Nixpkgs Manual

Version 21.05

Table of Contents

1. Preface

I. Using Nixpkgs

- 2. Global configuration
- 3. Overlays
- 4. Overriding
- 5. Functions reference

II. Standard environment

- 6. The Standard Environment
- 7. Meta-attributes
- 8. Multiple-output packages
- 9. Cross-compilation
- 10. Platform Notes

III. Builders

- 11. Fetchers
- 12. Trivial builders
- 13. Special builders
- 14. Images
- 15. Languages and frameworks
- 16. Packages

IV. Contributing to Nixpkgs

- 17. Quick Start to Adding a Package

18. Coding conventions

19. Submitting changes

20. Vulnerability Roundup

21. Reviewing contributions

22. Contributing to this documentation

List of Figures

19.1. Staging workflow

Chapter 1. Preface

Table of Contents

1.1. Overview of Nixpkgs

The Nix Packages collection (Nixpkgs) is a set of thousands of packages for the Nix package manager, released under a permissive MIT/X11 license. Packages are available for several platforms, and can be used with the Nix package manager on most GNU/Linux distributions as well as NixOS.

This manual primarily describes how to write packages for the Nix Packages collection (Nixpkgs). Thus it's mainly for packagers and developers who want to add packages to Nixpkgs. If you like to learn more about the Nix package manager and the Nix expression language, then you are kindly referred to the Nix manual. The NixOS distribution is documented in the NixOS manual.

1.1. Overview of Nixpkgs

Nix expressions describe how to build packages from source and are collected in the nixpkgs repository. Also included in the collection are Nix expressions for NixOS modules. With these expressions the Nix package manager can build binary packages.

Packages, including the Nix packages collection, are distributed through channels. The collection is distributed for users of Nix on non-NixOS distributions through the channel `nixpkgs`. Users of NixOS generally use one of the `nixos-*` channels, e.g. `nixos-19.09`, which includes all packages and modules for the stable NixOS 19.09. Stable NixOS releases are generally only given security updates. More up to date packages and modules are available via the `nixos-unstable` channel.

Both `nixos-unstable` and `nixpkgs` follow the `master` branch of the Nixpkgs repository, although both do lag the `master` branch by generally a couple of days. Updates to a channel are distributed as soon as all tests for that channel pass, e.g. this table shows the status of tests for the `nixpkgs` channel.

The tests are conducted by a cluster called Hydra, which also builds binary packages from the Nix expressions in Nixpkgs for `x86_64-linux`, `i686-linux` and `x86_64-darwin`. The binaries are made available via a binary cache.

The current Nix expressions of the channels are available in the `nixpkgs` repository in branches that correspond to the channel names (e.g. `nixos-19.09-small`).

Part I. Using Nixpkgs

Table of Contents

2. Global configuration

3. Overlays

4. Overriding

5. Functions reference

Chapter 2. Global configuration

Table of Contents

2.1. Installing broken packages

2.2. Installing packages on unsupported systems

2.3. Installing unfree packages

2.4. Installing insecure packages

2.5. Modify packages via `packageOverrides`

2.6. Declarative Package Management

Nix comes with certain defaults about what packages can and cannot be installed, based on a package's metadata. By default, Nix will prevent installation if any of the following criteria are true:

- The package is thought to be broken, and has had its `meta.broken` set to `true`.
- The package isn't intended to run on the given system, as none of its `meta.platforms` match the given system.
- The package's `meta.license` is set to a license which is considered to be unfree.
- The package has known security vulnerabilities but has not or can not be updated for some reason, and

a list of issues has been entered in to the package's `meta.knownVulnerabilities`.

Note that all this is checked during evaluation already, and the check includes any package that is evaluated. In particular, all build-time dependencies are checked. `nix-env -qa` will (attempt to) hide any packages that would be refused.

Each of these criteria can be altered in the nixpkgs configuration.

The nixpkgs configuration for a NixOS system is set in the `configuration.nix`, as in the following example:

```
{
  nixpkgs.config = {
    allowUnfree = true;
  };
}
```

However, this does not allow unfree software for individual users. Their configurations are managed separately.

A user's nixpkgs configuration is stored in a user-specific configuration file located at `~/.config/nixpkgs/config.nix`. For example:

```
{
  allowUnfree = true;
}
```

Note that we are not able to test or build unfree software on Hydra due to policy. Most unfree licenses prohibit us from either executing or distributing the software.

2.1. Installing broken packages

There are two ways to try compiling a package which has been marked as broken.

- For allowing the build of a broken package once, you can use an environment variable for a single invocation of the nix tools:

  ```
  $ export NIXPKGS_ALLOW_BROKEN=1
  ```

- For permanently allowing broken packages to be built, you may add `allowBroken = true;` to your user's configuration file, like this:

  ```
  {
    allowBroken = true;
  }
  ```

2.2. Installing packages on unsupported systems

There are also two ways to try compiling a package which has been marked as unsupported for the given system.

- For allowing the build of an unsupported package once, you can use an environment variable for a single invocation of the nix tools:

```
$ export NIXPKGS_ALLOW_UNSUPPORTED_SYSTEM=1
```

- For permanently allowing unsupported packages to be built, you may add `allowUnsupportedSystem = true;` to your user's configuration file, like this:

```
{
  allowUnsupportedSystem = true;
}
```

The difference between a package being unsupported on some system and being broken is admittedly a bit fuzzy. If a program *ought* to work on a certain platform, but doesn't, the platform should be included in `meta.platforms`, but marked as broken with e.g. `meta.broken = !hostPlatform.isWindows`. Of course, this begs the question of what "ought" means exactly. That is left to the package maintainer.

2.3. Installing unfree packages

There are several ways to tweak how Nix handles a package which has been marked as unfree.

- To temporarily allow all unfree packages, you can use an environment variable for a single invocation of the nix tools:

```
$ export NIXPKGS_ALLOW_UNFREE=1
```

- It is possible to permanently allow individual unfree packages, while still blocking unfree packages by default using the `allowUnfreePredicate` configuration option in the user configuration file.

  This option is a function which accepts a package as a parameter, and returns a boolean. The following example configuration accepts a package and always returns false:

```
{
  allowUnfreePredicate = (pkg: false);
}
```

  For a more useful example, try the following. This configuration only allows unfree packages named roon-server and visual studio code:

```
{
  allowUnfreePredicate = pkg: builtins.elem (lib.getName pkg) [
    "roon-server"
    "vscode"
  ];
}
```

- It is also possible to allow and block licenses that are specifically acceptable or not acceptable, using `allowlistedLicenses` and `blocklistedLicenses`, respectively.

  The following example configuration allowlists the licenses `amd` and `wtfpl`:

  ```
  {
    allowlistedLicenses = with lib.licenses; [ amd wtfpl ];
  }
  ```

  The following example configuration blocklists the `gpl3Only` and `agpl3Only` licenses:

  ```
  {
    blocklistedLicenses = with lib.licenses; [ agpl3Only gpl3Only ];
  }
  ```

  Note that `allowlistedLicenses` only applies to unfree licenses unless `allowUnfree` is enabled. It is not a generic allowlist for all types of licenses. `blocklistedLicenses` applies to all licenses.

A complete list of licenses can be found in the file `lib/licenses.nix` of the nixpkgs tree.

2.4. Installing insecure packages

There are several ways to tweak how Nix handles a package which has been marked as insecure.

- To temporarily allow all insecure packages, you can use an environment variable for a single invocation of the nix tools:

  ```
  $ export NIXPKGS_ALLOW_INSECURE=1
  ```

- It is possible to permanently allow individual insecure packages, while still blocking other insecure packages by default using the `permittedInsecurePackages` configuration option in the user configuration file.

  The following example configuration permits the installation of the hypothetically insecure package `hello`, version `1.2.3`:

```
{
  permittedInsecurePackages = [
    "hello-1.2.3"
  ];
}
```

- It is also possible to create a custom policy around which insecure packages to allow and deny, by overriding the `allowInsecurePredicate` configuration option.

  The `allowInsecurePredicate` option is a function which accepts a package and returns a boolean, much like `allowUnfreePredicate`.

  The following configuration example only allows insecure packages with very short names:

```
{
  allowInsecurePredicate = pkg: builtins.stringLength (lib.getName pkg) <
}
```

  Note that `permittedInsecurePackages` is only checked if `allowInsecurePredicate` is not specified.

2.5. Modify packages via `packageOverrides`

You can define a function called `packageOverrides` in your local `~/.config/nixpkgs/config.nix` to override Nix packages. It must be a function that takes pkgs as an argument and returns a modified set of packages.

```
{
  packageOverrides = pkgs: rec {
    foo = pkgs.foo.override { ... };
  };
}
```

2.6. Declarative Package Management

2.6.1. Build an environment

Using `packageOverrides`, it is possible to manage packages declaratively. This means that we can list all of our desired packages within a declarative Nix expression. For example, to have `aspell`, `bc`, `ffmpeg`, `coreutils`, `gdb`, `nixUnstable`, `emscripten`, `jq`, `nox`, and `silver-searcher`, we could use the following in `~/.config/nixpkgs/config.nix`:

```
{
```

```
  packageOverrides = pkgs: with pkgs; {
    myPackages = pkgs.buildEnv {
      name = "my-packages";
      paths = [
        aspell
        bc
        coreutils
        gdb
        ffmpeg
        nixUnstable
        emscripten
        jq
        nox
        silver-searcher
      ];
    };
  };
}
```

To install it into our environment, you can just run `nix-env -iA nixpkgs.myPackages`. If you want to load the packages to be built from a working copy of `nixpkgs` you just run `nix-env -f. -iA myPackages`. To explore what's been installed, just look through `~/.nix-profile/`. You can see that a lot of stuff has been installed. Some of this stuff is useful some of it isn't. Let's tell Nixpkgs to only link the stuff that we want:

```
{
  packageOverrides = pkgs: with pkgs; {
    myPackages = pkgs.buildEnv {
      name = "my-packages";
      paths = [
        aspell
        bc
        coreutils
        gdb
        ffmpeg
        nixUnstable
        emscripten
        jq
        nox
        silver-searcher
      ];
      pathsToLink = [ "/share" "/bin" ];
    };
```

```
  };
}
```

`pathsToLink` tells Nixpkgs to only link the paths listed which gets rid of the extra stuff in the profile. `/bin` and `/share` are good defaults for a user environment, getting rid of the clutter. If you are running on Nix on MacOS, you may want to add another path as well, `/Applications`, that makes GUI apps available.

2.6.2. Getting documentation

After building that new environment, look through `~/.nix-profile` to make sure everything is there that we wanted. Discerning readers will note that some files are missing. Look inside `~/.nix-profile/share/man/man1/` to verify this. There are no man pages for any of the Nix tools! This is because some packages like Nix have multiple outputs for things like documentation (see section 4). Let's make Nix install those as well.

```
{
  packageOverrides = pkgs: with pkgs; {
    myPackages = pkgs.buildEnv {
      name = "my-packages";
      paths = [
        aspell
        bc
        coreutils
        ffmpeg
        nixUnstable
        emscripten
        jq
        nox
        silver-searcher
      ];
      pathsToLink = [ "/share/man" "/share/doc" "/bin" ];
      extraOutputsToInstall = [ "man" "doc" ];
    };
  };
}
```

This provides us with some useful documentation for using our packages. However, if we actually want those manpages to be detected by man, we need to set up our environment. This can also be managed within Nix expressions.

```
{
  packageOverrides = pkgs: with pkgs; rec {
```

```
    myProfile = writeText "my-profile" ''
      export PATH=$HOME/.nix-profile/bin:/nix/var/nix/profiles/default/bin:
      export MANPATH=$HOME/.nix-profile/share/man:/nix/var/nix/profiles/def
    '';
    myPackages = pkgs.buildEnv {
      name = "my-packages";
      paths = [
        (runCommand "profile" {} ''
          mkdir -p $out/etc/profile.d
          cp ${myProfile} $out/etc/profile.d/my-profile.sh
        '')
        aspell
        bc
        coreutils
        ffmpeg
        man
        nixUnstable
        emscripten
        jq
        nox
        silver-searcher
      ];
      pathsToLink = [ "/share/man" "/share/doc" "/bin" "/etc" ];
      extraOutputsToInstall = [ "man" "doc" ];
    };
  };
}
```

For this to work fully, you must also have this script sourced when you are logged in. Try adding something like this to your `~/.profile` file:

```
#!/bin/sh
if [ -d $HOME/.nix-profile/etc/profile.d ]; then
  for i in $HOME/.nix-profile/etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
fi
```

Now just run `source $HOME/.profile` and you can starting loading man pages from your environment.

2.6.3. GNU info setup

Configuring GNU info is a little bit trickier than man pages. To work correctly, info needs a database to be generated. This can be done with some small modifications to our environment scripts.

```
{
  packageOverrides = pkgs: with pkgs; rec {
    myProfile = writeText "my-profile" ''
      export PATH=$HOME/.nix-profile/bin:/nix/var/nix/profiles/default/bin:
      export MANPATH=$HOME/.nix-profile/share/man:/nix/var/nix/profiles/def
      export INFOPATH=$HOME/.nix-profile/share/info:/nix/var/nix/profiles/d
    '';
    myPackages = pkgs.buildEnv {
      name = "my-packages";
      paths = [
        (runCommand "profile" {} ''
          mkdir -p $out/etc/profile.d
          cp ${myProfile} $out/etc/profile.d/my-profile.sh
        '')
        aspell
        bc
        coreutils
        ffmpeg
        man
        nixUnstable
        emscripten
        jq
        nox
        silver-searcher
        texinfoInteractive
      ];
      pathsToLink = [ "/share/man" "/share/doc" "/share/info" "/bin" "/etc"
      extraOutputsToInstall = [ "man" "doc" "info" ];
      postBuild = ''
        if [ -x $out/bin/install-info -a -w $out/share/info ]; then
          shopt -s nullglob
          for i in $out/share/info/*.info $out/share/info/*.info.gz; do
              $out/bin/install-info $i $out/share/info/dir
          done
        fi
      '';
    };
  };
```

```
}
```

`postBuild` tells Nixpkgs to run a command after building the environment. In this case, `install-info` adds the installed info pages to `dir` which is GNU info's default root node. Note that `texinfoInteractive` is added to the environment to give the `install-info` command.

Chapter 3. Overlays

Table of Contents

3.1. Installing overlays

3.2. Defining overlays

3.3. Using overlays to configure alternatives

This chapter describes how to extend and change Nixpkgs using overlays. Overlays are used to add layers in the fixed-point used by Nixpkgs to compose the set of all packages.

Nixpkgs can be configured with a list of overlays, which are applied in order. This means that the order of the overlays can be significant if multiple layers override the same package.

3.1. Installing overlays

The list of overlays can be set either explicitly in a Nix expression, or through `<nixpkgs-overlays>` or user configuration files.

3.1.1. Set overlays in NixOS or Nix expressions

On a NixOS system the value of the `nixpkgs.overlays` option, if present, is passed to the system Nixpkgs directly as an argument. Note that this does not affect the overlays for non-NixOS operations (e.g. `nix-env`), which are looked up independently.

The list of overlays can be passed explicitly when importing nixpkgs, for example `import <nixpkgs> { overlays = [ overlay1 overlay2 ]; }`.

NOTE: DO NOT USE THIS in nixpkgs. Further overlays can be added by calling the `pkgs.extend` or `pkgs.appendOverlays`, although it is often preferable to avoid these functions, because they recompute the Nixpkgs fixpoint, which is somewhat expensive to do.

3.1.2. Install overlays via configuration lookup

The list of overlays is determined as follows.

1. First, if an `overlays` argument to the Nixpkgs function itself is given, then that is used and no path

lookup will be performed.

2. Otherwise, if the Nix path entry `<nixpkgs-overlays>` exists, we look for overlays at that path, as described below.

   See the section on `NIX_PATH` in the Nix manual for more details on how to set a value for `<nixpkgs-overlays>`.

3. If one of `~/.config/nixpkgs/overlays.nix` and `~/.config/nixpkgs/overlays/` exists, then we look for overlays at that path, as described below. It is an error if both exist.

If we are looking for overlays at a path, then there are two cases:

- If the path is a file, then the file is imported as a Nix expression and used as the list of overlays.
- If the path is a directory, then we take the content of the directory, order it lexicographically, and attempt to interpret each as an overlay by:
  - Importing the file, if it is a `.nix` file.
  - Importing a top-level `default.nix` file, if it is a directory.

Because overlays that are set in NixOS configuration do not affect non-NixOS operations such as `nix-env`, the `overlays.nix` option provides a convenient way to use the same overlays for a NixOS system configuration and user configuration: the same file can be used as `overlays.nix` and imported as the value of `nixpkgs.overlays`.

3.2. Defining overlays

Overlays are Nix functions which accept two arguments, conventionally called `self` and `super`, and return a set of packages. For example, the following is a valid overlay.

```
self: super:

{
  boost = super.boost.override {
    python = self.python3;
  };
  rr = super.callPackage ./pkgs/rr {
    stdenv = self.stdenv_32bit;
  };
}
```

The first argument (`self`) corresponds to the final package set. You should use this set for the dependencies of all packages specified in your overlay. For example, all the dependencies of `rr` in the example above come from `self`, as well as the overridden dependencies used in the `boost` override.

The second argument (`super`) corresponds to the result of the evaluation of the previous stages of

Nixpkgs. It does not contain any of the packages added by the current overlay, nor any of the following overlays. This set should be used either to refer to packages you wish to override, or to access functions defined in Nixpkgs. For example, the original recipe of `boost` in the above example, comes from `super`, as well as the `callPackage` function.

The value returned by this function should be a set similar to `pkgs/top-level/all-packages.nix`, containing overridden and/or new packages.

Overlays are similar to other methods for customizing Nixpkgs, in particular the `packageOverrides` attribute described in Section 2.5, "Modify packages via `packageOverrides`". Indeed, `packageOverrides` acts as an overlay with only the `super` argument. It is therefore appropriate for basic use, but overlays are more powerful and easier to distribute.

3.3. Using overlays to configure alternatives

Certain software packages have different implementations of the same interface. Other distributions have functionality to switch between these. For example, Debian provides DebianAlternatives. Nixpkgs has what we call `alternatives`, which are configured through overlays.

3.3.1. BLAS/LAPACK

In Nixpkgs, we have multiple implementations of the BLAS/LAPACK numerical linear algebra interfaces. They are:

- OpenBLAS

  The Nixpkgs attribute is `openblas` for ILP64 (integer width = 64 bits) and `openblasCompat` for LP64 (integer width = 32 bits). `openblasCompat` is the default.

