April 3, 2026 - 9 min read

Making a Type Checker/LSP for Nix

0:00 / 0:09

Tix is a custom type checker/LSP for Nix. I wanted TypeScript for Nix — strong inference where possible, type annotations for the really nasty parts.

▼ Table of Contents

Tix Overview

Tix is based on Simple Sub (algebraic subtyping) + Negation types. My main goal was to make the LSP experience of Nix more on par with other modern languages. A typechecker helps track what can be autocompleted and where things are defined, so that felt like the right foundation.

Tix is not the only Nix lsp. Nil and Nixd are both great Nix LSPs. My assumption was a good type checker could provide more features and hopefully still be performant. TypeNix is a newer project with similar goals but a different approach, it translates Nix ASTs into TypeScript ASTs and reuses TypeScript's type checker. Honestly a really cool idea. Its LSP seems pretty good and seems to support overrides better than Tix currently does. With that said my gut assumption is Tix might be easier to tailor specifically for nix.

Here is what Tix can do

- Give you type errors for bad nix code
- Auto complete for pkgs in NixPkgs
- Auto complete + inline docs for Nixos config values
- Jump to def across files in your project
- Jump to def into Nixpkgs on pkgs + Nixos options

Tix is also pretty fast. A full type check of nixpkgs can be done in around 20 secs. Smaller projects like my nixos config check in 5ish seconds. I have some ideas to make this even faster...

> I used claude pretty heavily while making Tix. It started "all natural" but is now almost all Claude. If you feel strongly about that, I get it but hope you can still find Tix useful

The Type System

A very common type system people pick for functional languages is Damas-Hindley-Milner (commonly called Hindley-Milner or HM). It's somewhat simple to implement, and it is "Complete" meaning that without any type annotations we can infer the most general type of the program passed.

A Basic Implementation of HM

See this post for a great implementation overview, I will cover just the high level here.

It basically works like this (I am simplifying this heavily)

- Walk the AST of the program
  - As you walk assign unique type variables to each expression
  - Generate constraints on those type variables
    - EX: `let x = a` x and a must be the same type
    - EX: `let y = foo b`, foo must be a function that takes b as its arg
    - EX: `let z = bar.someKey` bar must be an attrset with at least the key `someKey`
- Solve all those generated constraints via unification, propagating the info gained from each constraint to the type variables involved.

I implemented HM initially and it worked great until I wanted to support union types, types whose value could be a number of different types, ie `string | int | {key: string}`. When you do inference in HM you can only equate two unknown types to be the same type, there is no kind of subtyping relationship that unions need.

Other languages that use HM support some kind of tagged union to work around this. I did not want to change the core nix language so this would not really work for Tix.

SimpleSub

Thankfully, SimpleSub is basically an extension of HM that supports subtyping. It has the same high level idea of walk the ast and generate constraints. But instead of requiring unification you can encode a subtyping relationship. For example in `let y = foo b`, instead of saying that b must be the same type as the arg of foo, b can be a subtype of the arg of foo.

This subtyping relationship is what makes union types fall out naturally. Type variables accumulate upper and lower bounds as constraints are solved, those bounds become intersections and unions in the inferred types. So if a value could be a `string` or an `int` depending on which branch was taken, that's not a type error, just a union type `string | int`.

Narrowing

Once SimpleSub was implemented the main challenge was now narrowing down unions to useful subsets when needed. For example

```
let
    foo = {name ? null}: if name != null then builtins.stringLength name else 0;
in foo {name = "John"}
```

Here the inferred type of `foo` would be `{name?: string | null } -> int`. Without a way to "narrow" the union to the non null case at the type level we would have a type error on builtins.stringLength name since its type only accepts string and not null.

To do this I added "Negation types". Basically in the type algebra we can track that something is `Not(<some inner type>)`. Then doing things like checking if something is not null or using the `builtins.is<Type>`, the type system can narrow the given type. For example

```
foo = x: # x here is a generic
  if builtins.isString x
  then { key = x; } # x here is properly treated as just a string
  else x; # x here is ~string (Not(string))
```

This way you can properly handle narrowing down unions without needing a bunch of type casting.

As of this first launch it only works within an expression, so something like

```
foo = x: let
  x_is_string = builtins.isString x;
in if x_is_string then { key = x; } else x;
```

would not narrow x in the conditional branches.

Stubs

All of the type system stuff is great but inference can't realistically be done on most "real" nix code. As [The hard part of type-checking Nix] explains, the real challenge isn't typing Nix-the-language, it's what got built on top, Nixpkgs overlays and Nixos modules. Those are extremely dynamic and rely on fix points to resolve.

