Flake stabilisation proposal

Preface

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

Problem Statement

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

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

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

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

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

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

Flake Components

Flakes themselves have many moving parts.

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

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

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

The Plan

Stage 0: Fork the Interfaces

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

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

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

backwards-incompatible ones) in a sane manner.

Stage 1: Eating Spaghetti

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

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

Stage 2: Improving the Interfaces

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

Stage 3: Maintenance

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

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

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

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

Notes

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

Appendix A: Flakes are a broken abstraction

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

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

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

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

...
```

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

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

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

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

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

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

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

Appendix B: Some thoughts on post-stabilisation world

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

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

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

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

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

Revision #5
Created 2024-10-28 17:48:55 UTC by kfearsoff
Updated 2026-02-01 13:14:00 UTC by piegames