- LAPACK reference (also provides BLAS)

  The Nixpkgs attribute is `lapack-reference`.

- Intel MKL (only works on the x86_64 architecture, unfree)

  The Nixpkgs attribute is `mkl`.

- BLIS

  BLIS, available through the attribute `blis`, is a framework for linear algebra kernels. In addition, it implements the BLAS interface.

- AMD BLIS/LIBFLAME (optimized for modern AMD x86_64 CPUs)

  The AMD fork of the BLIS library, with attribute `amd-blis`, extends BLIS with optimizations for modern AMD CPUs. The changes are usually submitted to the upstream BLIS project after some time. However, AMD BLIS typically provides some performance improvements on AMD Zen CPUs. The complementary AMD LIBFLAME library, with attribute `amd-libflame`, provides a LAPACK implementation.

Introduced in PR #83888, we are able to override the `blas` and `lapack` packages to use different implementations, through the `blasProvider` and `lapackProvider` argument. This can be used to select a different provider. BLAS providers will have symlinks in `$out/lib/libblas.so.3` and `$out/lib/libcblas.so.3` to their respective BLAS libraries. Likewise, LAPACK providers will have symlinks in `$out/lib/liblapack.so.3` and `$out/lib/liblapacke.so.3` to their respective LAPACK libraries. For example, Intel MKL is both a BLAS and LAPACK provider. An overlay can be created to use Intel MKL that looks like:

```
self: super:

{
  blas = super.blas.override {
    blasProvider = self.mkl;
  };

  lapack = super.lapack.override {
    lapackProvider = self.mkl;
  };
}
```

This overlay uses Intel's MKL library for both BLAS and LAPACK interfaces. Note that the same can be accomplished at runtime using `LD_LIBRARY_PATH` of `libblas.so.3` and `liblapack.so.3`. For instance:

```
$ LD_LIBRARY_PATH=$(nix-build -A mkl)/lib:$LD_LIBRARY_PATH nix-shell -p oct
```

Intel MKL requires an `openmp` implementation when running with multiple processors. By default, `mkl` will use Intel's `iomp` implementation if no other is specified, but this is a runtime-only dependency and binary compatible with the LLVM implementation. To use that one instead, Intel recommends users set it with `LD_PRELOAD`. Note that `mkl` is only available on `x86_64-linux` and `x86_64-darwin`. Moreover, Hydra is not building and distributing pre-compiled binaries using it.

For BLAS/LAPACK switching to work correctly, all packages must depend on `blas` or `lapack`. This ensures that only one BLAS/LAPACK library is used at one time. There are two versions of BLAS/LAPACK currently in the wild, `LP64` (integer size = 32 bits) and `ILP64` (integer size = 64 bits). Some software needs special flags or patches to work with `ILP64`. You can check if `ILP64` is used in Nixpkgs with `blas.isILP64` and `lapack.isILP64`. Some software does NOT work with `ILP64`, and derivations need to specify an assertion to prevent this. You can prevent `ILP64` from being used with the following:

```
{ stdenv, blas, lapack, ... }:

assert (!blas.isILP64) && (!lapack.isILP64);
```

```
stdenv.mkDerivation {
  ...
}
```

3.3.2. Switching the MPI implementation

All programs that are built with MPI support use the generic attribute `mpi` as an input. At the moment Nixpkgs natively provides two different MPI implementations:

- Open MPI (default), attribute name `openmpi`
- MPICH, attribute name `mpich`

To provide MPI enabled applications that use `MPICH`, instead of the default `Open MPI`, simply use the following overlay:

```
self: super:

{
  mpi = self.mpich;
}
```

Chapter 4. Overriding

Table of Contents

4.1. <pkg>.override

4.2. <pkg>.overrideAttrs

4.3. <pkg>.overrideDerivation

4.4. lib.makeOverridable

Sometimes one wants to override parts of `nixpkgs`, e.g. derivation attributes, the results of derivations.

These functions are used to make changes to packages, returning only single packages. Overlays, on the other hand, can be used to combine the overridden packages across the entire package set of Nixpkgs.

4.1. <pkg>.override

The function `override` is usually available for all the derivations in the nixpkgs expression (`pkgs`).

It is used to override the arguments passed to a function.

Example usages:

```
pkgs.foo.override { arg1 = val1; arg2 = val2; ... }
```

```
import pkgs.path { overlays = [ (self: super: {
  foo = super.foo.override { barSupport = true ; };
  })]};
```

```
mypkg = pkgs.callPackage ./mypkg.nix {
  mydep = pkgs.mydep.override { ... };
  }
```

In the first example, `pkgs.foo` is the result of a function call with some default arguments, usually a derivation. Using `pkgs.foo.override` will call the same function with the given new arguments.

4.2. <pkg>.overrideAttrs

The function `overrideAttrs` allows overriding the attribute set passed to a `stdenv.mkDerivation` call, producing a new derivation based on the original one. This function is available on all derivations produced by the `stdenv.mkDerivation` function, which is most packages in the nixpkgs expression `pkgs`.

Example usage:

```
helloWithDebug = pkgs.hello.overrideAttrs (oldAttrs: rec {
  separateDebugInfo = true;
});
```

In the above example, the `separateDebugInfo` attribute is overridden to be true, thus building debug info for `helloWithDebug`, while all other attributes will be retained from the original `hello` package.

The argument `oldAttrs` is conventionally used to refer to the attr set originally passed to `stdenv.mkDerivation`.

> **Note:** Note that `separateDebugInfo` is processed only by the `stdenv.mkDerivation` function, not the generated, raw Nix derivation. Thus, using `overrideDerivation` will not work in this case, as it overrides only the attributes of the final derivation. It is for this reason that `overrideAttrs` should be preferred in (almost) all cases to `overrideDerivation`, i.e. to allow using `stdenv.mkDerivation` to process input arguments, as well as the fact that it is easier to use (you can use the same attribute names you see in your Nix code, instead of the ones generated (e.g. `buildInputs` vs `nativeBuildInputs`), and it involves less typing).

4.3. <pkg>.overrideDerivation

Warning: You should prefer `overrideAttrs` in almost all cases, see its documentation for the reasons why. `overrideDerivation` is not deprecated and will continue to work, but is less nice to use and does not have as many abilities as `overrideAttrs`.

Warning: Do not use this function in Nixpkgs as it evaluates a Derivation before modifying it, which breaks package abstraction and removes error-checking of function arguments. In addition, this evaluation-per-function application incurs a performance penalty, which can become a problem if many overrides are used. It is only intended for ad-hoc customisation, such as in `~/.config/nixpkgs/config.nix`.

The function `overrideDerivation` creates a new derivation based on an existing one by overriding the original's attributes with the attribute set produced by the specified function. This function is available on all derivations defined using the `makeOverridable` function. Most standard derivation-producing functions, such as `stdenv.mkDerivation`, are defined using this function, which means most packages in the nixpkgs expression, `pkgs`, have this function.

Example usage:

```
mySed = pkgs.gnused.overrideDerivation (oldAttrs: {
  name = "sed-4.2.2-pre";
  src = fetchurl {
    url = ftp://alpha.gnu.org/gnu/sed/sed-4.2.2-pre.tar.bz2;
    sha256 = "11nq06d131y4wmf3drm0yk502d2xc6n5qy82cg88rb9nqd2lj41k";
  };
  patches = [];
});
```

In the above example, the `name`, `src`, and `patches` of the derivation will be overridden, while all other attributes will be retained from the original derivation.

The argument `oldAttrs` is used to refer to the attribute set of the original derivation.

Note: A package's attributes are evaluated *before* being modified by the `overrideDerivation` function. For example, the `name` attribute reference in `url = "mirror://gnu/hello/${name}.tar.gz";` is filled-in *before* the `overrideDerivation` function modifies the attribute set. This means that overriding the `name` attribute, in this example, *will not* change the value of the `url` attribute. Instead, we need to override both the `name` *and* `url` attributes.

4.4. lib.makeOverridable

The function `lib.makeOverridable` is used to make the result of a function easily customizable. This utility only makes sense for functions that accept an argument set and return an attribute set.

Example usage:

```
f = { a, b }: { result = a+b; };
c = lib.makeOverridable f { a = 1; b = 2; };
```

The variable `c` is the value of the `f` function applied with some default arguments. Hence the value of `c.result` is 3, in this example.

The variable `c` however also has some additional functions, like c.override which can be used to override the default arguments. In this example the value of `(c.override { a = 4; }).result` is 6.

Chapter 5. Functions reference

Table of Contents

5.1. Nixpkgs Library Functions

5.2. Generators

5.3. Debugging Nix Expressions

5.4. prefer-remote-fetch overlay

5.5. pkgs.nix-gitignore

The nixpkgs repository has several utility functions to manipulate Nix expressions.

5.1. Nixpkgs Library Functions

Nixpkgs provides a standard library at `pkgs.lib`, or through `import <nixpkgs/lib>`.

5.1.1. Assert functions

5.1.1.1. `lib.asserts.assertMsg`

`assertMsg :: Bool -> String -> Bool`

Located at lib/asserts.nix:21 in `<nixpkgs>`.

Print a trace message if `pred` is false.

Intended to be used to augment asserts with helpful error messages.

`pred`

Condition under which the `msg` should *not* be printed.

`msg`

Message to print.

Example 5.1. Printing when the predicate is false

```
assert lib.asserts.assertMsg ("foo" == "bar") "foo is not bar, silly"
stderr> trace: foo is not bar, silly
stderr> assert failed
```

5.1.1.2. `lib.asserts.assertOneOf`

`assertOneOf :: String -> String -> StringList -> Bool`

Located at lib/asserts.nix:38 in `<nixpkgs>`.

Specialized `asserts.assertMsg` for checking if `val` is one of the elements of `xs`. Useful for checking enums.

`name`

The name of the variable the user entered `val` into, for inclusion in the error message.

`val`

The value of what the user provided, to be compared against the values in `xs`.

`xs`

The list of valid values.

Example 5.2. Ensuring a user provided a possible value

```
let sslLibrary = "bearssl";
in lib.asserts.assertOneOf "sslLibrary" sslLibrary [ "openssl" "libressl" ]
=> false
stderr> trace: sslLibrary must be one of "openssl", "libressl", but is: "be
```

5.1.2. Attribute-Set Functions

5.1.2.1. `lib.attrset.attrByPath`

`attrByPath :: [String] -> Any -> AttrSet -> Any`

Located at lib/attrsets.nix:24 in `<nixpkgs>`.

Return an attribute from within nested attribute sets.

`attrPath`

A list of strings representing the path through the nested attribute set `set`.

`default`

Default value if `attrPath` does not resolve to an existing value.

`set`

The nested attributeset to select values from.

Example 5.3. Extracting a value from a nested attribute set

```
let set = { a = { b = 3; }; };
in lib.attrsets.attrByPath [ "a" "b" ] 0 set
=> 3
```

Example 5.4. No value at the path, instead using the default

```
lib.attrsets.attrByPath [ "a" "b" ] 0 {}
=> 0
```

5.1.2.2. `lib.attrsets.hasAttrByPath`

`hasAttrByPath :: [String] -> AttrSet -> Bool`

Located at lib/attrsets.nix:42 in `<nixpkgs>`.

Determine if an attribute exists within a nested attribute set.

`attrPath`

A list of strings representing the path through the nested attribute set `set`.

`set`

The nested attributeset to check.

Example 5.5. A nested value does exist inside a set

```
lib.attrsets.hasAttrByPath
  [ "a" "b" "c" "d" ]
  { a = { b = { c = { d = 123; }; }; }; }
=> true
```

5.1.2.3. `lib.attrsets.setAttrByPath`

`setAttrByPath :: [String] -> Any -> AttrSet`

Located at lib/attrsets.nix:57 in `<nixpkgs>`.

Create a new attribute set with `value` set at the nested attribute location specified in `attrPath`.

`attrPath`

A list of strings representing the path through the nested attribute set.

`value`

The value to set at the location described by `attrPath`.

Example 5.6. Creating a new nested attribute set

```
lib.attrsets.setAttrByPath [ "a" "b" ] 3
=> { a = { b = 3; }; }
```

5.1.2.4. `lib.attrsets.getAttrFromPath`

`getAttrFromPath :: [String] -> AttrSet -> Value`

Located at lib/attrsets.nix:73 in `<nixpkgs>`.

Like Section 5.1.2.1, "`lib.attrset.attrByPath`" except without a default, and it will throw if the value doesn't exist.

`attrPath`

A list of strings representing the path through the nested attribute set `set`.

`set`

The nested attribute set to find the value in.

Example 5.7. Succesfully getting a value from an attribute set

```
lib.attrsets.getAttrFromPath [ "a" "b" ] { a = { b = 3; }; }
=> 3
```

Example 5.8. Throwing after failing to get a value from an attribute set

```
lib.attrsets.getAttrFromPath [ "x" "y" ] { }
=> error: cannot find attribute `x.y'
```

5.1.2.5. `lib.attrsets.attrVals`

`attrVals :: [String] -> AttrSet -> [Any]`

Located at lib/attrsets.nix:84 in `<nixpkgs>`.

Return the specified attributes from a set. All values must exist.

`nameList`

The list of attributes to fetch from `set`. Each attribute name must exist on the attrbitue set.

`set`

The set to get attribute values from.

Example 5.9. Getting several values from an attribute set

```
lib.attrsets.attrVals [ "a" "b" "c" ] { a = 1; b = 2; c = 3; }
=> [ 1 2 3 ]
```

Example 5.10. Getting missing values from an attribute set

```
lib.attrsets.attrVals [ "d" ] { }
error: attribute 'd' missing
```

5.1.2.6. `lib.attrsets.attrValues`

`attrValues :: AttrSet -> [Any]`

Located at lib/attrsets.nix:94 in `<nixpkgs>`.

Get all the attribute values from an attribute set.

Provides a backwards-compatible interface of `builtins.attrValues` for Nix version older than 1.8.

`attrs`
: The attribute set.

Example 5.11.

```
lib.attrsets.attrValues { a = 1; b = 2; c = 3; }
=> [ 1 2 3 ]
```

5.1.2.7. `lib.attrsets.catAttrs`

`catAttrs :: String -> [AttrSet] -> [Any]`

Located at lib/attrsets.nix:113 in `<nixpkgs>`.

Collect each attribute named `attr' from the list of attribute sets, `sets`. Sets that don't contain the named attribute are ignored.

Provides a backwards-compatible interface of `builtins.catAttrs` for Nix version older than 1.9.

`attr`
: Attribute name to select from each attribute set in `sets`.

`sets`
: The list of attribute sets to select `attr` from.

Example 5.12. Collect an attribute from a list of attribute sets.

Attribute sets which don't have the attribute are ignored.

```
catAttrs "a" [{a = 1;} {b = 0;} {a = 2;}]
=> [ 1 2 ]
```

5.1.2.8. lib.attrsets.filterAttrs

filterAttrs :: (String -> Any -> Bool) -> AttrSet -> AttrSet

Located at lib/attrsets.nix:124 in <nixpkgs>.

Filter an attribute set by removing all attributes for which the given predicate return false.

`pred`

`String -> Any -> Bool`

Predicate which returns true to include an attribute, or returns false to exclude it.

`name`

The attribute's name

`value`

The attribute's value

Returns `true` to include the attribute, `false` to exclude the attribute.

`set`

The attribute set to filter

Example 5.13. Filtering an attributeset

```
filterAttrs (n: v: n == "foo") { foo = 1; bar = 2; }
=> { foo = 1; }
```

5.1.2.9. lib.attrsets.filterAttrsRecursive

filterAttrsRecursive :: (String -> Any -> Bool) -> AttrSet -> AttrSet

Located at lib/attrsets.nix:135 in <nixpkgs>.

Filter an attribute set recursively by removing all attributes for which the given predicate return false.

`pred`

`String -> Any -> Bool`

Predicate which returns true to include an attribute, or returns false to exclude it.

`name`

The attribute's name

`value`

The attribute's value

Returns `true` to include the attribute, `false` to exclude the attribute.

`set`

The attribute set to filter

Example 5.14. Recursively filtering an attribute set

```
lib.attrsets.filterAttrsRecursive
  (n: v: v != null)
  {
    levelA = {
      example = "hi";
      levelB = {
        hello = "there";
        this-one-is-present = {
          this-is-excluded = null;
        };
      };
      this-one-is-also-excluded = null;
    };
    also-excluded = null;
  }
=> {
     levelA = {
       example = "hi";
       levelB = {
         hello = "there";
         this-one-is-present = { };
       };
     };
   }
```

5.1.2.10. `lib.attrsets.foldAttrs`

`foldAttrs :: (Any -> Any -> Any) -> Any -> [AttrSets] -> Any`

Located at lib/attrsets.nix:154 in `<nixpkgs>`.

Apply fold function to values grouped by key.

`op`

`Any -> Any -> Any`

Given a value `val` and a collector `col`, combine the two.

`val`

An attribute's value

`col`

The result of previous `op` calls with other values and `nul`.

`nul`

The null-value, the starting value.

`list_of_attrs`

A list of attribute sets to fold together by key.

Example 5.15. Combining an attribute of lists in to one attribute set

```
lib.attrsets.foldAttrs
  (n: a: [n] ++ a) []
  [
    { a = 2; b = 7; }
    { a = 3; }
    { b = 6; }
  ]
=> { a = [ 2 3 ]; b = [ 7 6 ]; }
```

5.1.2.11. `lib.attrsets.collect`

`collect :: (Any -> Bool) -> AttrSet -> [Any]`

Located at lib/attrsets.nix:178 in `<nixpkgs>`.

Recursively collect sets that verify a given predicate named `pred` from the set `attrs`. The recursion stops when `pred` returns `true`.

`pred`

`Any -> Bool`

Given an attribute's value, determine if recursion should stop.

`value`

The attribute set value.

`attrs`

The attribute set to recursively collect.

Example 5.16. Collecting all lists from an attribute set

```
lib.attrsets.collect isList { a = { b = ["b"]; }; c = [1]; }
=> [["b"] [1]]
```

Example 5.17. Collecting all attribute-sets which contain the `outPath` attribute name.

```
collect (x: x ? outPath)
  { a = { outPath = "a/"; }; b = { outPath = "b/"; }; }
=> [{ outPath = "a/"; } { outPath = "b/"; }]
```

5.1.2.12. `lib.attrsets.nameValuePair`

`nameValuePair :: String -> Any -> AttrSet`

Located at lib/attrsets.nix:212 in `<nixpkgs>`.

Utility function that creates a `{name, value}` pair as expected by `builtins.listToAttrs`.

`name`

The attribute name.

`value`

The attribute value.