So to get useful types from nixpkgs and other large dependencies I took the declaration file (`.d.ts`) idea from TypeScript and we support `tix` stub files.

They look like

```
type NixosConfig = {
  appstream: {
    ## Whether to install files to support the
    ## [AppStream metadata specification](https://www.freedesktop.org/software/appst
    @source nixpkgs:nixos/modules/config/appstream.nix:4:5
    enable: bool,
    ...
  },
  ## This option allows modules to express conditions that must
  ## hold for the evaluation of the system configuration to
  ## succeed, along with associated error messages for the user.
  @source nixpkgs:nixos/modules/misc/assertions.nix:6:5
  assertions: [{ ... }],
  boot: {
    bcache: {
      ## Whether to enable bcache mount support.
      @source nixpkgs:nixos/modules/tasks/bcache.nix:11:3
      enable: bool,
      ...
    },
```

```
  }
  # many more fields....
}

type Derivation = {
  name: string,
  pname: string,
  version: string,
  type: string,
  outPath: string,
  drvPath: string,
  system: string,
  builder: path | string,
  args: [string],
  outputs: [string],
  meta: { ... },
  ...
};

module pkgs {

  # mkDerivation accepts either an attrset or a function (finalAttrs: { ... })
  val mkDerivation :: ({ name: string, ... } | { pname: string, version: string, ...

  val lib :: Lib;

  module stdenv {
    # mkDerivation accepts either an attrset or a function (finalAttrs: { ... })
    val mkDerivation :: ({ name: string, ... } | { pname: string, version: string, .
    val cc :: Derivation;
    val shell :: path;
    val isLinux :: bool;
    val isDarwin :: bool;
    val hostPlatform :: { system: string, isLinux: bool, isDarwin: bool, isx86_64: b
    val buildPlatform :: { system: string, isLinux: bool, isDarwin: bool, ... };
    val targetPlatform :: { system: string, isLinux: bool, isDarwin: bool, ... };
  }
}
```

These are mostly intended to be auto generated when possible (**`tix stubs generate`** which LSP calls for you). The syntax mostly matches nixdoc.

These stubs allow you to do things like

```nix
let
  /**
      type: lib :: Lib
  */
  lib = import ./lib.nix;

  # type: pkgs :: Pkgs
  pkgs = import ./pkgs.nix;

  greeting = lib.strings.concatStringsSep ", " [
    "hello"
    "world"
  ];
  identity = lib.id 42;
  names = lib.lists.map (x: x.name) [
    { name = "alice"; }
    { name = "bob"; }
  ];

  drv = pkgs.stdenv.mkDerivation {
    name = "my-package";
    src = ./.;
  };
  src = pkgs.fetchFromGitHub {
    owner = "NixOS";
    repo = "nixpkgs";
    rev = "abc123";
    sha256 = "000";
  };
in
{
  inherit
    greeting
    identity
    names
    drv
    src
    ;
}

# returned type is inferred as
# { drv: Derivation, greeting: string, identity: int, names: [string], src: Derivati
```

Tix context

Type annotations are useful but having to sprinkle comments on every file in your project is tedious. Most Nix projects follow a pattern where the same parameter names (`config` , `lib` , `pkgs`) always mean the same types depending on the kind of file, NixOS modules always get `config :: NixosConfig` , callPackage files always get their params from `pkgs` , etc.

Context lets you declare that pattern once and Tix auto-applies the type annotations to the root lambda of every matching file.

Without context, you'd need annotations on every file:

```
# type: config :: NixosConfig
# type: lib :: Lib
# type: pkgs :: Pkgs
{ config, lib, pkgs, ... }:
{
  networking.firewall.enable = true;
  services.nginx.enable = true;
}
```

With context, you configure `tix.toml` once and those same files just work — no annotations needed. The file above would require no changes and would be able to do type inference correctly.

Tix has 3 builtin contexts:

- **NixOS modules** — types `config` , `lib` , and `pkgs` args
- **Home Manager modules** — same idea, with `HomeManagerConfig` instead of `NixosConfig`
- **callPackage** — for files like `{ stdenv, fetchurl, lib, ... }: <derivation>` , each param is typed from the `Pkgs` stub

The context is configured in `tix.toml` with glob patterns. For example from my nixos config repo:

```
[context.nixos]
includes = [
    "common/**/*.nix",
    "hosts/**/default.nix",
```

```
    "hosts/desktop/**/*.nix",
]
excludes = ["common/default.nix"]
stubs = ["@nixos"]

[context.home-manager]
includes = [
    "common/default.nix",
    "common/homemanager/**/*.nix",
]
excludes = ["common/homemanager/cargo.nix"]
stubs = ["@home-manager"]

[context.callpackage]
includes = ["pkgs/*.nix"]
stubs = ["@callpackage"]
```

Files matching the `includes` globs get typed params automatically — no per-file annotations required.

