.systems/lix-project/lix/src/branch/main/doc/manual/src/contributing/hacking.md

Codebase overview

The Lix system is constituted of two broad parts, the evaluator and the store daemon. The two pieces may run on the same machine or on different machines.

For example, in a remote build setup like https://hydra.nixos.org, one node is running several evaluators in parallel, and builds are dispatched to several builder nodes.

(fyi to anyone editing this, double click the image in the preview to edit it)



Evaluator

The evaluator is an AST tree-walking evaluator with lazy semantics.

Notable files:

- libexpr/value.hh, which defines the interface for evaluated values' interactions.
- libexpr/eval.cc, where most of the evaluator is.
- libexpr/nixexpr.cc, where the most of the `nix::Expr` class hierarchy is implemented, which are the AST types for the evaluator.
- libexpr/primops.cc, defining builtins.
- libexpr/parser.y, the (current) yacc generated parser.
- libexpr/lexer.l, the bison-generated lexer.

Known design flaws

- GC issues (FIXME add details)
  - General tendencies to leak memory. Hydra restarts the evaluator every so often if it runs out of memory.
- AST based evaluator design limits perf
- Stack tracing has issues that make the traces confusing (FIXME add details)

- Funniness with attr ordering and equality that nixpkgs depends on, which is fragile
  - Currently no real tools to diagnose this and stop nixpkgs from depending on it.

    https://github.com/NixOS/nix/pull/8711 exists but regresses perf a lot and is not mergeable.
- Evaluation-time build dependencies (often called IFD) block the evaluator rather than allowing other evaluation to proceed
  - This has significant downstream effects such as typical derivations building hand-written large pieces rather than generated smaller pieces with IFD, since IFD is bad.
- The eval cache both has false hits and false misses, and needs redesign.

Lix team plans

- Rewrite parser (done, being ported for 2.91 by horrors)
- Rewrite evaluator to be amenable to moving to bytecode (horrors) (long term)
- Do something about GC (long term)

Store protocol

The store protocol is a hand rolled binary protocol with monotonically increasing versioning. It runs over a few different transports such as ssh (`src/libstore/ssh-store.cc`) or Unix sockets (`src/libstore/uds-remote-store.cc`).

Known design flaws

- We cannot extend the store protocol (not that it is Good) because of the monotonic version numbers: we must always be stuck at *some* released CppNix version. This significantly moves up the need to replace it.
  - The code is significantly tangled with the current protocol design.

Lix team plans

- Replace protocol with capnproto, transport with websockets?
  - Would likely be in addition to existing protocol; existing protocol likely would be run through a translator.

Store daemon

The store daemon takes derivations (≈ execve args and dependencies) and realises (builds or substitutes) them. It also implements store path garbage collection.

Lix's local store implementation currently uses a SQLite database as the source of truth for mapping derivation outputs to derivations as well as maintaining derivation metadata.

Notable files:

- libstore/build/local-derivation-goal.cc, which implements the local machine's builder including the sandbox
- libstore/build/entry-points.cc, the server side entry points of the store protocol
- libstore/daemon.cc
- libstore/uds-remote-store.cc, the client implementation of Unix socket stores

Known design flaws

- Sandbox is of dubious security especially on Linux (where it is actually expected to be somewhat secure)
  - Overall tangled code around the sandbox, particularly in platform specific parts
- Poor self-awareness: daemon doesn't know what it is building
  - Due to this plus the protocol being frozen, it would be very hard to implement e.g. dropping into a shell on failed builds
- Substitutions are inherently a kind of build so they can't happen out of dependency order or with better parallelism
- SQLite database has a habit of getting corrupted (probably due to Lix-side misuse)

Lix team plans

- Replace sandbox with other software, perhaps bwrap
- Fix daemon self-awareness, add protocol level features to make this better
- Rearchitect substitution to enqueue weakly ordered jobs that happen in parallel and can resume downloads
- Switch to xattrs as the source of truth of store path metadata such that the SQLite DB can be completely rebuilt

Freezes and recommended contributions

Suggested contributions

Consider taking an issue marked E-help wanted: assign it to yourself and have a go. Feel free to ask for help in the development channel. The Lix team wants these issues fixed, but they are not high on our agenda to fix ourselves.

When in doubt, please ask the Lix team before beginning work, to make sure it is in line with our current priorities.

Freezes

This document describes the state of freeze that Lix is in.

We *do* expect to have `main` always be in a state to run on machines you care about, unconditionally. Nightly builds should not be a problem to run in production in any freeze state.

The purpose of this policy is to set expectations of what we are looking for in contributions, rather than to set hard rules.

> Lix is currently in status "*milkshake*".

ice cube

No major features or code changes are accepted that touch the core (e.g. the hot paths of the evaluator, the store), absent a good justification, good test coverage and a strong belief that they will not cause regressions. In this state, we don't recommend external contributors do substantive work outside the roadmap without speaking with the Lix team first. However, this is a guideline, and such work can be done if discussed and planned carefully beforehand.

Simple surface features with low impact are likely to be accepted with tests, assuming that they do not impact reliability.

New tests are gleefully accepted.

Bug fixes (with tests) are gleefully accepted.

For example, the following would require discussion with the Lix team before work begins, as it is likely to not fit our goals:

- Adding new features to ca-derivations
- Doing substantial not-obviously-correct refactoring to the evaluator or daemon

For example, the following would likely be accepted assuming it has tests, without needing prior discussion:

- Backports of CLI UX features from CppNix
- New UX features in the REPL or in output of other commands considered to not have stable output
- Backtrace improvements that don't touch hot paths
- Bug fixes (to non load bearing bugs; be careful around evaluator semantics!)
- Improvements to development UX

hard ice cream

Changes that improve maintainability of the core are accepted, with careful review depending on their significance. Changes that add more complexity to the core need to pass scrutiny.

Features at the edge are accepted if they have low impact, assuming that they have tests.

soft serve

FIXME

milkshake

All kinds of changes are acceptable, but we still strive to keep `main` always as stable as possible and a safe decision to daily-drive for all your nixy needs. Please don't jam the ice cream machine!

Bug tracker organisation

We have a repo of directly imported nix bugs at https://git.lix.systems/nixos/nix. Please don't file bugs in there, we want the IDs to match. When we import a bug, we might put notes on there as we triage it, and potentially close it.

Bug labels on NixOS/nix

- lix-import - Should be imported, we think it is still a bug
- lix-ignore - We don't care about this bug, it probably doesn't affect us
- lix-stability - Fixing this would improve the stability and reliability of Lix.

Dispositions:

- lix-norepro - Tried repro on upstream 2.18.1 and did not repro
- lix-retest-after-backports - Request that this be tested again once backports are done
- lix-reproduces-2.18 - Confirmed to repro in 2.18.
- lix-unclear-repro - Unsure how to repro but believe it affects lix
- lix-closed-libgit2 - Caused by libgit2
- lix-closed-lazy-trees - Caused by lazy trees

Closed, marginal

- post-build-hook doesn't print a warning if not trusted-user
  https://git.lix.systems/NixOS/nix/issues/9790#issuecomment-273
- complaints about builtins.derivation https://git.lix.systems/NixOS/nix/issues/9774
- rejecting flake config still asks for confirm again
  https://git.lix.systems/NixOS/nix/issues/9788
- complaints of "substituter disabled", but is their bin cache just broken?
  https://git.lix.systems/NixOS/nix/issues/9749
- warn on eol https://git.lix.systems/NixOS/nix/issues/9556

Gerrit

What is Gerrit and why do people like it?