Example 5.18. Creating a name value pair

```
nameValuePair "some" 6
=> { name = "some"; value = 6; }
```

5.1.2.13. `lib.attrsets.mapAttrs`

Located at lib/attrsets.nix:225 in `<nixpkgs>`.

Apply a function to each element in an attribute set, creating a new attribute set.

Provides a backwards-compatible interface of `builtins.mapAttrs` for Nix version older than 2.1.

`fn`

`String -> Any -> Any`

Given an attribute's name and value, return a new value.

`name`

The name of the attribute.

`value`

The attribute's value.

Example 5.19. Modifying each value of an attribute set

```
lib.attrsets.mapAttrs
  (name: value: name + "-" value)
  { x = "foo"; y = "bar"; }
=> { x = "x-foo"; y = "y-bar"; }
```

5.1.2.14. `lib.attrsets.mapAttrs'`

`mapAttrs' :: (String -> Any -> { name = String; value = Any }) -> AttrSet -> AttrSet`

Located at lib/attrsets.nix:239 in `<nixpkgs>`.

Like `mapAttrs`, but allows the name of each attribute to be changed in addition to the value. The applied function should return both the new name and value as a `nameValuePair`.

`fn`

`String -> Any -> { name = String; value = Any }`

Given an attribute's name and value, return a new name value pair.

`name`

The name of the attribute.

`value`

The attribute's value.

`set`

The attribute set to map over.

Example 5.20. Change the name and value of each attribute of an attribute set

```
lib.attrsets.mapAttrs' (name: value: lib.attrsets.nameValuePair ("foo_" + n
   { x = "a"; y = "b"; }
=> { foo_x = "bar-a"; foo_y = "bar-b"; }
```

5.1.2.15. `lib.attrsets.mapAttrsToList`

`mapAttrsToList :: (String -> Any -> Any) -> AttrSet -> [Any]`

Located at lib/attrsets.nix:255 in `<nixpkgs>`.

Call `fn` for each attribute in the given `set` and return the result in a list.

`fn`

`String -> Any -> Any`

Given an attribute's name and value, return a new value.

`name`

The name of the attribute.

`value`

The attribute's value.

`set`

The attribute set to map over.

Example 5.21. Combine attribute values and names in to a list

```
lib.attrsets.mapAttrsToList (name: value: "${name}=${value}")
   { x = "a"; y = "b"; }
=> [ "x=a" "y=b" ]
```

5.1.2.16. `lib.attrsets.mapAttrsRecursive`

`mapAttrsRecursive :: ([String] > Any -> Any) -> AttrSet -> AttrSet`

Located at lib/attrsets.nix:272 in `<nixpkgs>`.

Like `mapAttrs`, except that it recursively applies itself to attribute sets. Also, the first argument of the argument function is a *list* of the names of the containing attributes.

`f`

`[ String ] -> Any -> Any`

Given a list of attribute names and value, return a new value.

`name_path`

The list of attribute names to this value.

For example, the `name_path` for the `example` string in the attribute set `{ foo = { bar = "example"; }; }` is `[ "foo" "bar" ]`.

`value`

The attribute's value.

`set`

The attribute set to recursively map over.

Example 5.22. A contrived example of using `lib.attrsets.mapAttrsRecursive`

```
mapAttrsRecursive
  (path: value: concatStringsSep "-" (path ++ [value]))
  {
    n = {
      a = "A";
      m = {
        b = "B";
        c = "C";
      };
    };
    d = "D";
  }
=> {
     n = {
       a = "n-a-A";
       m = {
         b = "n-m-b-B";
```

```
          c = "n-m-c-C";
        };
      };
      d = "d-D";
    }
```

5.1.2.17. `lib.attrsets.mapAttrsRecursiveCond`

`mapAttrsRecursiveCond :: (AttrSet -> Bool) -> ([ String ] -> Any -> Any) -> AttrSet -> AttrSet`

Located at lib/attrsets.nix:293 in `<nixpkgs>`.

Like `mapAttrsRecursive`, but it takes an additional predicate function that tells it whether to recursive into an attribute set. If it returns false, `mapAttrsRecursiveCond` does not recurse, but does apply the map function. It is returns true, it does recurse, and does not apply the map function.

`cond`

`(AttrSet -> Bool)`

Determine if `mapAttrsRecursive` should recurse deeper in to the attribute set.

`attributeset`

An attribute set.

`f`

`[ String ] -> Any -> Any`

Given a list of attribute names and value, return a new value.

`name_path`

The list of attribute names to this value.

For example, the `name_path` for the `example` string in the attribute set `{ foo = { bar = "example"; }; }` is `[ "foo" "bar" ]`.

`value`

The attribute's value.

`set`

The attribute set to recursively map over.

Example 5.23. Only convert attribute values to JSON if the containing attribute set is marked for recursion

```
lib.attrsets.mapAttrsRecursiveCond
  ({ recurse ? false, ... }: recurse)
  (name: value: builtins.toJSON value)
  {
    dorecur = {
      recurse = true;
      hello = "there";
    };
    dontrecur = {
      converted-to- = "json";
    };
  }
=> {
     dorecur = {
       hello = "\"there\"";
       recurse = "true";
     };
     dontrecur = "{\"converted-to\":\"json\"}";
   }
```

5.1.2.18. `lib.attrsets.genAttrs`

`genAttrs :: [ String ] -> (String -> Any) -> AttrSet`

Located at lib/attrsets.nix:313 in `<nixpkgs>`.

Generate an attribute set by mapping a function over a list of attribute names.

`names`

Names of values in the resulting attribute set.

`f`

`String -> Any`

Takes the name of the attribute and return the attribute's value.

`name`

The name of the attribute to generate a value for.

Example 5.24. Generate an attrset based on names only

```
lib.attrsets.genAttrs [ "foo" "bar" ] (name: "x_${name}")
```

```
=> { foo = "x_foo"; bar = "x_bar"; }
```

5.1.2.19. lib.attrsets.isDerivation

isDerivation :: Any -> Bool

Located at lib/attrsets.nix:327 in <nixpkgs>.

Check whether the argument is a derivation. Any set with `{ type = "derivation"; }` counts as a derivation.

`value`

The value which is possibly a derivation.

Example 5.25. A package is a derivation

```
lib.attrsets.isDerivation (import <nixpkgs> {}).ruby
=> true
```

Example 5.26. Anything else is not a derivation

```
lib.attrsets.isDerivation "foobar"
=> false
```

5.1.2.20. lib.attrsets.toDerivation

toDerivation :: Path -> Derivation

Located at lib/attrsets.nix:330 in <nixpkgs>.

Converts a store path to a fake derivation.

`path`

A store path to convert to a derivation.

5.1.2.21. `lib.attrsets.optionalAttrs`

`optionalAttrs :: Bool -> AttrSet`

Located at lib/attrsets.nix:353 in `<nixpkgs>`.

Conditionally return an attribute set or an empty attribute set.

`cond`

Condition under which the `as` attribute set is returned.

`as`

The attribute set to return if `cond` is true.

Example 5.27. Return the provided attribute set when `cond` is true

```
lib.attrsets.optionalAttrs true { my = "set"; }
=> { my = "set"; }
```

Example 5.28. Return an empty attribute set when `cond` is false

```
lib.attrsets.optionalAttrs false { my = "set"; }
=> { }
```

5.1.2.22. `lib.attrsets.zipAttrsWithNames`

`zipAttrsWithNames :: [ String ] -> (String -> [ Any ] -> Any) -> [ AttrSet ] -> AttrSet`

Located at lib/attrsets.nix:363 in `<nixpkgs>`.

Merge sets of attributes and use the function `f` to merge attribute values where the attribute name is in `names`.

`names`

A list of attribute names to zip.

`f`

`(String -> [ Any ] -> Any`

Accepts an attribute name, all the values, and returns a combined value.

`name`

The name of the attribute each value came from.

`vs`

A list of values collected from the list of attribute sets.

`sets`

A list of attribute sets to zip together.

Example 5.29. Summing a list of attribute sets of numbers

```
lib.attrsets.zipAttrsWithNames
  [ "a" "b" ]
  (name: vals: "${name} ${toString (builtins.foldl' (a: b: a + b) 0 vals)}"
  [
    { a = 1; b = 1; c = 1; }
    { a = 10; }
    { b = 100; }
    { c = 1000; }
  ]
=> { a = "a 11"; b = "b 101"; }
```

5.1.2.23. `lib.attrsets.zipAttrsWith`

`zipAttrsWith :: (String -> [ Any ] -> Any) -> [ AttrSet ] -> AttrSet`

Located at lib/attrsets.nix:378 in `<nixpkgs>`.

Merge sets of attributes and use the function `f` to merge attribute values. Similar to Section 5.1.2.22, "`lib.attrsets.zipAttrsWithNames`" where all key names are passed for `names`.

`f`

`(String -> [ Any ] -> Any`

Accepts an attribute name, all the values, and returns a combined value.

`name`

The name of the attribute each value came from.

`vs`

A list of values collected from the list of attribute sets.

`sets`

A list of attribute sets to zip together.

Example 5.30. Summing a list of attribute sets of numbers

```
lib.attrsets.zipAttrsWith
  (name: vals: "${name} ${toString (builtins.foldl' (a: b: a + b) 0 vals)}"
  [
    { a = 1; b = 1; c = 1; }
    { a = 10; }
    { b = 100; }
    { c = 1000; }
  ]
=> { a = "a 11"; b = "b 101"; c = "c 1001"; }
```

5.1.2.24. `lib.attrsets.zipAttrs`

`zipAttrsWith :: [ AttrSet ] -> AttrSet`

Located at lib/attrsets.nix:385 in `<nixpkgs>`.

Merge sets of attributes and combine each attribute value in to a list. Similar to Section 5.1.2.23, "`lib.attrsets.zipAttrsWith`" where the merge function returns a list of all values.

`sets`

A list of attribute sets to zip together.

Example 5.31. Combining a list of attribute sets

```
lib.attrsets.zipAttrs
  [
    { a = 1; b = 1; c = 1; }
    { a = 10; }
    { b = 100; }
    { c = 1000; }
  ]
=> { a = [ 1 10 ]; b = [ 1 100 ]; c = [ 1 1000 ]; }
```

5.1.2.25. lib.attrsets.recursiveUpdateUntil

recursiveUpdateUntil :: ([String] -> AttrSet -> AttrSet -> Bool) -> AttrSet -> AttrSet -> AttrSet

Located at lib/attrsets.nix:415 in <nixpkgs>.

Does the same as the update operator // except that attributes are merged until the given predicate is verified. The predicate should accept 3 arguments which are the path to reach the attribute, a part of the first attribute set and a part of the second attribute set. When the predicate is verified, the value of the first attribute set is replaced by the value of the second attribute set.

pred
: [String] -> AttrSet -> AttrSet -> Bool

path
: The path to the values in the left and right hand sides.

l
: The left hand side value.

r
: The right hand side value.

lhs
: The left hand attribute set of the merge.

rhs
: The right hand attribute set of the merge.

Example 5.32. Recursively merging two attribute sets

```
lib.attrsets.recursiveUpdateUntil (path: l: r: path == ["foo"])
  {
    # first attribute set
    foo.bar = 1;
    foo.baz = 2;
    bar = 3;
  }
  {
    #second attribute set
    foo.bar = 1;
    foo.quz = 2;
    baz = 4;
```

```
  }
=> {
  foo.bar = 1; # 'foo.*' from the second set
  foo.quz = 2; #
  bar = 3;     # 'bar' from the first set
  baz = 4;     # 'baz' from the second set
}
```

5.1.2.26. `lib.attrsets.recursiveUpdate`

`recursiveUpdate :: AttrSet -> AttrSet -> AttrSet`

Located at lib/attrsets.nix:446 in `<nixpkgs>`.

A recursive variant of the update operator `//`. The recursion stops when one of the attribute values is not an attribute set, in which case the right hand side value takes precedence over the left hand side value.

`lhs`
: The left hand attribute set of the merge.

`rhs`
: The right hand attribute set of the merge.

Example 5.33. Recursively merging two attribute sets

```
recursiveUpdate
  {
    boot.loader.grub.enable = true;
    boot.loader.grub.device = "/dev/hda";
  }
  {
    boot.loader.grub.device = "";
  }
=> {
  boot.loader.grub.enable = true;
  boot.loader.grub.device = "";
}
```

5.1.2.27. `lib.attrsets.recurseIntoAttrs`

`recurseIntoAttrs :: AttrSet -> AttrSet`

Located at lib/attrsets.nix:505 in `<nixpkgs>`.

Make various Nix tools consider the contents of the resulting attribute set when looking for what to build, find, etc.

This function only affects a single attribute set; it does not apply itself recursively for nested attribute sets.

`attrs`
: An attribute set to scan for derivations.

Example 5.34. Making Nix look inside an attribute set

```
{ pkgs ? import <nixpkgs> {} }:
{
  myTools = pkgs.lib.recurseIntoAttrs {
    inherit (pkgs) hello figlet;
  };
}
```

5.1.2.28. `lib.attrsets.cartesianProductOfSets`

`cartesianProductOfSets :: AttrSet -> [ AttrSet ]`

Located at lib/attrsets.nix:197 in `<nixpkgs>`.

Return the cartesian product of attribute set value combinations.

`set`
: An attribute set with attributes that carry lists of values.

Example 5.35. Creating the cartesian product of a list of attribute values

```
cartesianProductOfSets { a = [ 1 2 ]; b = [ 10 20 ]; }
=> [
     { a = 1; b = 10; }
     { a = 1; b = 20; }
     { a = 2; b = 10; }
     { a = 2; b = 20; }
   ]
```

5.1.3. String manipulation functions

5.1.3.1. `lib.strings.concatStrings`

`concatStrings :: [string] -> string`

Concatenate a list of strings.

Example 5.36. `lib.strings.concatStrings` usage example

```
concatStrings ["foo" "bar"]
=> "foobar"
```

Located at lib/strings.nix:43 in `<nixpkgs>`.

5.1.3.2. `lib.strings.concatMapStrings`

`concatMapStrings :: (a -> string) -> [a] -> string`

Map a function over a list and concatenate the resulting strings.

`f`

Function argument

`list`

Function argument

Example 5.37. `lib.strings.concatMapStrings` usage example

```
concatMapStrings (x: "a" + x) ["foo" "bar"]
=> "afooabar"
```

Located at lib/strings.nix:53 in `<nixpkgs>`.

5.1.3.3. `lib.strings.concatImapStrings`

`concatImapStrings :: (int -> a -> string) -> [a] -> string`

Like `concatMapStrings` except that the f functions also gets the position as a parameter.

`f`

Function argument

`list`

Function argument

Example 5.38. `lib.strings.concatImapStrings` usage example

```
concatImapStrings (pos: x: "${toString pos}-${x}") ["foo" "bar"]
=> "1-foo2-bar"
```

Located at lib/strings.nix:64 in `<nixpkgs>`.

5.1.3.4. `lib.strings.intersperse`

`intersperse :: a -> [a] -> [a]`

Place an element between each element of a list

`separator`

Separator to add between elements

`list`

Input list

Example 5.39. `lib.strings.intersperse` usage example

```
intersperse "/" ["usr" "local" "bin"]
=> ["usr" "/" "local" "/" "bin"].
```

Located at lib/strings.nix:74 in `<nixpkgs>`.

5.1.3.5. `lib.strings.concatStringsSep`

`concatStringsSep :: string -> [string] -> string`

Concatenate a list of strings with a separator between each element

Example 5.40. `lib.strings.concatStringsSep` usage example

```
concatStringsSep "/" ["usr" "local" "bin"]
=> "usr/local/bin"
```

Located at lib/strings.nix:91 in `<nixpkgs>`.

5.1.3.6. `lib.strings.concatMapStringsSep`

`concatMapStringsSep :: string -> (string -> string) -> [string] -> string`

Maps a function over a list of strings and then concatenates the result with the specified separator interspersed between elements.

`sep`
: Separator to add between elements

`f`
: Function to map over the list

`list`
: List of input strings

Example 5.41. `lib.strings.concatMapStringsSep` usage example

```
concatMapStringsSep "-" (x: toUpper x)  ["foo" "bar" "baz"]
=> "FOO-BAR-BAZ"
```

Located at lib/strings.nix:104 in `<nixpkgs>`.

5.1.3.7. `lib.strings.concatImapStringsSep`

`concatIMapStringsSep :: string -> (int -> string -> string) -> [string] -> string`

Same as `concatMapStringsSep`, but the mapping function additionally receives the position of its argument.

`sep`
: Separator to add between elements

`f`
: Function that receives elements and their positions

`list`

List of input strings

Example 5.42. `lib.strings.concatImapStringsSep` usage example

```
concatImapStringsSep "-" (pos: x: toString (x / pos)) [ 6 6 6 ]
=> "6-3-2"
```

Located at lib/strings.nix:121 in `<nixpkgs>`.

5.1.3.8. `lib.strings.makeSearchPath`

`makeSearchPath :: string -> [string] -> string`

Construct a Unix-style, colon-separated search path consisting of the given `subDir` appended to each of the given paths.

`subDir`

Directory name to append

`paths`

List of base paths

Example 5.43. `lib.strings.makeSearchPath` usage example

```
makeSearchPath "bin" ["/root" "/usr" "/usr/local"]
=> "/root/bin:/usr/bin:/usr/local/bin"
makeSearchPath "bin" [""]
=> "/bin"
```

Located at lib/strings.nix:140 in `<nixpkgs>`.

5.1.3.9. `lib.strings.makeSearchPathOutput`

`string -> string -> [package] -> string`

Construct a Unix-style search path by appending the given `subDir` to the specified `output` of each of the packages. If no output by the given name is found, fallback to `.out` and then to the default.

`output`

Package output to use

`subDir`

Directory name to append

`pkgs`

List of packages

Example 5.44. `lib.strings.makeSearchPathOutput` usage example

```
makeSearchPathOutput "dev" "bin" [ pkgs.openssl pkgs.zlib ]
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-1.0.1r-dev/bin:/nix
```

Located at lib/strings.nix:158 in `<nixpkgs>`.

5.1.3.10. `lib.strings.makeLibraryPath`

Construct a library search path (such as RPATH) containing the libraries for a set of packages

Example 5.45. `lib.strings.makeLibraryPath` usage example

```
makeLibraryPath [ "/usr" "/usr/local" ]
=> "/usr/lib:/usr/local/lib"
pkgs = import <nixpkgs> { }
makeLibraryPath [ pkgs.openssl pkgs.zlib ]
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-1.0.1r/lib:/nix/sto
```

Located at lib/strings.nix:176 in `<nixpkgs>`.

5.1.3.11. `lib.strings.makeBinPath`

Construct a binary search path (such as $PATH) containing the binaries for a set of packages.