Running `tix init` will generate a `tix.toml` for you with a guess of what `contexts` /globs your project needs. Its not perfect but better than starting from scratch.

Stub generation

To make sure the stubs for nixpkgs (and home manager) are correct for your specific checkout of nixpkgs, Tix can auto generate them. You can add this to your `tix.toml`:

```
[stubs.generate]
nixpkgs = { expr = "(builtins.getFlake (toString ./.)).inputs.nixpkgs" }
home-manager = { expr = "(builtins.getFlake (toString ./.)).inputs.home-manager" }
```

The expr can be any nix expression that resolves to the path of the nixpkgs/home manager checkout you have. `tix check` and the LSP will run the generation and cache the results.

LSP

0:00 / 0:32

Most of the LSP features fall out for free from the type checking work. The type checker doesn't care about docs or source locations, but it's easy to layer that information on top to get hover docs, jump-to-def into nixpkgs, and autocomplete with very little extra work.

Testing

Unit tests

Most type checker tests look like this:

```
test_case!(
    overloaded_add,
    "
    let
        add = a: b: a + b;
    in
        {
            int = add 1 2;
            float = add 3.14 2;
            str = add "hi" ./test.nix;
```

```
        }
    ",
    {
        "int": (Int),
        "float": (Float),
        "str": (String)
    }
);
```

Nix snippet in, expected type out. The `test_case!` macro handles parsing, inference, and comparison. There are also `error_case!` and `diagnostic_msg!` macros for testing that bad code produces the right errors.

Property-based testing

Unit tests are great for specific cases, but a type checker has a huge input space. Property-based tests help cover that by generating random (type, nix code) pairs and verifying that inference produces the expected type.

The key idea is that the generator works *backwards*, it picks a random type first, then constructs nix source code that should produce that type. For example, to generate code that has type `[int]`, the generator might produce `[(42)]` or `[((7) + (-3))]`.

Here's a simplified view of how the generator works:

```
fn arb_nix_text() -> impl Strategy<Value = (RawTy, NixTextStr)> {
    // Leaf: pick a random primitive type, generate matching nix code
    let leaf = any::<PrimitiveTy>()
        .prop_flat_map(|prim| (Just(RawTy::Primitive(prim)), prim_ty_to_string(prim)

    // Recursively build more complex types from leaves
    leaf.prop_recursive(depth, size, branch_size, |inner| {
        let list_strat = inner.clone()
            .prop_map(|(ty, text)| (RawTy::List(Box::new(ty)), format!("[({text})]")

        let union_strat = (inner.clone(), inner.clone())
            .prop_map(|((a_ty, a_text), (b_ty, b_text))| {
                let ty = RawTy::Union(vec![a_ty, b_ty]);
                let text = format!("(if true then ({a_text}) else ({b_text}))");
                (ty, text)
            });
```

```
            prop_oneof![
                list_strat,
                func_strat(inner.clone()),
                attr_strat(inner.clone()),
                union_strat,
            ]
        })
    }
```

It starts with random primitives (`42` , `true` , `"hello"` , etc.) and recursively wraps them in lists, attrsets, lambdas, and unions — always keeping the expected type in sync with the generated code. Unions are generated via `if true then <a> else <b>` since that's the simplest way to get the type checker to infer a union.

The actual test is then straightforward:

```
proptest! {
    #[test]
    fn test_type_check((ty, text) in arb_nix_text()) {
        let root_ty = get_inferred_root(&text).normalize_vars();
        let expected = raw_to_root(&ty.normalize_vars());
        prop_assert_eq!(root_ty, expected);
    }
}
```

PBT was the main thing that made me realize I needed to move away from HM — the first attempt at union types kept finding more and more edge cases in PBT that were fundamental to how HM worked.

If you want to see the full PBT logic it lives here.

Small shill moment, need to convert these to hegel now that my job made a PBT framework...

Try it out

If you want to give Tix a a try, check out the docs for installation and setup instructions. The quick start guide will walk you through adding Tix to your project.

If you run into issues or have feature requests, feel free to open an issue on GitHub. It still has some bugs and issues but I've been using it on "real" projects for a while and its been working pretty well. Just need to fix all these type errors...

Discuss on Twitter • View on GitHub

Load Comments

← Advent Of Code 2024 in Nix - Days 04-06