Gerrit is a code review system from Google in a similar style to Google's internal Critique tool, but based on Git, and publicly available as open source. It hosts a Git repo with the ability to submit changes for review and offers mirroring to other repos (like https://git.lix.systems/lix-project/lix). It has an entirely different review model to GitHub (and Forgejo, GitLab, etc that copy GitHub's review model), where, instead of pull requests, you have changelists (CLs): reviews on individual commits, with each revision of a commit being a different "patchset", rather than reviewing an entire branch at a time. CLs may be merged one by one or in a batch.

Although this has some learning curve, we expect that you will find it pleasant to work with after figuring it out. It has some rough edges and strong opinions that take some getting used to, but it has served us well and saved us an inordinate amount of time both as reviewers and change authors. The rest of this document gives some pointers on the workflows we use with Gerrit.

People like Gerrit because it makes the following things trivial or easy, all of which are somewhere between annoying and impossible on GitHub modeled systems:

Gerrit produces better code:

- Gerrit enforces good commit messages, since there is no second "pr message" so peoples' commit messages get actually looked at with some care
- Gerrit enforces good commit *hygiene*, since adding another commit is really just splitting a commit with `git revise -c` or other tools; since there are no PR dependencies or branches to worry about, splitting commits is no longer a big ask.
  - Relatedly, this directly makes reviews smaller since the overhead of doing another change is low.

Gerrit makes reviewers' lives easier and reduces review round trips:

- As a reviewer, you can look at what changed since you last reviewed, even in the presence of rebases, by looking at the patchset history of a CL. This avoids pointless rereview; you can actually diff versions of changes properly.
- The *change author* generally merges the change after approval, without them needing commit access. This means that they can do a final once-over of the change and make sure that they are ok with its state before merging it. This reduces miscommunication causing merging of unfinished code.
- As a reviewer, you can edit someone's change and/or commit message to fix a typo (*in the web interface*) and then stamp it, while giving them the final say on merging the

edited change.
- You can give feedback like the following: "I would merge this as-is but you can consider this feedback if you would like" and then let the change author decide to merge it.
  - Since the permission-requiring step in Gerrit is *approving* the change, not merging it, every change author can have final say in when the change gets merged.
- Review suggestions get applied as a batch without cluttering commit history in a confusing manner.
- You can download someone's change to look at it locally in one command that you can copy paste from the Gerrit interface.

Gerrit makes your life easier as a contributor:

- Submitting a new change is just a matter of committing it and pushing it. You don't need to think about branches or the web interface or extra commands. Want to do more changes building on it? Just commit them and push them.
- Branches are not required and you can easily build off of other peoples' changes by fetching them and rebasing against them; change dependencies are simply commit parents. They can then be merged in whichever manner they will be merged.
- If you are doing a larger change, it is natural to merge it piece by piece, adding little improvements as you go, and putting the highest risk parts of it at the tip, making the obviously good parts of the change land and keeping your diffs and rebases against `main` smaller.
- Gerrit makes it clear which comments still need action in a clean way, compared to GitHub where resolved comments get regularly broken or disappear altogether.
- Gerrit guesses (with reasonable accuracy) who a change is blocked on and shows it on the dashboard with a little arrow next to their name, allowing you to see at a glance which changes are your responsibility at a given time.
- There is a rebase button that just works. Trivial non-conflicting rebases do not require a rereview.

That being said, there are some downsides:

- Gerrit is very mean to you if you don't have your commit history in a clean presentable state, which takes some getting used to and Git does not make editing history easy, so it does involve a little more fighting of Git. However, this also means that the reviews can be of cleaner and smaller pieces of code with fewer unrelated changes.
  - This makes pushing work in progress code with questionable commit history harder; see below for solutions to this.
- Gerrit requires a little bit of local setup in the form of adding your SSH key or setting up the HTTP password. It also requires a Git commit-msg hook, but `nix develop` automatically does that for you.

Learning materials

- https://gerrit-review.googlesource.com/Documentation/intro-user.html

- https://docs.google.com/presentation/d/1C73UgQdzZDw0gzpaEqIC6SPujZJhqamyqO1XOHjH-uk/view

Our installation

Gerrit is at https://gerrit.lix.systems

The Gerrit SSH server is running on port 2022. The repo URLs are:

- `ssh://{username}@gerrit.lix.systems:2022/lix`
- `https://gerrit.lix.systems/lix` if using HTTP auth; see Gerrit settings for setting an HTTP password if desired

Hit the `d` key on any change to download it, which will give you the right URLs.

SSH config

You might like to add the following configuration to your `~/.ssh/config`:

```
Host gerrit.lix.systems
  User YOUR_GERRIT_USERNAME
  Port 2022
  # Keep sessions open for a bit in the background to make connections faster:
  ControlMaster auto
  ControlPath /tmp/ssh-%r@%h:%p
  ControlPersist 120
```

Basic workflow for a change

The unit of code review is a "change", which yields a single commit when "submitted" (merged). The commit message is taken from the change description in Gerrit; in our experience this tends to lead to more comprehensive commit messages.

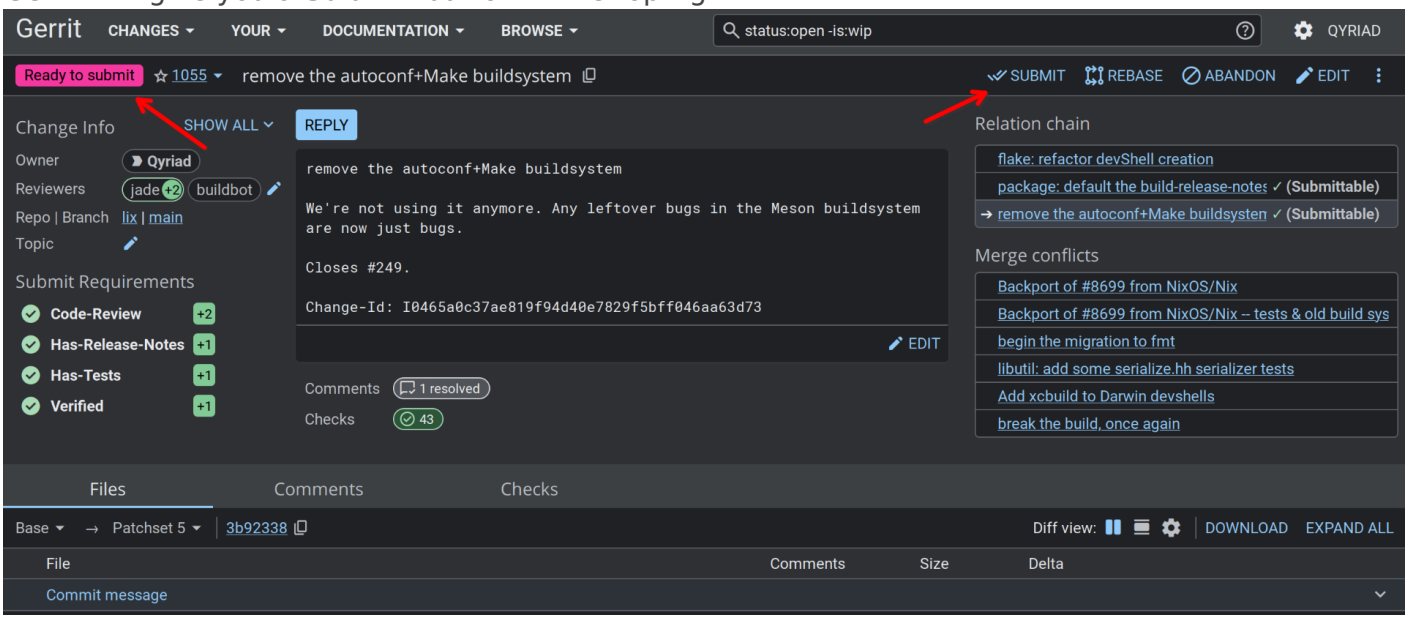
For a change to be merged, it must have the following four "votes", in Gerrit's terminology:

- Set by reviewers:
  - +2 Code-Review: the committer that reviewed this thinks it can be submitted as-is (all users can vote +1/-1, expressing a weaker view on code acceptability)
  - +1 Has-Release-Notes: means the reviewer thinks your commit added relevant release notes for that commit, or that it does not need any. This serves primarily as a reminder.
  - +1 Has-Tests: means the reviewer thinks your commit added all the tests that commit needs, or that it does not need additional tests. Like Has-Release-Notes, this

serves primarily as a reminder.
- Set automatically by CI:
  - +1 Verified: means CI successfully built for all our platforms and passed all tests

If you're newly part of the core team you will need to add yourself to the Gerrit `lix` group, otherwise you can't set the `Has-Release-Notes` or `Has-Tests` labels. If you're not, this doesn't affect you.

When all of those labels are set, a change becomes **Ready to submit**, in Gerrit's termology, and Gerrit will give you a **Submit** button in the top right:



By convention, ***the change author*** *has the final say on clicking the Submit button* (note: this is the opposite of the Github convention), and there is no special permission to merge a change once it has been fully reviewed (the permissions are in the reviewer +2'ing it). This gives you a last chance to have a look at your change before merging it.

Workflow tips

Local branches and commits

Gerrit is very mean to you if you don't have your local commit history in a linear presentable state, which takes getting used to but it is very low overhead once you get used to it. In short, amended commits become "patchsets", new commits become changes, and multiple commits help link your changes together as a "relation chain".

Note: if you're coming from Chromium, this is different to how they use Gerrit, where multiple commits become patchsets, and only the first commit on a local branch creates a new change.

Gerrit's commit-msg hook generates a new Change-Id for each commit you make, which in turn creates a new change that gets reviewed separately. To update an existing change after review feedback, amend or squash your changes into your old commit, keeping its Change-Id unchanged,

then push.

Consider not pushing for review before it is clean, or split commits up with `git-revise` (good) or `jj` (better) after the fact, amending as you work. If you want a backup of your changes, you can fork it on Forgejo and push to that fork.

Basic Pushing

If you cloned the repo from Forgejo, be sure to change your remote URL to point to Gerrit before continuing. Assuming your remote is called `origin` (which is the default):

`git remote set-url origin ssh://{username}@gerrit.lix.systems:2022/lix`

Then you can push to Gerrit with:

`git push origin HEAD:refs/for/main`

If you get tired of doing this every time, you can automate it by setting the `.git/config` as follows:

`git config remote.origin.push HEAD:refs/for/main`

You will have to do that in each fresh check-out. Once it's done, `git push` will work without additional options.

If you get a “remote unpack failed” error while pushing, run `git fetch` then try again.

If you wish to push a change and immediately mark it as WIP, you can push with `-o wip`, or make that the default behavior by checking `Set new changes to "work in progress" by default` in Gerrit's user settings, under "Preferences".

Topics & Push Arguments

A Gerrit topic may be set on push with:

```
git push origin HEAD:refs/for/main%topic=foo
```

Which will create all pushed changes with the topic "foo". Topics are helpful for grouping long series of related changes.

A change may also be marked as "work in progress" on push:

```
git push origin HEAD:refs/for/main%wip
```

Gerrit has documentation on other push arguments you can use here, but it also takes a `help` argument whose output is more canonical and might be easier to understand, which you can view with:

```
git push origin HEAD:refs/for/main%help
```

At the time of this writing (2024/06/26), that output looks like this:

```
$ git push origin @:refs/for/main%help
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Processing changes: refs: 1, done
remote:
remote: Help for refs/for/branch:
remote:
remote:  --base BASE                      : merge base of changes
remote:  --cc CC                          : add CC to changes
remote:  --create-cod-token                 : create a token for consistency-on-dema
remote:                                   nd (default: false)
remote:  --deadline NAME                    : deadline after which the push should
remote:                                   be aborted
remote:  --edit (-e)                       : upload as change edit (default: false)
remote:  --hashtag (-t) HASHTAG               : add hashtag to changes
remote:  --help (-h)                       : display this help text (default: true)
remote:  --ignore-automatic-attention-set-rules : do not change the attention set on
remote:  (-ias, -ignore-attention-set)          this push (default: false)
remote:  --label (-l) LABEL+VALUE             : label(s) to assign (defaults to +1 if
remote:                                   no value provided)
remote:  --merged                          : create single change for a merged
remote:                                   commit (default: false)
remote:  --message (-m) MESSAGE               : Comment message to apply to the review
remote:  --no-publish-comments (--np)         : do not publish draft comments
remote:                                   (default: false)
remote:  --notify [NONE | OWNER |             : Notify handling that defines to whom
remote:  OWNER_REVIEWERS | ALL]                 email notifications should be sent.
remote:                                   Allowed values are NONE, OWNER,
remote:                                   OWNER_REVIEWERS, ALL. If not set, the
remote:                                   default is ALL.
remote:  --notify-bcc USER                  : user that should be BCC'd one time by
remote:                                   email
remote:  --notify-cc USER                   : user that should be CC'd one time by
remote:                                   email
remote:  --notify-to USER                   : user that should be notified one time
remote:                                   by email
remote:  --private                         : mark new/updated change as private
```

```
remote:                                  (default: false)
remote:  --publish-comments                    : publish all draft comments on updated
remote:                                  changes (default: false)
remote:  --ready                         : mark change as ready (default: false)
remote:  --remove-private                      : remove privacy flag from updated
remote:                                  change (default: false)
remote:  --reviewer (-r) REVIEWER              : add reviewer to changes
remote:  --skip-validation               : skips commit validation (default:
remote:                                  false)
remote:  --submit                        : immediately submit the change
remote:                                  (default: false)
remote:  --topic NAME                    : attach topic to changes
remote:  --trace NAME                    : enable tracing
remote:  --wip (-work-in-progress)             : mark change as work in progress
remote:                                  (default: false)
remote:
To ssh://gerrit.lix.systems:2022/lix
 ! [remote rejected]     HEAD -> refs/for/main%help (see help)
error: failed to push some refs to 'ssh://gerrit.lix.systems:2022/lix'
```

Pulling

Pulling from Gerrit will work normally. It's worth keeping in mind that sometimes a CL you're working on has been edited in the web UI or by another contributor, so the commit in your repo isn't the latest. Rebasing will usually make the duplicate go away; this is part of the normal rebase semantics, not Gerrit magic. You might consider making rebase-on-pull your default.

Sandbox branches

> This feature has some notable ways to shoot yourself in the foot. We still support it, since it allows for running CI builds on things before they become proper CLs. If you don't need that and don't want to worry about the footguns, consider using a branch on a Forgejo fork for sharing WIP code.
>
> In particular, if a commit is in *any* branch already including a `sb/` branch, it will be rejected with the error "no new changes" if it is later pushed to `refs/for/main`. This can be worked around by amending all the commits so they are distinct, or by `git push origin HEAD:refs/for/main%base=$(git rev-parse origin/main)`, which forces the merge-base

Use `refs/heads/sb/USERNAME/*`.

CI rerun

Push the CL again with a no-changes commit amendment if you want to force CI to rerun.