Example 5.46. `lib.strings.makeBinPath` usage example

```
makeBinPath ["/root" "/usr" "/usr/local"]
=> "/root/bin:/usr/bin:/usr/local/bin"
```

Located at lib/strings.nix:185 in `<nixpkgs>`.

5.1.3.12. `lib.strings.optionalString`

`optionalString :: bool -> string -> string`

Depending on the boolean `cond', return either the given string or the empty string. Useful to concatenate against a bigger string.

`cond`

Condition

`string`

String to return if condition is true

Example 5.47. `lib.strings.optionalString` usage example

```
optionalString true "some-string"
=> "some-string"
optionalString false "some-string"
=> ""
```

Located at lib/strings.nix:198 in `<nixpkgs>`.

5.1.3.13. `lib.strings.hasPrefix`

`hasPrefix :: string -> string -> bool`

Determine whether a string has given prefix.

`pref`

Prefix to check for

`str`

Input string

Example 5.48. `lib.strings.hasPrefix` usage example

```
hasPrefix "foo" "foobar"
=> true
hasPrefix "foo" "barfoo"
=> false
```

Located at lib/strings.nix:214 in `<nixpkgs>`.

5.1.3.14. `lib.strings.hasSuffix`

`hasSuffix :: string -> string -> bool`

Determine whether a string has given suffix.

`suffix`
: Suffix to check for

`content`
: Input string

Example 5.49. `lib.strings.hasSuffix` usage example

```
hasSuffix "foo" "foobar"
=> false
hasSuffix "foo" "barfoo"
=> true
```

Located at lib/strings.nix:230 in `<nixpkgs>`.

5.1.3.15. `lib.strings.hasInfix`

`hasInfix :: string -> string -> bool`

Determine whether a string contains the given infix

`infix`
: Function argument

`content`
: Function argument

Example 5.50. `lib.strings.hasInfix` usage example

```
hasInfix "bc" "abcd"
=> true
hasInfix "ab" "abcd"
=> true
hasInfix "cd" "abcd"
=> true
hasInfix "foo" "abcd"
=> false
```

Located at lib/strings.nix:255 in `<nixpkgs>`.

5.1.3.16. `lib.strings.stringToCharacters`

`stringToCharacters :: string -> [string]`

Convert a string to a list of characters (i.e. singleton strings). This allows you to, e.g., map a function over each character. However, note that this will likely be horribly inefficient; Nix is not a general purpose programming language. Complex string manipulations should, if appropriate, be done in a derivation. Also note that Nix treats strings as a list of bytes and thus doesn't handle unicode.

`s`

Function argument

Example 5.51. `lib.strings.stringToCharacters` usage example

```
stringToCharacters ""
=> [ ]
stringToCharacters "abc"
=> [ "a" "b" "c" ]
stringToCharacters "💩"
=> [ "<?>" "<?>" "<?>" "<?>" ]
```

Located at lib/strings.nix:279 in `<nixpkgs>`.

5.1.3.17. `lib.strings.stringAsChars`

`stringAsChars :: (string -> string) -> string -> string`

Manipulate a string character by character and replace them by strings before concatenating the results.

`f`

Function to map over each individual character

`s`

Input string

Example 5.52. `lib.strings.stringAsChars` usage example

```
stringAsChars (x: if x == "a" then "i" else x) "nax"
=> "nix"
```

Located at lib/strings.nix:291 in <nixpkgs>.

5.1.3.18. lib.strings.escape

escape :: [string] -> string -> string

Escape occurrence of the elements of `list` in `string` by prefixing it with a backslash.

`list`

Function argument

Example 5.53. lib.strings.escape usage example

```
escape ["(" ")"] "(foo)"
=> "\\(foo\\)"
```

Located at lib/strings.nix:308 in <nixpkgs>.

5.1.3.19. lib.strings.escapeShellArg

escapeShellArg :: string -> string

Quote string to be used safely within the Bourne shell.

`arg`

Function argument

Example 5.54. lib.strings.escapeShellArg usage example

```
escapeShellArg "esc'ape\nme"
=> "'esc'\\''ape\nme'"
```

Located at lib/strings.nix:318 in <nixpkgs>.

5.1.3.20. lib.strings.escapeShellArgs

escapeShellArgs :: [string] -> string

Quote all arguments to be safely passed to the Bourne shell.

Example 5.55. lib.strings.escapeShellArgs usage example

```
escapeShellArgs ["one" "two three" "four'five"]
=> "'one' 'two three' 'four'\\''five'"
```

Located at lib/strings.nix:328 in `<nixpkgs>`.

5.1.3.21. `lib.strings.escapeNixString`

`string -> string`

Turn a string into a Nix expression representing that string

`s`

Function argument

Example 5.56. `lib.strings.escapeNixString` usage example

```
escapeNixString "hello\${}\n"
=> "\"hello\\\${}\\n\""
```

Located at lib/strings.nix:338 in `<nixpkgs>`.

5.1.3.22. `lib.strings.escapeRegex`

`string -> string`

Turn a string into an exact regular expression

Example 5.57. `lib.strings.escapeRegex` usage example

```
escapeRegex "[^a-z]*"
=> "\\[\\^a-z]\\*"
```

Located at lib/strings.nix:348 in `<nixpkgs>`.

5.1.3.23. `lib.strings.escapeNixIdentifier`

`string -> string`

Quotes a string if it can't be used as an identifier directly.

`s`

Function argument

Example 5.58. `lib.strings.escapeNixIdentifier` usage example

```
escapeNixIdentifier "hello"
=> "hello"
escapeNixIdentifier "0abc"
=> "\"0abc\""
```

Located at lib/strings.nix:360 in `<nixpkgs>`.

5.1.3.24. `lib.strings.toLower`

`toLower :: string -> string`

Converts an ASCII string to lower-case.

Example 5.59. `lib.strings.toLower` usage example

```
toLower "HOME"
=> "home"
```

Located at lib/strings.nix:391 in `<nixpkgs>`.

5.1.3.25. `lib.strings.toUpper`

`toUpper :: string -> string`

Converts an ASCII string to upper-case.

Example 5.60. `lib.strings.toUpper` usage example

```
toUpper "home"
=> "HOME"
```

Located at lib/strings.nix:401 in `<nixpkgs>`.

5.1.3.26. `lib.strings.addContextFrom`

Appends string context from another string. This is an implementation detail of Nix.

Strings in Nix carry an invisible `context` which is a list of strings representing store paths. If the string is later used in a derivation attribute, the derivation will properly populate the inputDrvs and inputSrcs.

a
: Function argument

b
: Function argument

Example 5.61. `lib.strings.addContextFrom` usage example

```
pkgs = import <nixpkgs> { };
addContextFrom pkgs.coreutils "bar"
=> "bar"
```

Located at lib/strings.nix:416 in `<nixpkgs>`.

5.1.3.27. `lib.strings.splitString`

Cut a string with a separator and produces a list of strings which were separated by this separator.

`_sep`
: Function argument

`_s`
: Function argument

Example 5.62. `lib.strings.splitString` usage example

```
splitString "." "foo.bar.baz"
=> [ "foo" "bar" "baz" ]
splitString "/" "/usr/local/bin"
=> [ "" "usr" "local" "bin" ]
```

Located at lib/strings.nix:427 in `<nixpkgs>`.

5.1.3.28. `lib.strings.removePrefix`

`string -> string -> string`

Return a string without the specified prefix, if the prefix matches.

`prefix`

Prefix to remove if it matches

`str`

Input string

Example 5.63. `lib.strings.removePrefix` usage example

```
removePrefix "foo." "foo.bar.baz"
=> "bar.baz"
removePrefix "xxx" "foo.bar.baz"
=> "foo.bar.baz"
```

Located at lib/strings.nix:445 in `<nixpkgs>`.

5.1.3.29. `lib.strings.removeSuffix`

`string -> string -> string`

Return a string without the specified suffix, if the suffix matches.

`suffix`

Suffix to remove if it matches

`str`

Input string

Example 5.64. `lib.strings.removeSuffix` usage example

```
removeSuffix "front" "homefront"
=> "home"
removeSuffix "xxx" "homefront"
=> "homefront"
```

Located at lib/strings.nix:469 in `<nixpkgs>`.

5.1.3.30. `lib.strings.versionOlder`

Return true if string v1 denotes a version older than v2.

`v1`

Function argument

v2

Function argument

Example 5.65. `lib.strings.versionOlder` usage example

```
versionOlder "1.1" "1.2"
=> true
versionOlder "1.1" "1.1"
=> false
```

Located at lib/strings.nix:491 in `<nixpkgs>`.

5.1.3.31. `lib.strings.versionAtLeast`

Return true if string v1 denotes a version equal to or newer than v2.

v1

Function argument

v2

Function argument

Example 5.66. `lib.strings.versionAtLeast` usage example

```
versionAtLeast "1.1" "1.0"
=> true
versionAtLeast "1.1" "1.1"
=> true
versionAtLeast "1.1" "1.2"
=> false
```

Located at lib/strings.nix:503 in `<nixpkgs>`.

5.1.3.32. `lib.strings.getName`

This function takes an argument that's either a derivation or a derivation's "name" attribute and extracts the name part from that argument.

x

Function argument

Example 5.67. `lib.strings.getName` usage example

```
getName "youtube-dl-2016.01.01"
=> "youtube-dl"
getName pkgs.youtube-dl
=> "youtube-dl"
```

Located at lib/strings.nix:515 in `<nixpkgs>`.

5.1.3.33. `lib.strings.getVersion`

This function takes an argument that's either a derivation or a derivation's "name" attribute and extracts the version part from that argument.

`x`

Function argument

Example 5.68. `lib.strings.getVersion` usage example

```
getVersion "youtube-dl-2016.01.01"
=> "2016.01.01"
getVersion pkgs.youtube-dl
=> "2016.01.01"
```

Located at lib/strings.nix:532 in `<nixpkgs>`.

5.1.3.34. `lib.strings.nameFromURL`

Extract name with version from URL. Ask for separator which is supposed to start extension.

`url`

Function argument

`sep`

Function argument

Example 5.69. `lib.strings.nameFromURL` usage example

```
nameFromURL "https://nixos.org/releases/nix/nix-1.7/nix-1.7-x86_64-linux.ta
=> "nix"
nameFromURL "https://nixos.org/releases/nix/nix-1.7/nix-1.7-x86_64-linux.ta
=> "nix-1.7-x86"
```

Located at lib/strings.nix:548 in `<nixpkgs>`.

5.1.3.35. `lib.strings.enableFeature`

Create an --{enable,disable}-<feat> string that can be passed to standard GNU Autoconf scripts.

`enable`
: Function argument

`feat`
: Function argument

Example 5.70. `lib.strings.enableFeature` usage example

```
enableFeature true "shared"
=> "--enable-shared"
enableFeature false "shared"
=> "--disable-shared"
```

Located at lib/strings.nix:564 in `<nixpkgs>`.

5.1.3.36. `lib.strings.enableFeatureAs`

Create an --{enable-<feat>=<value>,disable-<feat>} string that can be passed to standard GNU Autoconf scripts.

`enable`
: Function argument

`feat`
: Function argument

`value`
: Function argument

Example 5.71. `lib.strings.enableFeatureAs` usage example

```
enableFeatureAs true "shared" "foo"
=> "--enable-shared=foo"
enableFeatureAs false "shared" (throw "ignored")
=> "--disable-shared"
```

Located at lib/strings.nix:577 in `<nixpkgs>`.

5.1.3.37. `lib.strings.withFeature`

Create an --{with,without}-<feat> string that can be passed to standard GNU Autoconf scripts.

`with_`
: Function argument

`feat`
: Function argument

Example 5.72. `lib.strings.withFeature` usage example

```
withFeature true "shared"
=> "--with-shared"
withFeature false "shared"
=> "--without-shared"
```

Located at lib/strings.nix:588 in `<nixpkgs>`.

5.1.3.38. `lib.strings.withFeatureAs`

Create an --{with-<feat>=<value>,without-<feat>} string that can be passed to standard GNU Autoconf scripts.

`with_`
: Function argument

`feat`
: Function argument

`value`
: Function argument

Example 5.73. `lib.strings.withFeatureAs` usage example

```
withFeatureAs true "shared" "foo"
=> "--with-shared=foo"
withFeatureAs false "shared" (throw "ignored")
=> "--without-shared"
```

Located at lib/strings.nix:601 in `<nixpkgs>`.

5.1.3.39. `lib.strings.fixedWidthString`

`fixedWidthString :: int -> string -> string -> string`

Create a fixed width string with additional prefix to match required width.

This function will fail if the input string is longer than the requested length.

`width`
: Function argument

`filler`
: Function argument

`str`
: Function argument

Example 5.74. `lib.strings.fixedWidthString` usage example

```
fixedWidthString 5 "0" (toString 15)
=> "00015"
```

Located at lib/strings.nix:615 in `<nixpkgs>`.

5.1.3.40. `lib.strings.fixedWidthNumber`

Format a number adding leading zeroes up to fixed width.

`width`
: Function argument

`n`
: Function argument

Example 5.75. `lib.strings.fixedWidthNumber` usage example

```
fixedWidthNumber 5 15
=> "00015"
```

Located at lib/strings.nix:632 in `<nixpkgs>`.

5.1.3.41. `lib.strings.floatToString`

Convert a float to a string, but emit a warning when precision is lost during the conversion

`float`

Function argument

Example 5.76. `lib.strings.floatToString` usage example

```
floatToString 0.000001
=> "0.000001"
floatToString 0.0000001
=> trace: warning: Imprecise conversion from float to string 0.000000
"0.000000"
```

Located at lib/strings.nix:644 in `<nixpkgs>`.

5.1.3.42. `lib.strings.isCoercibleToString`

Check whether a value can be coerced to a string

x

Function argument

Located at lib/strings.nix:651 in `<nixpkgs>`.

5.1.3.43. `lib.strings.isStorePath`

Check whether a value is a store path.

x

Function argument

Example 5.77. `lib.strings.isStorePath` usage example

```
isStorePath "/nix/store/d945ibfx9x185xf04b890y4f9g3cbb63-python-2.7.11/bin/
=> false
isStorePath "/nix/store/d945ibfx9x185xf04b890y4f9g3cbb63-python-2.7.11"
=> true
isStorePath pkgs.python
=> true
isStorePath [] || isStorePath 42 || isStorePath {} || …
=> false
```

Located at lib/strings.nix:669 in `<nixpkgs>`.

5.1.3.44. `lib.strings.toInt`

`string -> int`

Parse a string as an int.

`str`

Function argument

Example 5.78. `lib.strings.toInt` usage example

```
toInt "1337"
=> 1337
toInt "-4"
=> -4
toInt "3.14"
=> error: floating point JSON numbers are not supported
```

Located at lib/strings.nix:690 in `<nixpkgs>`.

5.1.3.45. `lib.strings.readPathsFromFile`

Read a list of paths from `file`, relative to the `rootPath`. Lines beginning with `#` are treated as comments and ignored. Whitespace is significant.

NOTE: This function is not performant and should be avoided.

Example 5.79. `lib.strings.readPathsFromFile` usage example

```
readPathsFromFile /prefix
./pkgs/development/libraries/qt-5/5.4/qtbase/series
=> [ "/prefix/dlopen-resolv.patch" "/prefix/tzdir.patch"
"/prefix/dlopen-libXcursor.patch" "/prefix/dlopen-openssl.patch"
"/prefix/dlopen-dbus.patch" "/prefix/xdg-config-dirs.patch"
"/prefix/nix-profiles-library-paths.patch"
"/prefix/compose-search-path.patch" ]
```

Located at lib/strings.nix:711 in `<nixpkgs>`.

5.1.3.46. `lib.strings.fileContents`

`fileContents :: path -> string`

Read the contents of a file removing the trailing \n

`file`
: Function argument

Example 5.80. `lib.strings.fileContents` usage example

```
$ echo "1.0" > ./version

fileContents ./version
=> "1.0"
```

Located at lib/strings.nix:731 in `<nixpkgs>`.

5.1.3.47. `lib.strings.sanitizeDerivationName`

`sanitizeDerivationName :: String -> String`

Creates a valid derivation name from a potentially invalid one.

`string`
: Function argument

Example 5.81. `lib.strings.sanitizeDerivationName` usage example

```
sanitizeDerivationName "../hello.bar # foo"
=> "-hello.bar-foo"
sanitizeDerivationName ""
=> "unknown"
sanitizeDerivationName pkgs.hello
=> "-nix-store-2g75chlbpxlrqn15zlby2dfh8hr9qwbk-hello-2.10"
```

Located at lib/strings.nix:746 in `<nixpkgs>`.

5.1.4. Miscellaneous functions

5.1.4.1. `lib.trivial.id`

`id :: a -> a`

The identity function For when you need a function that does “nothing”.

x

The value to return

Located at lib/trivial.nix:12 in `<nixpkgs>`.

5.1.4.2. `lib.trivial.const`

`const :: a -> b -> a`

The constant function

Ignores the second argument. If called with only one argument, constructs a function that always returns a static value.

x

Value to return

y

Value to ignore

Example 5.82. `lib.trivial.const` usage example

```
let f = const 5; in f 10
=> 5
```

Located at lib/trivial.nix:26 in `<nixpkgs>`.

5.1.4.3. `lib.trivial.pipe`

`pipe :: a -> [<functions>] -> <return type of last function>`

Pipes a value through a list of functions, left to right.

`val`

Function argument

`functions`

Function argument

Example 5.83. `lib.trivial.pipe` usage example

```
pipe 2 [
(x: x + 2)  # 2 + 2 = 4
```

```
(x: x * 2)  # 4 * 2 = 8
]
=> 8

# ideal to do text transformations
pipe [ "a/b" "a/c" ] [

# create the cp command
(map (file: ''cp "${src}/${file}" $out\n''))

# concatenate all commands into one string
lib.concatStrings

# make that string into a nix derivation
(pkgs.runCommand "copy-to-out" {})

]
=> <drv which copies all files to $out>

The output type of each function has to be the input type
of the next function, and the last function returns the
final value.
```

Located at lib/trivial.nix:61 in `<nixpkgs>`.

5.1.4.4. `lib.trivial.concat`

note please don't add a function like `compose = flip pipe`. This would confuse users, because the order of the functions in the list is not clear. With pipe, it's obvious that it goes first-to-last. With `compose`, not so much.

x

Function argument

y

Function argument

Located at lib/trivial.nix:80 in `<nixpkgs>`.

5.1.4.5. `lib.trivial.or`

boolean "or"

x

Function argument

y

Function argument

Located at lib/trivial.nix:83 in `<nixpkgs>`.

