Bachelor Project Proposal: Nix API Tracking

Silvan Mosberger

March 26, 2019

Contents

1 Introduction

The Nix package manager (PM) allows for a pure and declarative way to specify packages and their dependencies in Nix, a simple, purely functional and dynamically typed programming language (this is a good overview of it). It has many benefits over conventional PMs, as does its derivative operating system NixOS, such as reproducibility, atomic updates and rollbacks, and more. While already over 15 years old, only in recent years, Nix has really seen a lot of growth. Nixpkgs, the main repository for Nix package declarations and NixOS, is receiving an ever-increasing influx of pull/merge requests (PR). It's great to see so many people wanting to contribute to the project, but the number of reviewers hasn't been growing at the same rate, making the number of open PR's increase over time. In order to combat this, automated tools can help out a lot, offloading work to machines. A number of automated tools are being used for nixpkgs already, such as ofborg (builds packages in PRs), r-ryantm (creates package update PRs based on repology.org) and vulnix (creates issues listing packages vulnerable to CVE's).

With the increasing size of nixpkgs, it's getting harder by the day to get an overview of it. Changes are happening all over the place without people knowing the full extent of the influence they're going to have. This in turn means that things can break when not paying attention, often due to past assumptions being broken in future versions, resulting in backwards incompatibilities. Not only can parts of nixpkgs itself break, but also third-party code depending on nixpkgs, which is actually a very big part, because every NixOS users system configuration depends on nixpkgs. When updating to a newer nixpkgs version, it is therefore not unheard of that something stops working, possibly without any easy fix in sight for the average user.

The problem is that Nix projects don't have any simple way to specify how they intend it to be used, they don't have an application programming interface (API). If we have a way to know the API of project for a specific version of it, we can also detect backwards incompatibilities between them. With this, a tool

like ABI-Laboratory could be built for Nix projects. ABI-Laboratory records changes of symbols in C API's over time for easy verification of whether changes are backwards compatible, see openssh as an example.

2 Goals

The goal of this project is therefore to build a tool to combat backwards incompatibilities in Nix projects, focusing on nixpkgs, by comparing APIs between versions. I think this would be a valuable helper for the community, because people need not spend as much care themselves for recognizing these problems, in turn hopefully accelerating development. There are multiple different parts to this project:

1. Analyse previous cases of backwards incompatibility in nixpkgs to get an idea of the biggest problems
2. Write a tool that implements a way to define/discover APIs for Nix projects, along with checks to verify the API against the codebase
3. Augment the tool with support for comparing APIs between different versions of the codebase and reporting potential backwards-incompatibilities
4. * Add GitHub integration and deploy the tool on the nixpkgs project for GitHub PRs
5. * Augment the tool with support for marking deprecation of parts of the API

As often in development, this isn't a strict sequential order but more of a feedback loop. The goals marked with * are optional ones, because even without them a useful tool can be made. For the implementation, I will be using Haskell due to its great type system and my familiarity and confidence with the language.

3 "FAQ"

3.1 How is this different from just writing test cases?

- For e.g. a list of 10000 symbols, test cases can't easily realize that a certain symbol has been removed between two versions, unless you're willing to write a test case for every symbol, which is impractical and would make maintenance only harder. The goal of this tool is roughly to write such tests for you automatically.
- Because tests are part of the project itself, any attempt to use them at checking compatibility is doomed to fail, because it can't compare between multiple versions of itself, only the current version of the project is "in scope". Any tool that intends to do such checks needs to be on a higher abstraction layer, like git, which similarly handles with multiple versions of projects.

3.2 Are there any other use cases?

- By using the tool to compare between a previous and the next stable release of a project, release notes listing incompatibilities can be generated.

4 Some Details

Nix projects typically have three aspects of publicly facing API, each of them depending on the previous one.

4.1 Parts of the API

4.1.1 Path API

A project has a directory structure with files in them. These files can become part of the API, meaning the project should guarantee that they continue to exist and have a certain structure. Since such files are the entrypoints for Nix evaluations under a project, they are necessary to be able to talk about the next point. In Nix, you can import expressions on paths with the `import` statement, e.g. to import the `default.nix` file in the nixpkgs project, you can use `import /path/to/nixpkgs/default.nix`. When `import` gets a directory as an argument, it uses the `default.nix` file in it by default, so this can be shortened to `import /path/to/nixpkgs`.

4.1.2 Expression API

Once we know what files are public API, people may rely on certain aspects of the expressions defined within them. The nixpkgs top-level default.nix file has a simplified structure as follows:

```
# A function taking an attribute set and returning an attribute set with at least the key ‘hello‘
{ ... }: {

  # Derivation/package definitions
  hello = derivation {
    # ...
  };

  # ...

}
```

Therefore to refer to the `hello` package, one can use `(import /path/to/nixpkgs {}).hello`, `{}` being the argument to the function in the file, and `.hello` for accessing the `hello` attribute of the result of the function application. Nix also has a non-trivial semantic for how functions/lambdas may be called. The API to nixpkgs both includes what packages are at which attribute paths and how functions may be called.

4.1.3 Output API

Derivations in Nix are a way to specify how to build a directory. For example the `hello` derivation, buildable by the Nix expression `(import /path/to/nixpkgs {}).hello` would have the `hello` binary in its `./bin/hello` path. This is usually also part of the API, people can refer to `"${(import /path/to/nixpkgs {}).hello}/bin/hello"` for a path to this binary, and one usually expects this to exist in future versions as well.

5 Open Questions/Future Work

- How to handle minor/major releases? Should major releases be allowed to be backwards incompatible? What changes are acceptable for minor versions? Could be enforce semver compatibility?
- Could we augment API's by taking into account what people are using the most? Would require collecting statistics, but be much more useful. We care more about when something breaks for 1000 people than for a single person. Backwards compatibility metric.
- What about forwards compatibility?

- Implement backwards compatibility tracking on the files contents in the derivation output, e.g. that C libraries keep their symbols the same (ABI-laboratory!) or that binaries keep accepting the same arguments.