Finding CLs to review

Consider bookmarking: https://gerrit.lix.systems/q/status:open+-is:wip+-author:me+label:Code-Review%3C2

Improving build times

Setup

Use a clang stdenv:

`nix develop .#native-clangStdenvPackages`

Then delete `build/` if you were using gcc before. Enable build-time profiling with:

`just setup; meson configure build -Dprofile-build=enabled`

Then run the build: `just compile`.

Enabling build-time profiling itself costs about 10% of compile time but has no other disadvantage.

Build time reports

Use `maintainers/buildtime_report.sh build/` to generate a build time report. This will tell you where all our build time went by looking at the trace files and producing a badness summary.

Sample report

Build-time report sample

```
lix/lix2 » ClangBuildAnalyzer --analyze buildtimeold.bin
Analyzing build trace from 'buildtimeold.bin'...
**** Time summary:
Compilation (551 times):
  Parsing (frontend):         1465.3 s
  Codegen & opts (backend):   1110.9 s

**** Files that took longest to parse (compiler frontend):
 10478 ms: build/src/libstore/liblixstore.so.p/build_local-derivation-goal.cc.o
 10319 ms: build/src/libexpr/liblixexpr.so.p/primops.cc.o
  9947 ms: build/src/nix/nix.p/flake.cc.o
```

```
  9850 ms: build/src/libexpr/liblixexpr.so.p/eval.cc.o
  9751 ms: build/src/nix/nix.p/profile.cc.o
  9643 ms: build/src/nix/nix.p/develop.cc.o
  9296 ms: build/src/libcmd/liblixcmd.so.p/installable-attr-path.cc.o
  9286 ms: build/src/libstore/liblixstore.so.p/build_derivation-goal.cc.o
  9208 ms: build/src/libcmd/liblixcmd.so.p/installables.cc.o
  9007 ms: build/src/nix/nix.p/..._nix-env_nix-env.cc.o

**** Files that took longest to codegen (compiler backend):
 24226 ms: build/src/libexpr/liblixexpr.so.p/primops_fromTOML.cc.o
 24019 ms: build/src/libexpr/liblixexpr.so.p/primops.cc.o
 21102 ms: build/src/libstore/liblixstore.so.p/build_local-derivation-goal.cc.o
 16246 ms: build/src/libstore/liblixstore.so.p/store-api.cc.o
 14586 ms: build/src/nix/nix.p/..._nix-build_nix-build.cc.o
 13746 ms: build/src/libexpr/liblixexpr.so.p/eval.cc.o
 13287 ms: build/src/libstore/liblixstore.so.p/binary-cache-store.cc.o
 13263 ms: build/src/nix/nix.p/profile.cc.o
 12970 ms: build/src/nix/nix.p/develop.cc.o
 12621 ms: build/src/libfetchers/liblixfetchers.so.p/github.cc.o

**** Templates that took longest to instantiate:
 42922 ms: nlohmann::basic_json<>::parse<const char *> (69 times, avg 622 ms)
 32180 ms: nlohmann::detail::parser<nlohmann::basic_json<>, nlohmann::detail::i... (69 times, avg 466 ms)
 27337 ms: nix::HintFmt::HintFmt<nix::Uncolored<std::basic_string<char>>> (246 times, avg 111 ms)
 25338 ms: nlohmann::basic_json<>::basic_json (293 times, avg 86 ms)
 23641 ms: nlohmann::detail::parser<nlohmann::basic_json<>, nlohmann::detail::i... (69 times, avg 342 ms)
 20203 ms: boost::basic_format<char>::basic_format (492 times, avg 41 ms)
 17174 ms: nlohmann::basic_json<>::json_value::json_value (368 times, avg 46 ms)
 15603 ms: boost::basic_format<char>::parse (246 times, avg 63 ms)
 13268 ms: std::basic_regex<char>::_M_compile (28 times, avg 473 ms)
 12757 ms: std::__detail::_Compiler<std::regex_traits<char>>::_Compiler (28 times, avg 455 ms)
 10813 ms: std::__detail::_Compiler<std::regex_traits<char>>::_M_disjunction (28 times, avg 386 ms)
 10719 ms: std::__detail::_Compiler<std::regex_traits<char>>::_M_alternative (28 times, avg 382 ms)
 10508 ms: std::__detail::_Compiler<std::regex_traits<char>>::_M_term (28 times, avg 375 ms)
  9516 ms: nlohmann::detail::json_sax_dom_callback_parser<nlohmann::basic_json<... (69 times, avg 137
ms)
  9112 ms: std::__detail::_Compiler<std::regex_traits<char>>::_M_atom (28 times, avg 325 ms)
  8683 ms: std::basic_regex<char>::basic_regex (18 times, avg 482 ms)
  8241 ms: std::operator<=> (438 times, avg 18 ms)
```

7561 ms: std::vector<boost::io::detail::format_item<char, std::char_traits<ch... (246 times, avg 30 ms)
7475 ms: std::vector<boost::io::detail::format_item<char, std::char_traits<ch... (246 times, avg 30 ms)
7309 ms: std::reverse_iterator<std::_Bit_iterator> (268 times, avg 27 ms)
7131 ms: boost::stacktrace::basic_stacktrace<>::basic_stacktrace (246 times, avg 28 ms)
6868 ms: boost::stacktrace::basic_stacktrace<>::init (246 times, avg 27 ms)
6518 ms: std::reverse_iterator<std::_Bit_const_iterator> (268 times, avg 24 ms)
5716 ms: std::__detail::_Synth3way::operator()<std::variant<nix::OutputsSpec:... (182 times, avg 31 ms)
5303 ms: nix::make_ref<nix::SingleDerivedPath, nix::DerivedPathOpaque> (178 times, avg 29 ms)
5244 ms: std::__uninitialized_move_a<boost::io::detail::format_item<char, std... (246 times, avg 21 ms)
4857 ms: std::make_shared<nix::SingleDerivedPath, nix::DerivedPathOpaque> (178 times, avg 27 ms)
4813 ms: std::__detail::_Synth3way::operator()<std::variant<nix::TextIngestio... (158 times, avg 30 ms)
4648 ms: nlohmann::detail::json_sax_dom_callback_parser<nlohmann::basic_json<... (69 times, avg 67 ms)
4597 ms: std::basic_regex<char>::basic_regex<std::char_traits<char>, std::all... (10 times, avg 459 ms)