5.1.4.6. `lib.trivial.and`

boolean “and”

x

Function argument

y

Function argument

Located at lib/trivial.nix:86 in `<nixpkgs>`.

5.1.4.7. `lib.trivial.bitAnd`

bitwise “and”

Located at lib/trivial.nix:89 in `<nixpkgs>`.

5.1.4.8. `lib.trivial.bitOr`

bitwise “or”

Located at lib/trivial.nix:94 in `<nixpkgs>`.

5.1.4.9. `lib.trivial.bitXor`

bitwise “xor”

Located at lib/trivial.nix:99 in `<nixpkgs>`.

5.1.4.10. `lib.trivial.bitNot`

bitwise “not”

Located at lib/trivial.nix:104 in `<nixpkgs>`.

5.1.4.11. `lib.trivial.boolToString`

`boolToString :: bool -> string`

Convert a boolean to a string.

This function uses the strings "true" and "false" to represent boolean values. Calling `toString` on a bool instead returns "1" and "" (sic!).

b
: Function argument

Located at lib/trivial.nix:114 in `<nixpkgs>`.

5.1.4.12. `lib.trivial.mergeAttrs`

Merge two attribute sets shallowly, right side trumps left

mergeAttrs :: attrs -> attrs -> attrs

x
: Left attribute set

y
: Right attribute set (higher precedence for equal keys)

Example 5.84. `lib.trivial.mergeAttrs` usage example

```
mergeAttrs { a = 1; b = 2; } { b = 3; c = 4; }
=> { a = 1; b = 3; c = 4; }
```

Located at lib/trivial.nix:124 in `<nixpkgs>`.

5.1.4.13. `lib.trivial.flip`

`flip :: (a -> b -> c) -> (b -> a -> c)`

Flip the order of the arguments of a binary function.

f
: Function argument

a
: Function argument

b
: Function argument

Example 5.85. `lib.trivial.flip` usage example

```
flip concat [1] [2]
```

```
=> [ 2 1 ]
```

Located at lib/trivial.nix:138 in `<nixpkgs>`.

5.1.4.14. `lib.trivial.mapNullable`

Apply function if the supplied argument is non-null.

`f`

Function to call

`a`

Argument to check for null before passing it to `f`

Example 5.86. `lib.trivial.mapNullable` usage example

```
mapNullable (x: x+1) null
=> null
mapNullable (x: x+1) 22
=> 23
```

Located at lib/trivial.nix:148 in `<nixpkgs>`.

5.1.4.15. `lib.trivial.version`

Returns the current full nixpkgs version number.

Located at lib/trivial.nix:164 in `<nixpkgs>`.

5.1.4.16. `lib.trivial.release`

Returns the current nixpkgs release number as string.

Located at lib/trivial.nix:167 in `<nixpkgs>`.

5.1.4.17. `lib.trivial.codeName`

Returns the current nixpkgs release code name.

On each release the first letter is bumped and a new animal is chosen starting with that new letter.

Located at lib/trivial.nix:174 in `<nixpkgs>`.

5.1.4.18. `lib.trivial.versionSuffix`

Returns the current nixpkgs version suffix as string.

Located at lib/trivial.nix:177 in `<nixpkgs>`.

5.1.4.19. `lib.trivial.revisionWithDefault`

`revisionWithDefault :: string -> string`

Attempts to return the the current revision of nixpkgs and returns the supplied default value otherwise.

`default`

Default value to return if revision can not be determined

Located at lib/trivial.nix:188 in `<nixpkgs>`.

5.1.4.20. `lib.trivial.inNixShell`

`inNixShell :: bool`

Determine whether the function is being called from inside a Nix shell.

Located at lib/trivial.nix:206 in `<nixpkgs>`.

5.1.4.21. `lib.trivial.min`

Return minimum of two numbers.

x

Function argument

y

Function argument

Located at lib/trivial.nix:212 in `<nixpkgs>`.

5.1.4.22. `lib.trivial.max`

Return maximum of two numbers.

x

Function argument

y

Function argument

Located at lib/trivial.nix:215 in `<nixpkgs>`.

5.1.4.23. `lib.trivial.mod`

Integer modulus

`base`

Function argument

`int`

Function argument

Example 5.87. `lib.trivial.mod` usage example

```
mod 11 10
=> 1
mod 1 10
=> 1
```

Located at lib/trivial.nix:225 in `<nixpkgs>`.

5.1.4.24. `lib.trivial.compare`

C-style comparisons

a < b, compare a b => -1 a == b, compare a b => 0 a > b, compare a b => 1

`a`

Function argument

`b`

Function argument

Located at lib/trivial.nix:236 in `<nixpkgs>`.

5.1.4.25. `lib.trivial.splitByAndCompare`

`(a -> bool) -> (a -> a -> int) -> (a -> a -> int) -> (a -> a -> int)`

Split type into two subtypes by predicate `p`, take all elements of the first subtype to be less than all the elements of the second subtype, compare elements of a single subtype with `yes` and `no` respectively.

`p`

Predicate

`yes`

Comparison function if predicate holds for both values

`no`

Comparison function if predicate holds for neither value

`a`

First value to compare

`b`

Second value to compare

Example 5.88. `lib.trivial.splitByAndCompare` usage example

```
let cmp = splitByAndCompare (hasPrefix "foo") compare compare; in

cmp "a" "z" => -1
cmp "fooa" "fooz" => -1

cmp "f" "a" => 1
cmp "fooa" "a" => -1
# while
compare "fooa" "a" => 1
```

Located at lib/trivial.nix:261 in `<nixpkgs>`.

5.1.4.26. `lib.trivial.importJSON`

Reads a JSON file.

Type :: path -> any

`path`

Function argument

Located at lib/trivial.nix:281 in `<nixpkgs>`.

5.1.4.27. `lib.trivial.importTOML`

Reads a TOML file.

Type :: path -> any

`path`

Function argument

Located at lib/trivial.nix:288 in `<nixpkgs>`.

5.1.4.28. `lib.trivial.setFunctionArgs`

Add metadata about expected function arguments to a function. The metadata should match the format given by builtins.functionArgs, i.e. a set from expected argument to a bool representing whether that argument has a default or not. setFunctionArgs : (a → b) → Map String Bool → (a → b)

This function is necessary because you can't dynamically create a function of the { a, b ? foo, ... }: format, but some facilities like callPackage expect to be able to query expected arguments.

`f`

Function argument

`args`

Function argument

Located at lib/trivial.nix:325 in `<nixpkgs>`.

5.1.4.29. `lib.trivial.functionArgs`

Extract the expected function arguments from a function. This works both with nix-native { a, b ? foo, ... }: style functions and functions with args set with 'setFunctionArgs'. It has the same return type and semantics as builtins.functionArgs. setFunctionArgs : (a → b) → Map String Bool.

`f`

Function argument

Located at lib/trivial.nix:337 in `<nixpkgs>`.

5.1.4.30. `lib.trivial.isFunction`

Check whether something is a function or something annotated with function args.

`f`

Function argument

Located at lib/trivial.nix:345 in `<nixpkgs>`.

5.1.4.31. `lib.trivial.toHexString`

Convert the given positive integer to a string of its hexadecimal representation. For example:

toHexString 0 => "0"

toHexString 16 => "10"

toHexString 250 => "FA"

`i`

Function argument

Located at lib/trivial.nix:357 in `<nixpkgs>`.

5.1.4.32. `lib.trivial.toBaseDigits`

`toBaseDigits base i` converts the positive integer i to a list of its digits in the given base. For example:

toBaseDigits 10 123 => [1 2 3]

toBaseDigits 2 6 => [1 1 0]

toBaseDigits 16 250 => [15 10]

`base`
: Function argument

`i`
: Function argument

Located at lib/trivial.nix:383 in `<nixpkgs>`.

5.1.5. List manipulation functions

5.1.5.1. `lib.lists.singleton`

`singleton :: a -> [a]`

Create a list consisting of a single element. `singleton x` is sometimes more convenient with respect to indentation than `[x]` when x spans multiple lines.

`x`
: Function argument

Example 5.89. `lib.lists.singleton` usage example

```
singleton "foo"
=> [ "foo" ]
```

Located at lib/lists.nix:22 in `<nixpkgs>`.

5.1.5.2. `lib.lists.forEach`

`forEach :: [a] -> (a -> b) -> [b]`

Apply the function to each element in the list. Same as `map`, but arguments flipped.

`xs`

Function argument

`f`

Function argument

Example 5.90. `lib.lists.forEach` usage example

```
forEach [ 1 2 ] (x:
toString x
)
=> [ "1" "2" ]
```

Located at lib/lists.nix:35 in `<nixpkgs>`.

5.1.5.3. `lib.lists.foldr`

`foldr :: (a -> b -> b) -> b -> [a] -> b`

"right fold" a binary function `op` between successive elements of `list` with `nul' as the starting value, i.e., `foldr op nul [x_1 x_2 ... x_n] == op x_1 (op x_2 ... (op x_n nul))`.

`op`

Function argument

`nul`

Function argument

`list`

Function argument

Example 5.91. `lib.lists.foldr` usage example

```
concat = foldr (a: b: a + b) "z"
concat [ "a" "b" "c" ]
=> "abcz"
# different types
strange = foldr (int: str: toString (int + 1) + str) "a"
strange [ 1 2 3 4 ]
=> "2345a"
```

Located at lib/lists.nix:52 in `<nixpkgs>`.

5.1.5.4. `lib.lists.fold`

`fold` is an alias of `foldr` for historic reasons

Located at lib/lists.nix:63 in `<nixpkgs>`.

5.1.5.5. `lib.lists.foldl`

`foldl :: (b -> a -> b) -> b -> [a] -> b`

"left fold", like `foldr`, but from the left: `foldl op nul [x_1 x_2 ... x_n] == op (... (op (op nul x_1) x_2) ... x_n)`.

`op`
: Function argument

`nul`
: Function argument

`list`
: Function argument

Example 5.92. `lib.lists.foldl` usage example

```
lconcat = foldl (a: b: a + b) "z"
lconcat [ "a" "b" "c" ]
=> "zabc"
# different types
lstrange = foldl (str: int: str + toString (int + 1)) "a"
lstrange [ 1 2 3 4 ]
=> "a2345"
```

Located at lib/lists.nix:80 in `<nixpkgs>`.

5.1.5.6. `lib.lists.foldl'`

`foldl' :: (b -> a -> b) -> b -> [a] -> b`

Strict version of `foldl`.

The difference is that evaluation is forced upon access. Usually used with small whole results (in contrast with lazily-generated list or large lists where only a part is consumed.)

Located at lib/lists.nix:96 in `<nixpkgs>`.

5.1.5.7. `lib.lists.imap0`

`imap0 :: (int -> a -> b) -> [a] -> [b]`

Map with index starting from 0

`f`

Function argument

`list`

Function argument

Example 5.93. `lib.lists.imap0` usage example

```
imap0 (i: v: "${v}-${toString i}") ["a" "b"]
=> [ "a-0" "b-1" ]
```

Located at lib/lists.nix:106 in `<nixpkgs>`.

5.1.5.8. `lib.lists.imap1`

`imap1 :: (int -> a -> b) -> [a] -> [b]`

Map with index starting from 1

`f`

Function argument

`list`

Function argument

Example 5.94. `lib.lists.imap1` usage example

```
imap1 (i: v: "${v}-${toString i}") ["a" "b"]
=> [ "a-1" "b-2" ]
```

Located at lib/lists.nix:116 in `<nixpkgs>`.

5.1.5.9. `lib.lists.concatMap`

`concatMap :: (a -> [b]) -> [a] -> [b]`

Map and concatenate the result.

Example 5.95. `lib.lists.concatMap` usage example

```
concatMap (x: [x] ++ ["z"]) ["a" "b"]
=> [ "a" "z" "b" "z" ]
```

Located at lib/lists.nix:126 in `<nixpkgs>`.

5.1.5.10. `lib.lists.flatten`

Flatten the argument into a single list; that is, nested lists are spliced into the top-level lists.

x

Function argument

Example 5.96. `lib.lists.flatten` usage example

```
flatten [1 [2 [3] 4] 5]
=> [1 2 3 4 5]
flatten 1
=> [1]
```

Located at lib/lists.nix:137 in `<nixpkgs>`.

5.1.5.11. `lib.lists.remove`

`remove :: a -> [a] -> [a]`

Remove elements equal to 'e' from a list. Useful for buildInputs.

e

Element to remove from the list

Example 5.97. `lib.lists.remove` usage example

```
remove 3 [ 1 3 4 3 ]
=> [ 1 4 ]
```

Located at lib/lists.nix:150 in `<nixpkgs>`.

5.1.5.12. `lib.lists.findSingle`

`findSingle :: (a -> bool) -> a -> a -> [a] -> a`

Find the sole element in the list matching the specified predicate, returns `default` if no such element exists, or `multiple` if there are multiple matching elements.

`pred`
: Predicate

`default`
: Default value to return if element was not found.

`multiple`
: Default value to return if more than one element was found

`list`
: Input list

Example 5.98. `lib.lists.findSingle` usage example

```
findSingle (x: x == 3) "none" "multiple" [ 1 3 3 ]
=> "multiple"
findSingle (x: x == 3) "none" "multiple" [ 1 3 ]
=> 3
findSingle (x: x == 3) "none" "multiple" [ 1 9 ]
=> "none"
```

Located at lib/lists.nix:168 in `<nixpkgs>`.

5.1.5.13. `lib.lists.findFirst`

`findFirst :: (a -> bool) -> a -> [a] -> a`

Find the first element in the list matching the specified predicate or return `default` if no such element exists.

`pred`
: Predicate

`default`
: Default value to return

`list`
: Input list

Example 5.99. `lib.lists.findFirst` usage example

```
findFirst (x: x > 3) 7 [ 1 6 4 ]
=> 6
findFirst (x: x > 9) 7 [ 1 6 4 ]
=> 7
```

Located at lib/lists.nix:193 in `<nixpkgs>`.

5.1.5.14. `lib.lists.any`

`any :: (a -> bool) -> [a] -> bool`

Return true if function `pred` returns true for at least one element of `list`.

Example 5.100. `lib.lists.any` usage example

```
any isString [ 1 "a" { } ]
=> true
any isString [ 1 { } ]
=> false
```

Located at lib/lists.nix:214 in `<nixpkgs>`.

5.1.5.15. `lib.lists.all`

`all :: (a -> bool) -> [a] -> bool`

Return true if function `pred` returns true for all elements of `list`.

Example 5.101. `lib.lists.all` usage example

```
all (x: x < 3) [ 1 2 ]
=> true
all (x: x < 3) [ 1 2 3 ]
=> false
```

Located at lib/lists.nix:227 in `<nixpkgs>`.

5.1.5.16. `lib.lists.count`

`count :: (a -> bool) -> [a] -> int`

Count how many elements of `list` match the supplied predicate function.

`pred`
: Predicate

Example 5.102. `lib.lists.count` usage example

```
count (x: x == 3) [ 3 2 3 4 6 ]
=> 2
```

Located at lib/lists.nix:238 in `<nixpkgs>`.

5.1.5.17. `lib.lists.optional`

`optional :: bool -> a -> [a]`

Return a singleton list or an empty list, depending on a boolean value. Useful when building lists with optional elements (e.g. `++ optional (system == "i686-linux") firefox').

`cond`
: Function argument

`elem`
: Function argument

Example 5.103. `lib.lists.optional` usage example

```
optional true "foo"
=> [ "foo" ]
optional false "foo"
=> [ ]
```

Located at lib/lists.nix:254 in `<nixpkgs>`.

5.1.5.18. `lib.lists.optionals`

`optionals :: bool -> [a] -> [a]`

Return a list or an empty list, depending on a boolean value.

`cond`

Condition

`elems`

List to return if condition is true

Example 5.104. `lib.lists.optionals` usage example

```
optionals true [ 2 3 ]
=> [ 2 3 ]
optionals false [ 2 3 ]
=> [ ]
```

Located at lib/lists.nix:266 in `<nixpkgs>`.

5.1.5.19. `lib.lists.toList`

If argument is a list, return it; else, wrap it in a singleton list. If you're using this, you should almost certainly reconsider if there isn't a more "well-typed" approach.

`x`

Function argument

Example 5.105. `lib.lists.toList` usage example

```
toList [ 1 2 ]
=> [ 1 2 ]
toList "hi"
=> [ "hi "]
```

Located at lib/lists.nix:283 in `<nixpkgs>`.

5.1.5.20. `lib.lists.range`

`range :: int -> int -> [int]`

Return a list of integers from `first' up to and including `last'.

`first`

First integer in the range

`last`

Last integer in the range

Example 5.106. `lib.lists.range` usage example

```
range 2 4
=> [ 2 3 4 ]
range 3 2
=> [ ]
```

Located at lib/lists.nix:295 in `<nixpkgs>`.

5.1.5.21. `lib.lists.partition`

`(a -> bool) -> [a] -> { right :: [a], wrong :: [a] }`

Splits the elements of a list in two lists, `right` and `wrong`, depending on the evaluation of a predicate.

Example 5.107. `lib.lists.partition` usage example

```
partition (x: x > 2) [ 5 1 2 3 4 ]
=> { right = [ 5 3 4 ]; wrong = [ 1 2 ]; }
```

Located at lib/lists.nix:314 in `<nixpkgs>`.

5.1.5.22. `lib.lists.groupBy'`

Splits the elements of a list into many lists, using the return value of a predicate. Predicate should return a string which becomes keys of attrset `groupBy' returns.

`groupBy'` allows to customise the combining function and initial value

`op`

Function argument

`nul`

Function argument

`pred`

Function argument

`lst`

Function argument

Example 5.108. `lib.lists.groupBy'` usage example

```
groupBy (x: boolToString (x > 2)) [ 5 1 2 3 4 ]
=> { true = [ 5 3 4 ]; false = [ 1 2 ]; }
groupBy (x: x.name) [ {name = "icewm"; script = "icewm &";}
{name = "xfce";  script = "xfce4-session &";}
{name = "icewm"; script = "icewmbg &";}
{name = "mate";  script = "gnome-session &";}
]
=> { icewm = [ { name = "icewm"; script = "icewm &"; }
{ name = "icewm"; script = "icewmbg &"; } ];
mate  = [ { name = "mate";  script = "gnome-session &"; } ];
xfce  = [ { name = "xfce";  script = "xfce4-session &"; } ];
}

groupBy' builtins.add 0 (x: boolToString (x > 2)) [ 5 1 2 3 4 ]
=> { true = 12; false = 3; }
```

Located at lib/lists.nix:343 in `<nixpkgs>`.