**** Template sets that took longest to instantiate:
55715 ms: std::__do_visit<$> (3603 times, avg 15 ms)
47739 ms: std::__detail::__variant::__gen_vtable_impl<$>::__visit_invoke (11132 times, avg 4 ms)
43338 ms: nlohmann::basic_json<$>::parse<$> (85 times, avg 509 ms)
43097 ms: std::__detail::__variant::__raw_idx_visit<$> (2435 times, avg 17 ms)
32390 ms: nlohmann::detail::parser<$>::parse (83 times, avg 390 ms)
30986 ms: nix::HintFmt::HintFmt<$> (1261 times, avg 24 ms)
30255 ms: std::__and_<$> (25661 times, avg 1 ms)
29762 ms: std::unique_ptr<$> (2116 times, avg 14 ms)
28609 ms: std::__tuple_compare<$>::__eq (2978 times, avg 9 ms)
27560 ms: nlohmann::detail::parser<$>::sax_parse_internal<$> (167 times, avg 165 ms)
27239 ms: std::variant<$> (1959 times, avg 13 ms)
26837 ms: std::__invoke_result<$> (10782 times, avg 2 ms)
25972 ms: std::tuple<$> (5714 times, avg 4 ms)
24247 ms: std::__uniq_ptr_data<$> (2116 times, avg 11 ms)
24061 ms: std::__result_of_impl<$> (9029 times, avg 2 ms)
23949 ms: std::__uniq_ptr_impl<$> (2116 times, avg 11 ms)
21185 ms: std::optional<$> (2502 times, avg 8 ms)
21044 ms: std::pair<$> (4989 times, avg 4 ms)
20852 ms: std::__or_<$> (24005 times, avg 0 ms)
20203 ms: boost::basic_format<$>::basic_format (492 times, avg 41 ms)
20184 ms: std::tie<$> (2895 times, avg 6 ms)
19938 ms: nlohmann::basic_json<$>::create<$> (668 times, avg 29 ms)
19798 ms: std::allocator_traits<$>::construct<$> (5720 times, avg 3 ms)

```
 19182 ms: std::__detail::__variant::_Variant_base<$> (1959 times, avg 9 ms)
 19151 ms: std::_Rb_tree<$>::_M_erase (2320 times, avg 8 ms)
 19094 ms: std::_Rb_tree<$>::~_Rb_tree (2022 times, avg 9 ms)
 18735 ms: nlohmann::basic_json<$>::basic_json (243 times, avg 77 ms)
 18546 ms: std::__detail::_Synth3way::_S_noexcept<$> (2542 times, avg 7 ms)
 17174 ms: nlohmann::basic_json<$>::json_value::json_value (368 times, avg 46 ms)
 17111 ms: nlohmann::detail::conjunction<$> (907 times, avg 18 ms)

**** Functions that took longest to compile:
  2091 ms: _GLOBAL__sub_I_primops.cc (../src/libexpr/primops.cc)
  1799 ms: nix::fetchers::GitInputScheme::fetch(nix::ref<nix::Store>, nix::fetc... (../src/libfetchers/git.cc)
  1388 ms: nix::Settings::Settings() (../src/libstore/globals.cc)
  1244 ms: main_nix_build(int, char**) (../src/nix-build/nix-build.cc)
  1021 ms: nix::LocalDerivationGoal::startBuilder() (../src/libstore/build/local-derivation-goal.cc)
   918 ms: nix::LocalStore::LocalStore(std::map<std::__cxx11::basic_string<char... (../src/libstore/local-store.cc)
   835 ms: opQuery(Globals&, std::__cxx11::list<std::__cxx11::basic_string<char... (../src/nix-env/nix-env.cc)
   733 ms: nix::daemon::performOp(nix::daemon::TunnelLogger*, nix::ref<nix::Sto... (../src/libstore/daemon.cc)
   589 ms: _GLOBAL__sub_I_tests.cc (../tests/unit/libutil/tests.cc)
   578 ms: main_build_remote(int, char**) (../src/build-remote/build-remote.cc)
   522 ms: nix::fetchers::MercurialInputScheme::fetch(nix::ref<nix::Store>, nix... (../src/libfetchers/mercurial.cc)
   521 ms: nix::LocalDerivationGoal::registerOutputs[abi:cxx11]() (../src/libstore/build/local-derivation-goal.cc)
   461 ms: nix::getNameFromURL_getNameFromURL_Test::TestBody() (../tests/unit/libutil/url-name.cc)
   440 ms: nix::Installable::build2(nix::ref<nix::Store>, nix::ref<nix::Store>,... (../src/libcmd/installables.cc)
   392 ms: nix::prim_fetchClosure(nix::EvalState&, nix::PosIdx, nix::Value**, n... (../src/libexpr/primops/fetchClosure.cc)
   390 ms: nix::NixArgs::NixArgs() (../src/nix/main.cc)
   388 ms: update(std::set<std::__cxx11::basic_string<char, std::char_traits<ch... (../src/nix-channel/nix-channel.cc)
   340 ms: _GLOBAL__sub_I_primops.cc (../tests/unit/libexpr/primops.cc)
   332 ms: nix::flake::lockFlake(nix::EvalState&, nix::FlakeRef const&, nix::fl... (../src/libexpr/flake/flake.cc)
   305 ms: _GLOBAL__sub_I_lockfile.cc (../src/libexpr/flake/lockfile.cc)
   300 ms: nix_store::opQuery(std::__cxx11::list<std::__cxx11::basic_string<cha... (../src/nix-store/nix-store.cc)
   296 ms: nix::parseFlakeRefWithFragment(std::__cxx11::basic_string<char, std:... (../src/libexpr/flake/flakeref.cc)
```

289 ms: _GLOBAL__sub_I_error_traces.cc (../tests/unit/libexpr/error_traces.cc)
278 ms: nix::ErrorTraceTest_genericClosure_Test::TestBody() (../tests/unit/libexpr/error_traces.cc)
274 ms: CmdDevelop::run(nix::ref<nix::Store>, nix::ref<nix::Installable>) (../src/nix/develop.cc)
269 ms: nix::flake::lockFlake(nix::EvalState&, nix::FlakeRef const&, nix::fl... (../src/libexpr/flake/flake.cc)
257 ms: nix::NixRepl::processLine(std::__cxx11::basic_string<char, std::char... (../src/libcmd/repl.cc)
251 ms: nix::derivationStrictInternal(nix::EvalState&, std::__cxx11::basic_s... (../src/libexpr/primops.cc)
249 ms: toml::result<toml::basic_value<toml::discard_comments, std::unordere... (../src/libexpr/primops/fromTOML.cc)
238 ms: nix::LocalDerivationGoal::runChild() (../src/libstore/build/local-derivation-goal.cc)

**** Function sets that took longest to compile / optimize:
10243 ms: std::vector<$>::_M_fill_insert(__gnu_cxx::__normal_iterator<$>, unsi... (190 times, avg 53 ms)
9752 ms: bool boost::io::detail::parse_printf_directive<$>(__gnu_cxx::__norma... (190 times, avg 51 ms)
8377 ms: void boost::io::detail::put<$>(boost::io::detail::put_holder<$> cons... (191 times, avg 43 ms)
5863 ms: boost::basic_format<$>::parse(std::__cxx11::basic_string<$> const&) (190 times, avg 30 ms)
5660 ms: std::vector<$>::_M_fill_insert(std::_Bit_iterator, unsigned long, bo... (190 times, avg 29 ms)
4264 ms: non-virtual thunk to boost::wrapexcept<$>::~wrapexcept() (549 times, avg 7 ms)
4023 ms: std::_Rb_tree<$>::_M_erase(std::_Rb_tree_node<$>*) (1238 times, avg 3 ms)
3715 ms: boost::stacktrace::detail::to_string_impl_base<boost::stacktrace::de... (166 times, avg 22 ms)
3705 ms: std::vector<$>::_M_fill_assign(unsigned long, boost::io::detail::for... (190 times, avg 19 ms)
3326 ms: boost::basic_format<$>::str[abi:cxx11]() const (144 times, avg 23 ms)
3070 ms: void boost::io::detail::mk_str<$>(std::__cxx11::basic_string<$>&, ch... (191 times, avg 16 ms)
2839 ms: boost::basic_format<$>::make_or_reuse_data(unsigned long) (190 times, avg 14 ms)
2321 ms: std::__cxx11::basic_string<$>::_M_replace(unsigned long, unsigned lo... (239 times, avg 9 ms)
2213 ms: std::_Rb_tree<$>::_M_get_insert_hint_unique_pos(std::_Rb_tree_const_... (203 times, avg 10 ms)
2200 ms: boost::wrapexcept<$>::~wrapexcept() (549 times, avg 4 ms)
2093 ms: std::vector<$>::~vector() (574 times, avg 3 ms)
1894 ms: bool std::__detail::_Compiler<$>::_M_expression_term<$>(std::__detai... (112 times, avg 16 ms)
1871 ms: int boost::io::detail::upper_bound_from_fstring<$>(std::__cxx11::bas... (190 times, avg 9 ms)
1867 ms: boost::wrapexcept<$>::clone() const (549 times, avg 3 ms)
1824 ms: std::_Rb_tree_iterator<$> std::_Rb_tree<$>::_M_emplace_hint_unique<$... (244 times, avg 7 ms)
1821 ms: toml::result<$> toml::detail::sequence<$>::invoke<$>(toml::detail::l... (93 times, avg 19 ms)
1814 ms: nlohmann::json_abi_v3_11_2::detail::serializer<$>::dump(nlohmann::js... (39 times, avg 46 ms)
1799 ms: nix::fetchers::GitInputScheme::fetch(nix::ref<$>, nix::fetchers::Inp... (1 times, avg 1799 ms)
1771 ms: boost::io::detail::format_item<char, std::char_traits<char>, std::al... (190 times, avg 9 ms)
1762 ms: std::__detail::_BracketMatcher<$>::_BracketMatcher(std::__detail::_B... (112 times, avg 15 ms)
1760 ms: std::_Function_handler<$>::_M_manager(std::_Any_data&, std::_Any_dat... (981 times, avg 1 ms)
1733 ms: std::__detail::_Compiler<$>::_M_quantifier() (28 times, avg 61 ms)

```
  1694 ms: std::__cxx11::basic_string<$>::_M_mutate(unsigned long, unsigned lon... (251 times, avg 6 ms)
  1650 ms: std::vector<$>::vector(std::vector<$> const&) (210 times, avg 7 ms)
  1650 ms: boost::io::basic_altstringbuf<$>::overflow(int) (190 times, avg 8 ms)

**** Expensive headers:
178153 ms: ../src/libcmd/installable-value.hh (included 52 times, avg 3426 ms), included via:
  40x: command.hh
  5x: command-installable-value.hh
  3x: installable-flake.hh
  2x: <direct include>
  2x: installable-attr-path.hh

176217 ms: ../src/libutil/error.hh (included 246 times, avg 716 ms), included via:
  36x: command.hh installable-value.hh installables.hh derived-path.hh config.hh experimental-features.hh
  12x: globals.hh config.hh experimental-features.hh
  11x: file-system.hh file-descriptor.hh
  6x: serialise.hh strings.hh
  6x: <direct include>
  6x: archive.hh serialise.hh strings.hh
  ...

173243 ms: ../src/libstore/store-api.hh (included 152 times, avg 1139 ms), included via:
  55x: <direct include>
  39x: command.hh installable-value.hh installables.hh
  7x: libexpr.hh
  4x: local-store.hh
  4x: command-installable-value.hh installable-value.hh installables.hh
  3x: binary-cache-store.hh
  ...

170482 ms: ../src/libutil/serialise.hh (included 201 times, avg 848 ms), included via:
  37x: command.hh installable-value.hh installables.hh built-path.hh realisation.hh hash.hh
  14x: store-api.hh nar-info.hh hash.hh
  11x: <direct include>
  7x: primops.hh eval.hh attr-set.hh nixexpr.hh value.hh source-path.hh archive.hh
  7x: libexpr.hh value.hh source-path.hh archive.hh
  6x: fetchers.hh hash.hh
  ...
```

169397 ms: ../src/libcmd/installables.hh (included 53 times, avg 3196 ms), included via:
  40x: command.hh installable-value.hh
  5x: command-installable-value.hh installable-value.hh
  3x: installable-flake.hh installable-value.hh
  2x: <direct include>
  1x: installable-derived-path.hh
  1x: installable-value.hh
  ...

159740 ms: ../src/libutil/strings.hh (included 221 times, avg 722 ms), included via:
  37x: command.hh installable-value.hh installables.hh built-path.hh realisation.hh hash.hh serialise.hh
  19x: <direct include>
  14x: store-api.hh nar-info.hh hash.hh serialise.hh
  11x: serialise.hh
  7x: primops.hh eval.hh attr-set.hh nixexpr.hh value.hh source-path.hh archive.hh serialise.hh
  7x: libexpr.hh value.hh source-path.hh archive.hh serialise.hh
  ...

156796 ms: ../src/libcmd/command.hh (included 51 times, avg 3074 ms), included via:
  42x: <direct include>
  7x: command-installable-value.hh
  2x: installable-attr-path.hh

150392 ms: ../src/libutil/types.hh (included 251 times, avg 599 ms), included via:
  36x: command.hh installable-value.hh installables.hh path.hh
  11x: file-system.hh
  10x: globals.hh
  6x: fetchers.hh
  6x: serialise.hh strings.hh error.hh
  5x: archive.hh
  ...

133101 ms: /nix/store/644b90j1vms44nr18yw3520pzkrg4dd1-boost-1.81.0-dev/include/boost/lexical_cast.hpp (included 226 times, avg 588 ms), included via
:
  37x: command.hh installable-value.hh installables.hh built-path.hh realisation.hh hash.hh serialise.hh strings.hh
  19x: file-system.hh
  11x: store-api.hh nar-info.hh hash.hh serialise.hh strings.hh

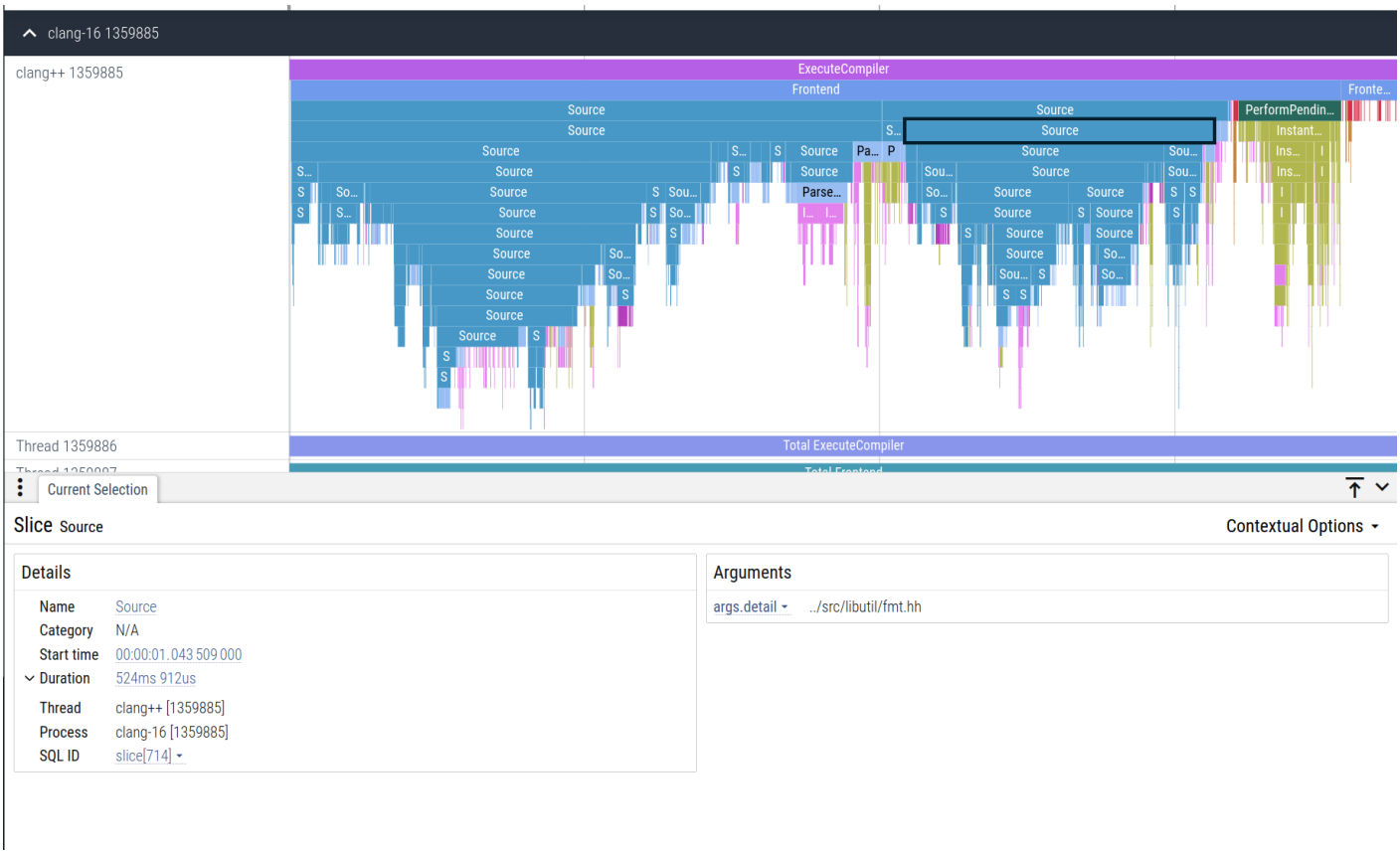
```
  7x: primops.hh eval.hh attr-set.hh nixexpr.hh value.hh source-path.hh archive.hh serialise.hh strings.hh
  7x: libexpr.hh value.hh source-path.hh archive.hh serialise.hh strings.hh
  6x: eval.hh attr-set.hh nixexpr.hh value.hh source-path.hh archive.hh serialise.hh strings.hh
  ...

132887 ms: /nix/store/h2abv2l8irqj942i5rq9wbrj42kbsh5y-gcc-12.3.0/include/c++/12.3.0/memory (included
262 times, avg 507 ms), included via:
  36x: command.hh installable-value.hh installables.hh path.hh types.hh ref.hh
  16x: gtest.h
  11x: file-system.hh types.hh ref.hh
  10x: globals.hh types.hh ref.hh
  10x: json.hpp
  6x: serialise.hh
  ...

  done in 0.6s.
```