5.1.5.23. `lib.lists.zipListsWith`

`zipListsWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

Merges two lists of the same size together. If the sizes aren't the same the merging stops at the shortest. How both lists are merged is defined by the first argument.

`f`
: Function to zip elements of both lists

`fst`
: First list

`snd`
: Second list

Example 5.109. `lib.lists.zipListsWith` usage example

```
zipListsWith (a: b: a + b) ["h" "l"] ["e" "o"]
=> ["he" "lo"]
```

Located at lib/lists.nix:363 in `<nixpkgs>`.

5.1.5.24. `lib.lists.zipLists`

`zipLists :: [a] -> [b] -> [{ fst :: a, snd :: b}]`

Merges two lists of the same size together. If the sizes aren't the same the merging stops at the shortest.

Example 5.110. `lib.lists.zipLists` usage example

```
zipLists [ 1 2 ] [ "a" "b" ]
=> [ { fst = 1; snd = "a"; } { fst = 2; snd = "b"; } ]
```

Located at lib/lists.nix:382 in `<nixpkgs>`.

5.1.5.25. `lib.lists.reverseList`

`reverseList :: [a] -> [a]`

Reverse the order of the elements of a list.

`xs`

Function argument

Example 5.111. `lib.lists.reverseList` usage example

```
reverseList [ "b" "o" "j" ]
=> [ "j" "o" "b" ]
```

Located at lib/lists.nix:393 in `<nixpkgs>`.

5.1.5.26. `lib.lists.listDfs`

Depth-First Search (DFS) for lists `list != []`.

`before a b == true` means that `b` depends on `a` (there's an edge from `b` to `a`).

`stopOnCycles`

Function argument

`before`

Function argument

`list`

Function argument

Example 5.112. `lib.lists.listDfs` usage example

```
listDfs true hasPrefix [ "/home/user" "other" "/" "/home" ]
== { minimal = "/";                  # minimal element
visited = [ "/home/user" ];     # seen elements (in reverse order)
rest    = [ "/home" "other" ];  # everything else
}

listDfs true hasPrefix [ "/home/user" "other" "/" "/home" "/" ]
== { cycle   = "/";                  # cycle encountered at this element
loops   = [ "/" ];              # and continues to these elements
visited = [ "/" "/home/user" ]; # elements leading to the cycle (in reverse
rest    = [ "/home" "other" ];  # everything else
```

Located at lib/lists.nix:415 in `<nixpkgs>`.

5.1.5.27. `lib.lists.toposort`

Sort a list based on a partial ordering using DFS. This implementation is O(N^2), if your ordering is linear, use `sort` instead.

`before a b == true` means that `b` should be after `a` in the result.

`before`

Function argument

`list`

Function argument

Example 5.113. `lib.lists.toposort` usage example

```
toposort hasPrefix [ "/home/user" "other" "/" "/home" ]
== { result = [ "/" "/home" "/home/user" "other" ]; }

toposort hasPrefix [ "/home/user" "other" "/" "/home" "/" ]
== { cycle = [ "/home/user" "/" "/" ]; # path leading to a cycle
loops = [ "/" ]; }                # loops back to these elements

toposort hasPrefix [ "other" "/home/user" "/home" "/" ]
```

```
== { result = [ "other" "/" "/home" "/home/user" ]; }

toposort (a: b: a < b) [ 3 2 1 ] == { result = [ 1 2 3 ]; }
```

Located at lib/lists.nix:454 in `<nixpkgs>`.

5.1.5.28. `lib.lists.sort`

Sort a list based on a comparator function which compares two elements and returns true if the first argument is strictly below the second argument. The returned list is sorted in an increasing order. The implementation does a quick-sort.

Example 5.114. `lib.lists.sort` usage example

```
sort (a: b: a < b) [ 5 3 7 ]
=> [ 3 5 7 ]
```

Located at lib/lists.nix:482 in `<nixpkgs>`.

5.1.5.29. `lib.lists.compareLists`

Compare two lists element-by-element.

`cmp`

Function argument

`a`

Function argument

`b`

Function argument

Example 5.115. `lib.lists.compareLists` usage example

```
compareLists compare [] []
=> 0
compareLists compare [] [ "a" ]
=> -1
compareLists compare [ "a" ] []
=> 1
compareLists compare [ "a" "b" ] [ "a" "c" ]
=> 1
```

Located at lib/lists.nix:511 in `<nixpkgs>`.

5.1.5.30. `lib.lists.naturalSort`

Sort list using "Natural sorting". Numeric portions of strings are sorted in numeric order.

`lst`

Function argument

Example 5.116. `lib.lists.naturalSort` usage example

```
naturalSort ["disk11" "disk8" "disk100" "disk9"]
=> ["disk8" "disk9" "disk11" "disk100"]
naturalSort ["10.46.133.149" "10.5.16.62" "10.54.16.25"]
=> ["10.5.16.62" "10.46.133.149" "10.54.16.25"]
naturalSort ["v0.2" "v0.15" "v0.0.9"]
=> [ "v0.0.9" "v0.2" "v0.15" ]
```

Located at lib/lists.nix:534 in `<nixpkgs>`.

5.1.5.31. `lib.lists.take`

`take :: int -> [a] -> [a]`

Return the first (at most) N elements of a list.

`count`

Number of elements to take

Example 5.117. `lib.lists.take` usage example

```
take 2 [ "a" "b" "c" "d" ]
=> [ "a" "b" ]
take 2 [ ]
=> [ ]
```

Located at lib/lists.nix:552 in `<nixpkgs>`.

5.1.5.32. `lib.lists.drop`

`drop :: int -> [a] -> [a]`

Remove the first (at most) N elements of a list.

`count`

Number of elements to drop

`list`

Input list

Example 5.118. `lib.lists.drop` usage example

```
drop 2 [ "a" "b" "c" "d" ]
=> [ "c" "d" ]
drop 2 [ ]
=> [ ]
```

Located at lib/lists.nix:566 in `<nixpkgs>`.

5.1.5.33. `lib.lists.sublist`

`sublist :: int -> int -> [a] -> [a]`

Return a list consisting of at most `count` elements of `list`, starting at index `start`.

`start`

Index at which to start the sublist

`count`

Number of elements to take

`list`

Input list

Example 5.119. `lib.lists.sublist` usage example

```
sublist 1 3 [ "a" "b" "c" "d" "e" ]
=> [ "b" "c" "d" ]
sublist 1 3 [ ]
=> [ ]
```

Located at lib/lists.nix:583 in `<nixpkgs>`.

5.1.5.34. `lib.lists.last`

`last :: [a] -> a`

Return the last element of a list.

This function throws an error if the list is empty.

`list`
: Function argument

Example 5.120. `lib.lists.last` usage example

```
last [ 1 2 3 ]
=> 3
```

Located at lib/lists.nix:607 in `<nixpkgs>`.

5.1.5.35. `lib.lists.init`

`init :: [a] -> [a]`

Return all elements but the last.

This function throws an error if the list is empty.

`list`
: Function argument

Example 5.121. `lib.lists.init` usage example

```
init [ 1 2 3 ]
=> [ 1 2 ]
```

Located at lib/lists.nix:621 in `<nixpkgs>`.

5.1.5.36. `lib.lists.crossLists`

Return the image of the cross product of some lists by a function.

Example 5.122. `lib.lists.crossLists` usage example

```
crossLists (x:y: "${toString x}${toString y}") [[1 2] [3 4]]
```

```
=> [ "13" "14" "23" "24" ]
```

Located at lib/lists.nix:632 in `<nixpkgs>`.

5.1.5.37. `lib.lists.unique`

`unique :: [a] -> [a]`

Remove duplicate elements from the list. O(n^2) complexity.

Example 5.123. `lib.lists.unique` usage example

```
unique [ 3 2 3 4 ]
=> [ 3 2 4 ]
```

Located at lib/lists.nix:645 in `<nixpkgs>`.

5.1.5.38. `lib.lists.intersectLists`

Intersects list 'e' and another list. O(nm) complexity.

e

Function argument

Example 5.124. `lib.lists.intersectLists` usage example

```
intersectLists [ 1 2 3 ] [ 6 3 2 ]
=> [ 3 2 ]
```

Located at lib/lists.nix:653 in `<nixpkgs>`.

5.1.5.39. `lib.lists.subtractLists`

Subtracts list 'e' from another list. O(nm) complexity.

e

Function argument

Example 5.125. `lib.lists.subtractLists` usage example

```
subtractLists [ 3 2 ] [ 1 2 3 4 5 3 ]
```

```
=> [ 1 4 5 ]
```

Located at lib/lists.nix:661 in `<nixpkgs>`.

5.1.5.40. `lib.lists.mutuallyExclusive`

Test if two lists have no common element. It should be slightly more efficient than (intersectLists a b == [])

`a`

Function argument

`b`

Function argument

Located at lib/lists.nix:666 in `<nixpkgs>`.

5.1.6. Debugging functions

5.1.6.1. `lib.debug.traceIf`

`traceIf :: bool -> string -> a -> a`

Conditionally trace the supplied message, based on a predicate.

`pred`

Predicate to check

`msg`

Message that should be traced

`x`

Value to return

Example 5.126. `lib.debug.traceIf` usage example

```
traceIf true "hello" 3
trace: hello
=> 3
```

Located at lib/debug.nix:51 in `<nixpkgs>`.

5.1.6.2. `lib.debug.traceValFn`

`traceValFn :: (a -> b) -> a -> a`

Trace the supplied value after applying a function to it, and return the original value.

f
: Function to apply

x
: Value to trace and return

Example 5.127. `lib.debug.traceValFn` usage example

```
traceValFn (v: "mystring ${v}") "foo"
trace: mystring foo
=> "foo"
```

Located at lib/debug.nix:69 in `<nixpkgs>`.

5.1.6.3. `lib.debug.traceVal`

`traceVal :: a -> a`

Trace the supplied value and return it.

Example 5.128. `lib.debug.traceVal` usage example

```
traceVal 42
# trace: 42
=> 42
```

Located at lib/debug.nix:84 in `<nixpkgs>`.

5.1.6.4. `lib.debug.traceSeq`

`traceSeq :: a -> b -> b`

`builtins.trace`, but the value is `builtins.deepSeq`ed first.

x
: The value to trace

y
: The value to return

Example 5.129. `lib.debug.traceSeq` usage example

```
trace { a.b.c = 3; } null
trace: { a = <CODE>; }
=> null
traceSeq { a.b.c = 3; } null
trace: { a = { b = { c = 3; }; }; }
=> null
```

Located at lib/debug.nix:98 in `<nixpkgs>`.

5.1.6.5. `lib.debug.traceSeqN`

Like `traceSeq`, but only evaluate down to depth n. This is very useful because lots of `traceSeq` usages lead to an infinite recursion.

`depth`
: Function argument

`x`
: Function argument

`y`
: Function argument

Example 5.130. `lib.debug.traceSeqN` usage example

```
traceSeqN 2 { a.b.c = 3; } null
trace: { a = { b = {…}; }; }
=> null
```

Located at lib/debug.nix:113 in `<nixpkgs>`.

5.1.6.6. `lib.debug.traceValSeqFn`

A combination of `traceVal` and `traceSeq` that applies a provided function to the value to be traced after `deepSeq`ing it.

`f`
: Function to apply

`v`

Value to trace

Located at lib/debug.nix:130 in `<nixpkgs>`.

5.1.6.7. `lib.debug.traceValSeq`

A combination of `traceVal` and `traceSeq`.

Located at lib/debug.nix:137 in `<nixpkgs>`.

5.1.6.8. `lib.debug.traceValSeqNFn`

A combination of `traceVal` and `traceSeqN` that applies a provided function to the value to be traced.

`f`

Function to apply

`depth`

Function argument

`v`

Value to trace

Located at lib/debug.nix:141 in `<nixpkgs>`.

5.1.6.9. `lib.debug.traceValSeqN`

A combination of `traceVal` and `traceSeqN`.

Located at lib/debug.nix:149 in `<nixpkgs>`.

5.1.6.10. `lib.debug.traceFnSeqN`

Trace the input and output of a function `f` named `name`, both down to `depth`.

This is useful for adding around a function call, to see the before/after of values as they are transformed.

`depth`

Function argument

`name`

Function argument

`f`

Function argument

`v`

Function argument

Example 5.131. `lib.debug.traceFnSeqN` usage example

```
traceFnSeqN 2 "id" (x: x) { a.b.c = 3; }
trace: { fn = "id"; from = { a.b = {…}; }; to = { a.b = {…}; }; }
=> { a.b.c = 3; }
```

Located at lib/debug.nix:162 in `<nixpkgs>`.

5.1.6.11. `lib.debug.runTests`

Evaluate a set of tests. A test is an attribute set `{expr, expected}`, denoting an expression and its expected result. The result is a list of failed tests, each represented as `{name, expected, actual}`, denoting the attribute name of the failing test and its expected and actual results.

Used for regression testing of the functions in lib; see tests.nix for an example. Only tests having names starting with "test" are run.

Add attr { tests = ["testName"]; } to run these tests only.

`tests`
: Tests to run

Located at lib/debug.nix:188 in `<nixpkgs>`.

5.1.6.12. `lib.debug.testAllTrue`

Create a test assuming that list elements are `true`.

`expr`
: Function argument

Example 5.132. `lib.debug.testAllTrue` usage example

```
{ testX = allTrue [ true ]; }
```

Located at lib/debug.nix:204 in `<nixpkgs>`.

5.1.7. NixOS / nixpkgs option handling

5.1.7.1. `lib.options.isOption`

`isOption :: a -> bool`

Returns true when the given argument is an option

Example 5.133. `lib.options.isOption` usage example

```
isOption 1             // => false
isOption (mkOption {}) // => true
```

Located at lib/options.nix:48 in `<nixpkgs>`.

5.1.7.2. `lib.options.mkOption`

Creates an Option attribute set. mkOption accepts an attribute set with the following keys:

All keys default to `null` when not given.

`pattern`

Structured function argument

`default`

Default value used when no definition is given in the configuration.

`defaultText`

Textual representation of the default, for the manual.

`example`

Example value used in the manual.

`description`

String describing the option.

`relatedPackages`

Related packages used in the manual (see `genRelatedPackages` in ../nixos/lib/make-options-doc/default.nix).

`type`

Option type, providing type-checking and value merging.

`apply`

Function that converts the option value to something else.

`internal`

Whether the option is for NixOS developers only.

`visible`

Whether the option shows up in the manual.

`readOnly`

Whether the option can be set only once

`options`

Deprecated, used by types.optionSet.

Example 5.134. `lib.options.mkOption` usage example

```
mkOption { }  // => { _type = "option"; }
mkOption { defaultText = "foo"; } // => { _type = "option"; defaultText = "
```

Located at lib/options.nix:58 in `<nixpkgs>`.

5.1.7.3. `lib.options.mkEnableOption`

Creates an Option attribute set for a boolean value option i.e an option to be toggled on or off:

`name`

Name for the created option

Example 5.135. `lib.options.mkEnableOption` usage example

```
mkEnableOption "foo"
=> { _type = "option"; default = false; description = "Whether to enable fo
```

Located at lib/options.nix:92 in `<nixpkgs>`.

5.1.7.4. `lib.options.mkSinkUndeclaredOptions`

This option accepts anything, but it does not produce any result.

This is useful for sharing a module across different module sets without having to implement similar features as long as the values of the options are not accessed.

`attrs`

Function argument

Located at lib/options.nix:106 in `<nixpkgs>`.

5.1.7.5. `lib.options.mergeEqualOption`

"Merge" option definitions by checking that they all have the same value.

`loc`

Function argument

`defs`

Function argument

Located at lib/options.nix:137 in `<nixpkgs>`.

5.1.7.6. `lib.options.getValues`

`getValues :: [ { value :: a } ] -> [a]`

Extracts values of all "value" keys of the given list.

Example 5.136. `lib.options.getValues` usage example

```
getValues [ { value = 1; } { value = 2; } ] // => [ 1 2 ]
getValues [ ]                               // => [ ]
```

Located at lib/options.nix:157 in `<nixpkgs>`.

5.1.7.7. `lib.options.getFiles`

`getFiles :: [ { file :: a } ] -> [a]`

Extracts values of all "file" keys of the given list

Example 5.137. `lib.options.getFiles` usage example

```
getFiles [ { file = "file1"; } { file = "file2"; } ] // => [ "file1" "file2
getFiles [ ]                                         // => [ ]
```

Located at lib/options.nix:167 in `<nixpkgs>`.

5.1.7.8. `lib.options.scrubOptionValue`

This function recursively removes all derivation attributes from `x` except for the `name` attribute.

This is to make the generation of `options.xml` much more efficient: the XML representation of derivations is very large (on the order of megabytes) and is not actually used by the manual generator.

`x`

Function argument

Located at lib/options.nix:206 in `<nixpkgs>`.

5.1.7.9. `lib.options.literalExample`

For use in the `example` option attribute. It causes the given text to be included verbatim in documentation. This is necessary for example values that are not simple values, e.g., functions.

`text`

Function argument

Located at lib/options.nix:218 in `<nixpkgs>`.

5.1.7.10. `lib.options.showOption`

Convert an option, described as a list of the option parts in to a safe, human readable version.

`parts`

Function argument

Example 5.138. `lib.options.showOption` usage example

```
(showOption ["foo" "bar" "baz"]) == "foo.bar.baz"
(showOption ["foo" "bar.baz" "tux"]) == "foo.bar.baz.tux"

Placeholders will not be quoted as they are not actual values:
(showOption ["foo" "*" "bar"]) == "foo.*.bar"
(showOption ["foo" "<name>" "bar"]) == "foo.<name>.bar"

Unlike attributes, options can also start with numbers:
(showOption ["windowManager" "2bwm" "enable"]) == "windowManager.2bwm.enabl
```

Located at lib/options.nix:236 in `<nixpkgs>`.

5.2. Generators

Generators are functions that create file formats from nix data structures, e. g. for configuration files. There are generators available for: `INI`, `JSON` and `YAML`

All generators follow a similar call interface: `generatorName configFunctions data`, where `configFunctions` is an attrset of user-defined functions that format nested parts of the content. They each have common defaults, so often they do not need to be set manually. An example is `mkSectionName ? (name: libStr.escape [ "[" "]" ] name)` from the `INI` generator. It receives the name of a section and sanitizes it. The default `mkSectionName` escapes `[` and `]` with a backslash.