Manually looking at traces

Note that the summary in the report can miss details like *why* one particular header is bad; to find that out, use a trace viewer to inspect the JSON trace file; we suggest `rg -t json -uu error\.hh build/ | less` to find some `.cc` trace that the bad header (in this example, `error.hh`) appears in.

You can look at individual file traces by opening some file like `build/src/libcmd/liblixcmd.so.p/command.cc.json` in https://ui.perfetto.dev or another Chrome-trace-json compatible trace viewer like Speedscope.

This will produce a flamegraph of the trace (screenshot shows Perfetto):



The most general spans of compile time are at the top, and the constituent spans are shown as you go down.

Successful build time reduction CLs

- cl/588: 15%
- cl/1351: 10%

See the build-time-optimisation Gerrit topic for more related things.

Backport guide

Don't forget, using Gerrit is a bit different than other systems.

single commits

try `git cherry-pick -x` first. if this works, excellent. if not, apply the usual cherry-picking procedures:

- track down apply failures to intermediate changes. maybe cherry-pick those first if they're not too awful, but experience shows that they usually are too awful
- anything that touches the store or contains the word `Accessor` in the vicinity probably needs to be picked about and rewritten from scratch
- *always* test ofc, even if something applies cleanly it will sometimes just fail to build (or worse, run)
- nix prs tend to have broken inner commits. it is often necessary to pick parts from later commits in a pr to fix ci, in that case note this down as `fixes taken from <commits...>`
- sometimes a commit may be so broken that it can't reasonably be fixed except by squasing it with some other commit, in that case just squash them and not it down somehow (either with multiple `cherry picked from` commit hashes or multiple commit message+cherry-pick-hash blocks, depending on whether the fix messages were any useful)

full prs

single-commit prs were mostly picked using `cherry-pick -x -m1` to keep the association with the upstream pr number for clarity. this implicitly squashes the pr into a single commit so it's only useful for single-commit prs. (some prs that have broken intermediate commits also benefit from this, but see above for that)

when pushing these to gerrit please set a topic like `backport-<pr-number>` using push options (`-o topic=backport-<pr-number>` in `git push`) to delineate one picked pr from a pr that depends on it

Misc tips

buildbot user style to make the pulsing pills bearable

```
@keyframes pulse_animation {
  0% { transform:scale(.9) }
  50% { transform:scale(1) }
  to { transform:scale(.9) }
}

.pulse {
    animation-duration: 10s !important;
}
```

FIXME: someone should PR this, now that we have the ability to patch buildbot

run all lix vm tests locally

```
tests=$(
    nix eval --json --impure \
        --apply '
            let f = n: t:
                if __isAttrs t
                then (if t.type or "" == "derivation"
                    then (if t.system == __currentSystem
                            then [ n ]
                            else [])
                    else __concatMap (m: f "${n}.${m}" t.${m}) (__attrNames t))
                else [];
            in f ".#hydraJobs.tests"
        ' \
        .#hydraJobs.tests \
        | jq -r '.[]'
```

```
)


nix build --no-link -L ${tests[@]}
```

check out current patchset of a cl by git alias

put this in a gitconfig that can configure aliases:

```
[alias]
	cocl = !\
	ps=$(\
	  ssh $(git config remote.origin.gerriturl) \
	    gerrit query --format=json --current-patch-set $1 \
	      | jq -sr .[0].currentPatchSet.ref \
	) && git fetch origin $ps && git checkout FETCH_HEAD && true
```

then run as `git cocl <cl-number>`. needs a `git config remote.origin.gerriturl gerrit.lix.systems`, or some other url that ssh can connect to. (could've extracted it from the remote url but we didn't want to do that much shell)

git stuff

git-revise

git-revise is a cool tool for splitting and shuffling commits in-memory without breaking your working tree. it's great.

It also has some broken stuff with respect to gerrit commit-msg hooks. However, this can be fixed (this is opt-in because some commit-msg hooks make unsound assumptions but the gerrit one should be fine):

```
# allow gerrit git hooks to run on git-revise
[revise "run-hooks"]
	commit-msg = true
```

Making `git clean` clean the stuff that isn't removed by `make clean`

If you don't want to use `git clean -x` to remove all git-ignored stuff, but want to remove things that are generated in Lix's build process but aren't removed by `make clean`, apply this patch: no-ignore-not-cleaned.patch

Building Locally

See hacking.md in the Lix repo for the main documentation. Extra tips can go here.

RISC-V support

Goal: install lix on a riscv64-linux system

The target is a DevTerm R-01, so it's an AllWinner D1 RISC-V processor @ 1GHz, with 1GB of memory and 32GB of microSD.

We can't run the Lix installer without building it, because there's no canned build for it. So let's try building it natively:

```
$ rustup
$ git clone https://git.lix.systems/lix-project/lix-installer
$ cd lix-installer
$ RUSTFLAGS="--cfg tokio_unstable" cargo install --path .
```

This doesn't work because there's some conditional complication^Wcompilation that doesn't cover riscv64. So we need to open `self_test.rs` and add an entry:

```
    #[cfg(all(target_os = "linux", target_arch = "riscv64"))]
    const SYSTEM: &str = "riscv64-linux";
```

At this point, it will, in principle, build. In practice, however, 1GB is just not enough RAM. If you add some swap it'll make it to the last step, but then it wants 1.5GB+ for that. I wouldn't try it on a system with less than 2GB, and ideally more.

Ok, native build is a bust unless I want to let it thrash all night. So let's cross-compile it on `ancilla`, which is, conveniently, already running nixos.

The nix-installer flake doesn't come with riscv64 cross support, and rather than try to figure it out I just winged it with nix-shell. I am skipping over a lot of false starts and blind alleys here as I ran into things like dependency crates needing a cross-compiling gcc, or rust not having a stdlib on riscv64-musl.

```
$ git clone https://git.lix.systems/lix-project/lix-installer
$ cd lix-installer
$ $EDITOR shell.nix
with import <nixpkgs> {
  crossSystem.config = "riscv64-unknown-linux-gnu";
};
mkShell {
```

```
  nativeBuildInputs = with import <unstable> {}; [ cargo rustup ];
}

$ nix-shell
[long wait for gcc to compile]

$ export RUSTUP_HOME=$PWD/.rustup-home
$ export CARGO_HOME=$PWD/.cargo-home
$ rustup default stable
$ rustup target add riscv64gc-unknown-linux-gnu
$ edit src/self_test.rs
[apply that same patch to SYSTEM]
```

The build invocation is a bit more complicated here, because we need to tell it where to find the linker:

```
$ RUSTFLAGS="--cfg tokio_unstable" cargo build \
    --target riscv64gc-unknown-linux-gnu \
    --config target.riscv64gc-unknown-linux-gnu.linker='"riscv64-unknown-linux-gnu-gcc"'
[another long wait]

$ file target/riscv64gc-unknown-linux-gnu/debug/lix-installer
target/riscv64gc-unknown-linux-gnu/debug/lix-installer:
  ELF 64-bit LSB pie executable, UCB RISC-V, RVC, double-float ABI,
  version 1 (SYSV), dynamically linked,
  interpreter /nix/store/g4xam7gr35sziib1zc033xvn1vy9gg8m-glibc-riscv64-unknown-linux-gnu-2.38-44/lib/ld-
linux-riscv64-lp64d.so.1,
  for GNU/Linux 4.15.0, with debug_info, not stripped
```

Since we couldn't do a static musl build it needs the nix ld.so, but we can get around that!

```
$ scp target/riscv64gc-unknown-linux-gnu/debug/lix-installer root@riscv:.
$ ssh root@riscv
# ./lix-installer
-bash: ./lix-installer: no such file or directory

# ldd ./lix-installer
/nix/store/.../ld-linux-riscv64-lp64d.so.1 => /lib/ld-linux-riscv64-lp64d.so.1
[other output elided]
```

```
# /lib/ld-linux-riscv64-lp64d.so.1 ./lix-installer
The Determinate Nix installer (lix variant)
[...]
```

Sadly we can't actually use it to install, because `nix_package_url` needs a default value, and on RISC-V, it doesn't have one! It's `self_test.rs` all over again except it doesn't manifest until runtime.

So, off to `src/settings.rs` we go. It doesn't need to be a valid URL, just something URL-shaped.

```
/// Default [`nix_package_url`](CommonSettings::nix_package_url) for unknown platforms
pub const NIX_UNKNOWN_PLATFORM_URL: &str =
    "https://releases.lix.systems/unknown-platform";
```

```
    #[cfg_attr(
        all(target_os = "linux", target_arch = "riscv64", feature = "cli"),
        clap(
            default_value = NIX_UNKNOWN_PLATFORM_URL,
        )
    )]
```

Rebuild, re-push, re-run:

```
# /lib/ld-linux-riscv64-lp64d.so.1 /opt/lix-installer install linux
Error:
   0: Planner error
   1: `nix-installer` does not support the `riscv64gc-unknown-linux-gnu` architecture right now
```

Ok, missed a few places in settings.rs, let's put a quick and dirty hack in there:

```
            #[cfg(target_os = "linux")]
            (_, OperatingSystem::Linux) => {
                url = NIX_UNKNOWN_PLATFORM_URL;
                nix_build_user_prefix = "nixbld";
                nix_build_user_id_base = 30000;
                nix_build_user_count = 32;
            },
```

```
            #[cfg(target_os = "linux")]
            (_, OperatingSystem::Linux) => {
                (InitSystem::Systemd, linux_detect_systemd_started().await)
```

```
        },
```

It also needs a tarball to install; jade_ kindly updated the flake for it to support riscv64, so we just check it out (or, well, check out review branch 1444) and then `nix build -L .#nix-riscv64-linux.binaryTarball` and away we go.

This, it turns out, also doesn't work, because the installer is hardcoded to expect the directory the tarball contains to start with `nix-*`. You can either unpack and repack the tarball to meet that requirement, or find all the places in lix-installer that assume that and edit them -- they're in `src/action/base/move_unpacked_nix.rs` and `src/action/base/setup_default_profile.rs`.

Finally, this particular kernel lacks `seccomp` support -- in order to get it working, I had to edit the lix (not lix-installer) `package.nix` and add `(lib.mesonEnable "seccomp-sandboxing" false)` to the meson flags.

And with that done, it works!

```
root@devterm-R01:~# uname -a && nix --version
Linux devterm-R01 5.4.61 #12 PREEMPT Wed Mar 30 14:44:22 CST 2022 riscv64 riscv64 riscv64 GNU/Linux
nix (Lix, like Nix) 2.90.0pre20240613_dirty
```

Branches

The Lix repository contains multiple releases in parallel. The branches work as follows:

- `main`. This contains *major* tags (except for 2.90 because of an early branch-off. We might fix that manually?), and is for the *next* major version of the software. This is where new development typically happens.
- `release-*`. These contain tags for `*.0` and further minor releases on a major release. We generally try to not backport things, since we would much rather get another major release out. (subject to revision; we would really like to not have LTS releases, but distro may make us do it?). These branches are *development branches* for a given release after it is released.
- (suggestion?) `stable-*` - Branch which is always pointed at the latest tag in that given major version.

Version types

- Full release, e.g. 2.90.0. This is a snapshot of HEAD that we believe is stable for release and that we have fully performed out-of-tree validation on.
- Beta release, e.g. 2.90.0-beta.1. This is something that we would consider running in more or less any environment, given that we all run HEAD ourselves. This is an arbitrarily selected snapshot of HEAD that we are deciding to produce installers for, and is not special.
- Release candidate, e.g. 2.90.0-rc1. This is something that we would just release but it needs a bit more out-of-tree validation.

Git tags

Git tags are created with the format `2.90.0`.

Docker tags

- `latest` - The latest minor version of the latest major version
- `2.90` and similar - The latest minor version of the `2.90` major release.
- `2.90.0` - Exactly `2.90.0`.

Design documents

This category contains design documents written by the Lix team, which may or may not be implemented.

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)		

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)

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

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, dubbed "Nix 2". Currently, design drafts for that language are being collected and iterated upon in https://md.darmstadt.ccc.de/nix2. (The goal is to integrate these into this page as they are being implemented.)

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.

Flake stabilisation proposal

Preface

FIXME: this page hasn't been reviewed by Lix Core team members, so it's effectively a draft/suggestion/pre-RFC/dream, whatever. It's not an official design document, but thought has been put into making it good, anyway.

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.

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"

Here are some ideas of release names:

- Bombe glacée
- Kulfi
- Tartufo
- Soft serve (with some flavour?)
  - Cherry Dip Soft Serve Vanilla
- Italian ice
- Gelato
- Granita
- Frozen yogurt
- Frozen custard
- Açaí na tigela
- Bici bici
- Sorbet
- Frozen chocolate banana
- Watermelon slush
- Freeze pop