Generators can be fine-tuned to produce exactly the file format required by your application/service. One example is an INI-file format which uses `:` as separator, the strings `"yes"`/`"no"` as boolean values and requires all string values to be quoted:

```
with lib;
let
  customToINI = generators.toINI {
    # specifies how to format a key/value pair
    mkKeyValue = generators.mkKeyValueDefault {
      # specifies the generated string for a subset of nix values
      mkValueString = v:
            if v == true then '''"yes"'''
        else if v == false then '''"no"'''
        else if isString v then '''"${v}"'''
        # and delegats all other values to the default generator
        else generators.mkValueStringDefault {} v;
    } ":";
  };

# the INI file can now be given as plain old nix values
in customToINI {
  main = {
    pushinfo = true;
    autopush = false;
    host = "localhost";
    port = 42;
  };
  mergetool = {
    merge = "diff3";
  };
}
```

This will produce the following INI file as nix string:

```
[main]
autopush:"no"
host:"localhost"
port:42
pushinfo:"yes"
str\:ange:"very::strange"

[mergetool]
merge:"diff3"
```

Note: Nix store paths can be converted to strings by enclosing a derivation attribute like so: `"${drv}"`.

Detailed documentation for each generator can be found in `lib/generators.nix`.

5.3. Debugging Nix Expressions

Nix is a unityped, dynamic language, this means every value can potentially appear anywhere. Since it is also non-strict, evaluation order and what ultimately is evaluated might surprise you. Therefore it is important to be able to debug nix expressions.

In the `lib/debug.nix` file you will find a number of functions that help (pretty-)printing values while evaluation is runnnig. You can even specify how deep these values should be printed recursively, and transform them on the fly. Please consult the docstrings in `lib/debug.nix` for usage information.

5.4. prefer-remote-fetch overlay

`prefer-remote-fetch` is an overlay that download sources on remote builder. This is useful when the evaluating machine has a slow upload while the builder can fetch faster directly from the source. To use it, put the following snippet as a new overlay:

```
self: super:
  (super.prefer-remote-fetch self super)
```

A full configuration example for that sets the overlay up for your own account, could look like this

```
$ mkdir ~/.config/nixpkgs/overlays/
$ cat > ~/.config/nixpkgs/overlays/prefer-remote-fetch.nix <<EOF
  self: super: super.prefer-remote-fetch self super
EOF
```

5.5. pkgs.nix-gitignore

`pkgs.nix-gitignore` is a function that acts similarly to `builtins.filterSource` but also allows filtering with the help of the gitignore format.

5.5.1. Usage

`pkgs.nix-gitignore` exports a number of functions, but you'll most likely need either `gitignoreSource` or `gitignoreSourcePure`. As their first argument, they both accept either 1. a file with gitignore lines or 2. a string with gitignore lines, or 3. a list of either of the two. They will be

concatenated into a single big string.

```
{ pkgs ? import <nixpkgs> {} }:

 nix-gitignore.gitignoreSource [] ./source
     # Simplest version

 nix-gitignore.gitignoreSource "supplemental-ignores\n" ./source
     # This one reads the ./source/.gitignore and concats the auxiliary ign

 nix-gitignore.gitignoreSourcePure "ignore-this\nignore-that\n" ./source
     # Use this string as gitignore, don't read ./source/.gitignore.

 nix-gitignore.gitignoreSourcePure ["ignore-this\nignore-that\n", ~/.gitign
     # It also accepts a list (of strings and paths) that will be concatena
     # once the paths are turned to strings via readFile.
```

These functions are derived from the `Filter` functions by setting the first filter argument to `(_: _: true)`:

```
gitignoreSourcePure = gitignoreFilterSourcePure (_: _: true);
gitignoreSource = gitignoreFilterSource (_: _: true);
```

Those filter functions accept the same arguments the `builtins.filterSource` function would pass to its filters, thus `fn: gitignoreFilterSourcePure fn ""` should be extensionally equivalent to `filterSource`. The file is blacklisted iff it's blacklisted by either your filter or the gitignoreFilter.

If you want to make your own filter from scratch, you may use

```
gitignoreFilter = ign: root: filterPattern (gitignoreToPatterns ign) root;
```

5.5.2. gitignore files in subdirectories

If you wish to use a filter that would search for .gitignore files in subdirectories, just like git does by default, use this function:

```
gitignoreFilterRecursiveSource = filter: patterns: root:
# OR
gitignoreRecursiveSource = gitignoreFilterSourcePure (_: _: true);
```

Part II. Standard environment

Table of Contents

6. The Standard Environment

7. Meta-attributes

8. Multiple-output packages

9. Cross-compilation

10. Platform Notes

Chapter 6. The Standard Environment

Table of Contents

6.1. Using `stdenv`

6.2. Tools provided by `stdenv`

6.3. Specifying dependencies

6.4. Attributes

6.5. Phases

6.6. Shell functions

6.7. Package setup hooks

6.8. Purity in Nixpkgs

6.9. Hardening in Nixpkgs

The standard build environment in the Nix Packages collection provides an environment for building Unix packages that does a lot of common build tasks automatically. In fact, for Unix packages that use the standard `./configure; make; make install` build interface, you don't need to write a build script at all; the standard environment does everything automatically. If `stdenv` doesn't do what you need automatically, you can easily customise or override the various build phases.

6.1. Using `stdenv`

To build a package with the standard environment, you use the function `stdenv.mkDerivation`, instead of the primitive built-in function `derivation`, e.g.

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  src = fetchurl {
    url = "http://example.org/libfoo-1.2.3.tar.bz2";
    sha256 = "0x2g1jqygyr5wiwg4ma1nd7w4ydpy82z9gkcv8vh2v8dn3y58v5m";
  };
}
```

(`stdenv` needs to be in scope, so if you write this in a separate Nix expression from `pkgs/all-packages.nix`, you need to pass it as a function argument.) Specifying a `name` and a `src` is the absolute minimum Nix requires. For convenience, you can also use `pname` and `version` attributes and `mkDerivation` will automatically set `name` to `"${pname}-${version}"` by default. Since RFC 0035, this is preferred for packages in Nixpkgs, as it allows us to reuse the version easily:

```
stdenv.mkDerivation rec {
  pname = "libfoo";
  version = "1.2.3";
  src = fetchurl {
    url = "http://example.org/libfoo-source-${version}.tar.bz2";
    sha256 = "0x2g1jqygyr5wiwg4ma1nd7w4ydpy82z9gkcv8vh2v8dn3y58v5m";
  };
}
```

Many packages have dependencies that are not provided in the standard environment. It's usually sufficient to specify those dependencies in the `buildInputs` attribute:

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  ...
  buildInputs = [libbar perl ncurses];
}
```

This attribute ensures that the `bin` subdirectories of these packages appear in the `PATH` environment variable during the build, that their `include` subdirectories are searched by the C compiler, and so on. (See Section 6.7, "Package setup hooks" for details.)

Often it is necessary to override or modify some aspect of the build. To make this easier, the standard environment breaks the package build into a number of *phases*, all of which can be overridden or modified individually: unpacking the sources, applying patches, configuring, building, and installing. (There are

some others; see Section 6.5, "Phases".) For instance, a package that doesn't supply a makefile but instead has to be compiled "manually" could be handled like this:

```
stdenv.mkDerivation {
  name = "fnord-4.5";
  ...
  buildPhase = ''
    gcc foo.c -o foo
  '';
  installPhase = ''
    mkdir -p $out/bin
    cp foo $out/bin
  '';
}
```

(Note the use of `''`-style string literals, which are very convenient for large multi-line script fragments because they don't need escaping of `"` and \, and because indentation is intelligently removed.)

There are many other attributes to customise the build. These are listed in Section 6.4, "Attributes".

While the standard environment provides a generic builder, you can still supply your own build script:

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  ...
  builder = ./builder.sh;
}
```

where the builder can do anything it wants, but typically starts with

```
source $stdenv/setup
```

to let `stdenv` set up the environment (e.g., process the `buildInputs`). If you want, you can still use `stdenv`'s generic builder:

```
source $stdenv/setup

buildPhase() {
  echo "... this is my custom build phase ..."
  gcc foo.c -o foo
}

installPhase() {
```

```
  mkdir -p $out/bin
  cp foo $out/bin
}

genericBuild
```

6.2. Tools provided by stdenv

The standard environment provides the following packages:

- The GNU C Compiler, configured with C and C++ support.
- GNU coreutils (contains a few dozen standard Unix commands).
- GNU findutils (contains `find`).
- GNU diffutils (contains `diff`, `cmp`).
- GNU `sed`.
- GNU `grep`.
- GNU `awk`.
- GNU `tar`.
- `gzip`, `bzip2` and `xz`.
- GNU Make.
- Bash. This is the shell used for all builders in the Nix Packages collection. Not using `/bin/sh` removes a large source of portability problems.
- The `patch` command.

On Linux, `stdenv` also includes the `patchelf` utility.

6.3. Specifying dependencies

As described in the Nix manual, almost any `*.drv` store path in a derivation's attribute set will induce a dependency on that derivation. `mkDerivation`, however, takes a few attributes intended to, between them, include all the dependencies of a package. This is done both for structure and consistency, but also so that certain other setup can take place. For example, certain dependencies need their bin directories added to the `PATH`. That is built-in, but other setup is done via a pluggable mechanism that works in conjunction with these dependency attributes. See Section 6.7, "Package setup hooks" for details.

Dependencies can be broken down along three axes: their host and target platforms relative to the new derivation's, and whether they are propagated. The platform distinctions are motivated by cross compilation; see Chapter 9, *Cross-compilation* for exactly what each platform means. [1] But even if one is

not cross compiling, the platforms imply whether or not the dependency is needed at run-time or build-time, a concept that makes perfect sense outside of cross compilation. By default, the run-time/build-time distinction is just a hint for mental clarity, but with `strictDeps` set it is mostly enforced even in the native case.

The extension of `PATH` with dependencies, alluded to above, proceeds according to the relative platforms alone. The process is carried out only for dependencies whose host platform matches the new derivation's build platform i.e. dependencies which run on the platform where the new derivation will be built. [2] For each dependency <dep> of those dependencies, `dep/bin`, if present, is added to the `PATH` environment variable.

The dependency is propagated when it forces some of its other-transitive (non-immediate) downstream dependencies to also take it on as an immediate dependency. Nix itself already takes a package's transitive dependencies into account, but this propagation ensures nixpkgs-specific infrastructure like setup hooks (mentioned above) also are run as if the propagated dependency.

It is important to note that dependencies are not necessarily propagated as the same sort of dependency that they were before, but rather as the corresponding sort so that the platform rules still line up. The exact rules for dependency propagation can be given by assigning to each dependency two integers based one how its host and target platforms are offset from the depending derivation's platforms. Those offsets are given below in the descriptions of each dependency list attribute. Algorithmically, we traverse propagated inputs, accumulating every propagated dependency's propagated dependencies and adjusting them to account for the "shift in perspective" described by the current dependency's platform offsets. This results in sort a transitive closure of the dependency relation, with the offsets being approximately summed when two dependency links are combined. We also prune transitive dependencies whose combined offsets go out-of-bounds, which can be viewed as a filter over that transitive closure removing dependencies that are blatantly absurd.

We can define the process precisely with Natural Deduction using the inference rules. This probably seems a bit obtuse, but so is the bash code that actually implements it! [3] They're confusing in very different ways so… hopefully if something doesn't make sense in one presentation, it will in the other!

```
let mapOffset(h, t, i) = i + (if i <= 0 then h else t - 1)

propagated-dep(h0, t0, A, B)
propagated-dep(h1, t1, B, C)
h0 + h1 in {-1, 0, 1}
h0 + t1 in {-1, 0, 1}
------------------------------------- Transitive property
propagated-dep(mapOffset(h0, t0, h1),
               mapOffset(h0, t0, t1),
               A, C)
```

```
let mapOffset(h, t, i) = i + (if i <= 0 then h else t - 1)
```

```
dep(h0, _, A, B)
propagated-dep(h1, t1, B, C)
h0 + h1 in {-1, 0, 1}
h0 + t1 in {-1, 0, -1}
---------------------------- Take immediate dependencies' propagated depen
propagated-dep(mapOffset(h0, t0, h1),
               mapOffset(h0, t0, t1),
               A, C)
```

```
propagated-dep(h, t, A, B)
---------------------------- Propagated dependencies count as dependencies
dep(h, t, A, B)
```

Some explanation of this monstrosity is in order. In the common case, the target offset of a dependency is the successor to the target offset: `t = h + 1`. That means that:

```
let f(h, t, i) = i + (if i <= 0 then h else t - 1)
let f(h, h + 1, i) = i + (if i <= 0 then h else (h + 1) - 1)
let f(h, h + 1, i) = i + (if i <= 0 then h else h)
let f(h, h + 1, i) = i + h
```

This is where “sum-like” comes in from above: We can just sum all of the host offsets to get the host offset of the transitive dependency. The target offset is the transitive dependency is simply the host offset + 1, just as it was with the dependencies composed to make this transitive one; it can be ignored as it doesn’t add any new information.

Because of the bounds checks, the uncommon cases are `h = t` and `h + 2 = t`. In the former case, the motivation for `mapOffset` is that since its host and target platforms are the same, no transitive dependency of it should be able to “discover” an offset greater than its reduced target offsets. `mapOffset` effectively “squashes” all its transitive dependencies’ offsets so that none will ever be greater than the target offset of the original `h = t` package. In the other case, `h + 1` is skipped over between the host and target offsets. Instead of squashing the offsets, we need to “rip” them apart so no transitive dependencies’ offset is that one.

Overall, the unifying theme here is that propagation shouldn’t be introducing transitive dependencies involving platforms the depending package is unaware of. [One can imagine the dependending package asking for dependencies with the platforms it knows about; other platforms it doesn’t know how to ask for. The platform description in that scenario is a kind of unforagable capability.] The offset bounds checking and definition of `mapOffset` together ensure that this is the case. Discovering a new offset is discovering a new platform, and since those platforms weren’t in the derivation “spec” of the needing package, they cannot be relevant. From a capability perspective, we can imagine that the host and target

platforms of a package are the capabilities a package requires, and the depending package must provide the capability to the dependency.

6.3.1. Variables specifying dependencies

6.3.1.1. `depsBuildBuild`

A list of dependencies whose host and target platforms are the new derivation's build platform. This means a `-1` host and `-1` target offset from the new derivation's platforms. These are programs and libraries used at build time that produce programs and libraries also used at build time. If the dependency doesn't care about the target platform (i.e. isn't a compiler or similar tool), put it in `nativeBuildInputs` instead. The most common use of this `buildPackages.stdenv.cc`, the default C compiler for this role. That example crops up more than one might think in old commonly used C libraries.

Since these packages are able to be run at build-time, they are always added to the `PATH`, as described above. But since these packages are only guaranteed to be able to run then, they shouldn't persist as run-time dependencies. This isn't currently enforced, but could be in the future.

6.3.1.2. `nativeBuildInputs`

A list of dependencies whose host platform is the new derivation's build platform, and target platform is the new derivation's host platform. This means a `-1` host offset and `0` target offset from the new derivation's platforms. These are programs and libraries used at build-time that, if they are a compiler or similar tool, produce code to run at run-time—i.e. tools used to build the new derivation. If the dependency doesn't care about the target platform (i.e. isn't a compiler or similar tool), put it here, rather than in `depsBuildBuild` or `depsBuildTarget`. This could be called `depsBuildHost` but `nativeBuildInputs` is used for historical continuity.

Since these packages are able to be run at build-time, they are added to the `PATH`, as described above. But since these packages are only guaranteed to be able to run then, they shouldn't persist as run-time dependencies. This isn't currently enforced, but could be in the future.

6.3.1.3. `depsBuildTarget`

A list of dependencies whose host platform is the new derivation's build platform, and target platform is the new derivation's target platform. This means a `-1` host offset and `1` target offset from the new derivation's platforms. These are programs used at build time that produce code to run with code produced by the depending package. Most commonly, these are tools used to build the runtime or standard library that the currently-being-built compiler will inject into any code it compiles. In many cases, the currently-being-built-compiler is itself employed for that task, but when that compiler won't run (i.e. its build and host platform differ) this is not possible. Other times, the compiler relies on some other tool, like binutils, that is always built separately so that the dependency is unconditional.

This is a somewhat confusing concept to wrap one's head around, and for good reason. As the only dependency type where the platform offsets are not adjacent integers, it requires thinking of a

bootstrapping stage *two* away from the current one. It and its use-case go hand in hand and are both considered poor form: try to not need this sort of dependency, and try to avoid building standard libraries and runtimes in the same derivation as the compiler produces code using them. Instead strive to build those like a normal library, using the newly-built compiler just as a normal library would. In short, do not use this attribute unless you are packaging a compiler and are sure it is needed.

Since these packages are able to run at build time, they are added to the `PATH`, as described above. But since these packages are only guaranteed to be able to run then, they shouldn't persist as run-time dependencies. This isn't currently enforced, but could be in the future.

6.3.1.4. `depsHostHost`

A list of dependencies whose host and target platforms match the new derivation's host platform. This means a `0` host offset and `0` target offset from the new derivation's host platform. These are packages used at run-time to generate code also used at run-time. In practice, this would usually be tools used by compilers for macros or a metaprogramming system, or libraries used by the macros or metaprogramming code itself. It's always preferable to use a `depsBuildBuild` dependency in the derivation being built over a `depsHostHost` on the tool doing the building for this purpose.

6.3.1.5. `buildInputs`

A list of dependencies whose host platform and target platform match the new derivation's. This means a `0` host offset and a `1` target offset from the new derivation's host platform. This would be called `depsHostTarget` but for historical continuity. If the dependency doesn't care about the target platform (i.e. isn't a compiler or similar tool), put it here, rather than in `depsBuildBuild`.

These are often programs and libraries used by the new derivation at *run*-time, but that isn't always the case. For example, the machine code in a statically-linked library is only used at run-time, but the derivation containing the library is only needed at build-time. Even in the dynamic case, the library may also be needed at build-time to appease the linker.

6.3.1.6. `depsTargetTarget`

A list of dependencies whose host platform matches the new derivation's target platform. This means a `1` offset from the new derivation's platforms. These are packages that run on the target platform, e.g. the standard library or run-time deps of standard library that a compiler insists on knowing about. It's poor form in almost all cases for a package to depend on another from a future stage [future stage corresponding to positive offset]. Do not use this attribute unless you are packaging a compiler and are sure it is needed.

6.3.1.7. `depsBuildBuildPropagated`

The propagated equivalent of `depsBuildBuild`. This perhaps never ought to be used, but it is included for consistency [see below for the others].

6.3.1.8. `propagatedNativeBuildInputs`

The propagated equivalent of `nativeBuildInputs`. This would be called `depsBuildHostPropagated` but for historical continuity. For example, if package `Y` has `propagatedNativeBuildInputs = [X]`, and package `Z` has `buildInputs = [Y]`, then package `Z` will be built as if it included package `X` in its `nativeBuildInputs`. If instead, package `Z` has `nativeBuildInputs = [Y]`, then `Z` will be built as if it included `X` in the `depsBuildBuild` of package `Z`, because of the sum of the two `-1` host offsets.

6.3.1.9. `depsBuildTargetPropagated`

The propagated equivalent of `depsBuildTarget`. This is prefixed for the same reason of alerting potential users.

6.3.1.10. `depsHostHostPropagated`

The propagated equivalent of `depsHostHost`.

6.3.1.11. `propagatedBuildInputs`

The propagated equivalent of `buildInputs`. This would be called `depsHostTargetPropagated` but for historical continuity.

6.3.1.12. `depsTargetTargetPropagated`

The propagated equivalent of `depsTargetTarget`. This is prefixed for the same reason of alerting potential users.

6.4. Attributes

6.4.1. Variables affecting `stdenv` initialisation

6.4.1.1. `NIX_DEBUG`

A natural number indicating how much information to log. If set to 1 or higher, `stdenv` will print moderate debugging information during the build. In particular, the `gcc` and `ld` wrapper scripts will print out the complete command line passed to the wrapped tools. If set to 6 or higher, the `stdenv` setup script will be run with `set -x` tracing. If set to 7 or higher, the `gcc` and `ld` wrapper scripts will also be run with `set -x` tracing.

6.4.2. Attributes affecting build properties

6.4.2.1. `enableParallelBuilding`

If set to `true`, `stdenv` will pass specific flags to `make` and other build tools to enable parallel building with up to `build-cores` workers.

Unless set to `false`, some build systems with good support for parallel building including `cmake`, `meson`, and `qmake` will set it to `true`.

6.4.3. Special variables

6.4.3.1. `passthru`

This is an attribute set which can be filled with arbitrary values. For example:

```
passthru = {
  foo = "bar";
  baz = {
    value1 = 4;
    value2 = 5;
  };
}
```

Values inside it are not passed to the builder, so you can change them without triggering a rebuild. However, they can be accessed outside of a derivation directly, as if they were set inside a derivation itself, e.g. `hello.baz.value1`. We don't specify any usage or schema of `passthru` - it is meant for values that would be useful outside the derivation in other parts of a Nix expression (e.g. in other derivations). An example would be to convey some specific dependency of your derivation which contains a program with plugins support. Later, others who make derivations with plugins can use passed-through dependency to ensure that their plugin would be binary-compatible with built program.

6.4.3.2. `passthru.updateScript`

A script to be run by `maintainers/scripts/update.nix` when the package is matched. It needs to be an executable file, either on the file system:

```
passthru.updateScript = ./update.sh;
```

or inside the expression itself:

```
passthru.updateScript = writeScript "update-zoom-us" ''
  #!/usr/bin/env nix-shell
  #!nix-shell -i bash -p curl pcre common-updater-scripts

  set -eu -o pipefail
```

```
  version="$(curl -sI https://zoom.us/client/latest/zoom_x86_64.tar.xz | gr
  update-source-version zoom-us "$version"
'';
```

The attribute can also contain a list, a script followed by arguments to be passed to it:

```
passthru.updateScript = [ ../../update.sh pname "--requested-release=unstab
```

The script will be run with `UPDATE_NIX_ATTR_PATH` environment variable set to the attribute path it is supposed to update.

> **Note:** The script will be usually run from the root of the Nixpkgs repository but you should not rely on that. Also note that the update scripts will be run in parallel by default; you should avoid running `git commit` or any other commands that cannot handle that.

For information about how to run the updates, execute `nix-shell maintainers/scripts/update.nix`.

6.5. Phases

The generic builder has a number of *phases*. Package builds are split into phases to make it easier to override specific parts of the build (e.g., unpacking the sources or installing the binaries). Furthermore, it allows a nicer presentation of build logs in the Nix build farm.

Each phase can be overridden in its entirety either by setting the environment variable `namePhase` to a string containing some shell commands to be executed, or by redefining the shell function `namePhase`. The former is convenient to override a phase from the derivation, while the latter is convenient from a build script. However, typically one only wants to *add* some commands to a phase, e.g. by defining `postInstall` or `preFixup`, as skipping some of the default actions may have unexpected consequences. The default script for each phase is defined in the file `pkgs/stdenv/generic/setup.sh`.

6.5.1. Controlling phases

There are a number of variables that control what phases are executed and in what order:

6.5.1.1. Variables affecting phase control

6.5.1.1.1. phases

Specifies the phases. You can change the order in which phases are executed, or add new phases, by setting this variable. If it's not set, the default value is used, which is

```
$prePhases unpackPhase patchPhase $preConfigurePhases configurePhase
$preBuildPhases buildPhase checkPhase $preInstallPhases installPhase
fixupPhase installCheckPhase $preDistPhases distPhase $postPhases
```

.

Usually, if you just want to add a few phases, it's more convenient to set one of the variables below (such as `preInstallPhases`), as you then don't specify all the normal phases.

6.5.1.1.2. `prePhases`

Additional phases executed before any of the default phases.

6.5.1.1.3. `preConfigurePhases`

Additional phases executed just before the configure phase.

6.5.1.1.4. `preBuildPhases`

Additional phases executed just before the build phase.

6.5.1.1.5. `preInstallPhases`

Additional phases executed just before the install phase.

6.5.1.1.6. `preFixupPhases`

Additional phases executed just before the fixup phase.

6.5.1.1.7. `preDistPhases`

Additional phases executed just before the distribution phase.

6.5.1.1.8. `postPhases`

Additional phases executed after any of the default phases.

6.5.2. The unpack phase

The unpack phase is responsible for unpacking the source code of the package. The default implementation of `unpackPhase` unpacks the source files listed in the `src` environment variable to the current directory. It supports the following files by default:

6.5.2.1. Tar files

These can optionally be compressed using `gzip` (`.tar.gz`, `.tgz` or `.tar.Z`), `bzip2` (`.tar.bz2`,

`.tbz2` or `.tbz`) or `xz` (`.tar.xz`, `.tar.lzma` or `.txz`).

6.5.2.2. Zip files

Zip files are unpacked using `unzip`. However, `unzip` is not in the standard environment, so you should add it to `nativeBuildInputs` yourself.

6.5.2.3. Directories in the Nix store

These are simply copied to the current directory. The hash part of the file name is stripped, e.g. `/nix/store/1wydxgby13cz...-my-sources` would be copied to `my-sources`.

Additional file types can be supported by setting the `unpackCmd` variable (see below).

6.5.2.4. Variables controlling the unpack phase

6.5.2.4.1. `srcs` / `src`

The list of source files or directories to be unpacked or copied. One of these must be set.

6.5.2.4.2. `sourceRoot`

After running `unpackPhase`, the generic builder changes the current directory to the directory created by unpacking the sources. If there are multiple source directories, you should set `sourceRoot` to the name of the intended directory.

6.5.2.4.3. `setSourceRoot`

Alternatively to setting `sourceRoot`, you can set `setSourceRoot` to a shell command to be evaluated by the unpack phase after the sources have been unpacked. This command must set `sourceRoot`.

6.5.2.4.4. `preUnpack`

Hook executed at the start of the unpack phase.

6.5.2.4.5. `postUnpack`

Hook executed at the end of the unpack phase.

6.5.2.4.6. `dontUnpack`

Set to true to skip the unpack phase.

6.5.2.4.7. `dontMakeSourcesWritable`

If set to `1`, the unpacked sources are *not* made writable. By default, they are made writable to prevent problems with read-only sources. For example, copied store directories would be read-only without this.

6.5.2.4.8. `unpackCmd`

The unpack phase evaluates the string `$unpackCmd` for any unrecognised file. The path to the current source file is contained in the `curSrc` variable.

6.5.3. The patch phase

The patch phase applies the list of patches defined in the `patches` variable.

6.5.3.1. Variables controlling the patch phase

6.5.3.1.1. `dontPatch`

Set to true to skip the patch phase.

6.5.3.1.2. `patches`

The list of patches. They must be in the format accepted by the `patch` command, and may optionally be compressed using `gzip` (`.gz`), `bzip2` (`.bz2`) or `xz` (`.xz`).

6.5.3.1.3. `patchFlags`

Flags to be passed to `patch`. If not set, the argument `-p1` is used, which causes the leading directory component to be stripped from the file names in each patch.

6.5.3.1.4. `prePatch`

Hook executed at the start of the patch phase.

6.5.3.1.5. `postPatch`

Hook executed at the end of the patch phase.

6.5.4. The configure phase

The configure phase prepares the source tree for building. The default `configurePhase` runs `./configure` (typically an Autoconf-generated script) if it exists.

6.5.4.1. Variables controlling the configure phase

6.5.4.1.1. `configureScript`

The name of the configure script. It defaults to `./configure` if it exists; otherwise, the configure phase is skipped. This can actually be a command (like `perl ./Configure.pl`).

6.5.4.1.2. `configureFlags`

A list of strings passed as additional arguments to the configure script.

6.5.4.1.3. `dontConfigure`

Set to true to skip the configure phase.

6.5.4.1.4. `configureFlagsArray`

A shell array containing additional arguments passed to the configure script. You must use this instead of `configureFlags` if the arguments contain spaces.

6.5.4.1.5. `dontAddPrefix`

By default, the flag `--prefix=$prefix` is added to the configure flags. If this is undesirable, set this variable to true.

6.5.4.1.6. `prefix`

The prefix under which the package must be installed, passed via the `--prefix` option to the configure script. It defaults to `$out`.

6.5.4.1.7. `prefixKey`

The key to use when specifying the prefix. By default, this is set to `--prefix=` as that is used by the majority of packages.

6.5.4.1.8. `dontAddDisableDepTrack`

By default, the flag `--disable-dependency-tracking` is added to the configure flags to speed up Automake-based builds. If this is undesirable, set this variable to true.

6.5.4.1.9. `dontFixLibtool`

By default, the configure phase applies some special hackery to all files called `ltmain.sh` before running the configure script in order to improve the purity of Libtool-based packages [4] . If this is undesirable, set this variable to true.

6.5.4.1.10. `dontDisableStatic`

By default, when the configure script has `--enable-static`, the option `--disable-static` is added to the configure flags.

If this is undesirable, set this variable to true.

6.5.4.1.11. `configurePlatforms`

By default, when cross compiling, the configure script has `--build=...` and `--host=...` passed. Packages can instead pass `[ "build" "host" "target" ]` or a subset to control exactly which platform flags are passed. Compilers and other tools can use this to also pass the target platform. [5]

6.5.4.1.12. `preConfigure`

Hook executed at the start of the configure phase.

6.5.4.1.13. `postConfigure`

Hook executed at the end of the configure phase.

6.5.5. The build phase

The build phase is responsible for actually building the package (e.g. compiling it). The default `buildPhase` simply calls `make` if a file named `Makefile`, `makefile` or `GNUmakefile` exists in the current directory (or the `makefile` is explicitly set); otherwise it does nothing.

6.5.5.1. Variables controlling the build phase

6.5.5.1.1. `dontBuild`

Set to true to skip the build phase.

6.5.5.1.2. `makefile`

The file name of the Makefile.

6.5.5.1.3. `makeFlags`

A list of strings passed as additional flags to `make`. These flags are also used by the default install and check phase. For setting make flags specific to the build phase, use `buildFlags` (see below).

```
makeFlags = [ "PREFIX=$(out)" ];
```

Note: The flags are quoted in bash, but environment variables can be specified by using the make

syntax.

6.5.5.1.4. `makeFlagsArray`

A shell array containing additional arguments passed to `make`. You must use this instead of `makeFlags` if the arguments contain spaces, e.g.

```
preBuild = ''
  makeFlagsArray+=(CFLAGS="-O0 -g" LDFLAGS="-lfoo -lbar")
'';
```

Note that shell arrays cannot be passed through environment variables, so you cannot set `makeFlagsArray` in a derivation attribute (because those are passed through environment variables): you have to define them in shell code.

6.5.5.1.5. `buildFlags` / `buildFlagsArray`

A list of strings passed as additional flags to `make`. Like `makeFlags` and `makeFlagsArray`, but only used by the build phase.

6.5.5.1.6. `preBuild`

Hook executed at the start of the build phase.

6.5.5.1.7. `postBuild`

Hook executed at the end of the build phase.

You can set flags for `make` through the `makeFlags` variable.

Before and after running `make`, the hooks `preBuild` and `postBuild` are called, respectively.

6.5.6. The check phase

The check phase checks whether the package was built correctly by running its test suite. The default `checkPhase` calls `make check`, but only if the `doCheck` variable is enabled.

6.5.6.1. Variables controlling the check phase

6.5.6.1.1. doCheck

Controls whether the check phase is executed. By default it is skipped, but if `doCheck` is set to true, the check phase is usually executed. Thus you should set

```
doCheck = true;
```

in the derivation to enable checks. The exception is cross compilation. Cross compiled builds never run tests, no matter how `doCheck` is set, as the newly-built program won't run on the platform used to build it.

6.5.6.1.2. `makeFlags` / `makeFlagsArray` / `makefile`

See the build phase for details.

6.5.6.1.3. `checkTarget`

The make target that runs the tests. Defaults to `check`.

6.5.6.1.4. `checkFlags` / `checkFlagsArray`

A list of strings passed as additional flags to `make`. Like `makeFlags` and `makeFlagsArray`, but only used by the check phase.

6.5.6.1.5. `checkInputs`

A list of dependencies used by the phase. This gets included in `nativeBuildInputs` when `doCheck` is set.

6.5.6.1.6. `preCheck`

Hook executed at the start of the check phase.

6.5.6.1.7. `postCheck`

Hook executed at the end of the check phase.

6.5.7. The install phase

The install phase is responsible for installing the package in the Nix store under `out`. The default `installPhase` creates the directory `$out` and calls `make install`.

6.5.7.1. Variables controlling the install phase

6.5.7.1.1. `dontInstall`

Set to true to skip the install phase.

6.5.7.1.2. `makeFlags` / `makeFlagsArray` / `makefile`

See the build phase for details.

6.5.7.1.3. `installTargets`

The make targets that perform the installation. Defaults to `install`. Example:

```
installTargets = "install-bin install-doc";
```

6.5.7.1.4. `installFlags` / `installFlagsArray`

A list of strings passed as additional flags to `make`. Like `makeFlags` and `makeFlagsArray`, but only used by the install phase.

6.5.7.1.5. `preInstall`

Hook executed at the start of the install phase.

6.5.7.1.6. `postInstall`

Hook executed at the end of the install phase.

6.5.8. The fixup phase

The fixup phase performs some (Nix-specific) post-processing actions on the files installed under `$out` by the install phase. The default `fixupPhase` does the following:

- It moves the `man/`, `doc/` and `info/` subdirectories of `$out` to `share/`.
- It strips libraries and executables of debug information.
- On Linux, it applies the `patchelf` command to ELF executables and libraries to remove unused directories from the `RPATH` in order to prevent unnecessary runtime dependencies.
- It rewrites the interpreter paths of shell scripts to paths found in `PATH`. E.g., `/usr/bin/perl` will be rewritten to `/nix/store/some-perl/bin/perl` found in `PATH`.

6.5.8.1. Variables controlling the fixup phase

6.5.8.1.1. `dontFixup`

Set to true to skip the fixup phase.

6.5.8.1.2. `dontStrip`

If set, libraries and executables are not stripped. By default, they are.

6.5.8.1.3. `dontStripHost`

Like `dontStrip`, but only affects the `strip` command targetting the package's host platform. Useful when supporting cross compilation, but otherwise feel free to ignore.

6.5.8.1.4. `dontStripTarget`

Like `dontStrip`, but only affects the `strip` command targetting the packages' target platform. Useful when supporting cross compilation, but otherwise feel free to ignore.

6.5.8.1.5. `dontMoveSbin`

If set, files in `$out/sbin` are not moved to `$out/bin`. By default, they are.

6.5.8.1.6. `stripAllList`

List of directories to search for libraries and executables from which *all* symbols should be stripped. By default, it's empty. Stripping all symbols is risky, since it may remove not just debug symbols but also ELF information necessary for normal execution.

6.5.8.1.7. `stripAllFlags`

Flags passed to the `strip` command applied to the files in the directories listed in `stripAllList`. Defaults to `-s` (i.e. `--strip-all`).

6.5.8.1.8. `stripDebugList`

List of directories to search for libraries and executables from which only debugging-related symbols should be stripped. It defaults to `lib lib32 lib64 libexec bin sbin`.

6.5.8.1.9. `stripDebugFlags`

Flags passed to the `strip` command applied to the files in the directories listed in `stripDebugList`. Defaults to `-S` (i.e. `--strip-debug`).

6.5.8.1.10. `dontPatchELF`

If set, the `patchelf` command is not used to remove unnecessary `RPATH` entries. Only applies to Linux.

6.5.8.1.11. `dontPatchShebangs`

If set, scripts starting with `#!` do not have their interpreter paths rewritten to paths in the Nix store.

6.5.8.1.12. `dontPruneLibtoolFiles`

If set, libtool `.la` files associated with shared libraries won't have their `dependency_libs` field cleared.

6.5.8.1.13. `forceShare`

The list of directories that must be moved from `$out` to `$out/share`. Defaults to `man doc info`.

6.5.8.1.14. `setupHook`

A package can export a setup hook by setting this variable. The setup hook, if defined, is copied to `$out/nix-support/setup-hook`. Environment variables are then substituted in it using `substituteAll`.

6.5.8.1.15. `preFixup`

Hook executed at the start of the fixup phase.

6.5.8.1.16. `postFixup`

Hook executed at the end of the fixup phase.

6.5.8.1.17. `separateDebugInfo`

If set to `true`, the standard environment will enable debug information in C/C++ builds. After installation, the debug information will be separated from the executables and stored in the output named `debug`. (This output is enabled automatically; you don't need to set the `outputs` attribute explicitly.) To be precise, the debug information is stored in `debug/lib/debug/.build-id/XX/YYYY…`, where <XXYYYY…> is the <build ID> of the binary — a SHA-1 hash of the contents of the binary. Debuggers like GDB use the build ID to look up the separated debug information.

For example, with GDB, you can add

```
set debug-file-directory ~/.nix-profile/lib/debug
```

to `~/.gdbinit`. GDB will then be able to find debug information installed via `nix-env -i`.

6.5.9. The installCheck phase

The installCheck phase checks whether the package was installed correctly by running its test suite against the installed directories. The default `installCheck` calls `make installcheck`.

6.5.9.1. Variables controlling the installCheck phase

6.5.9.1.1. `doInstallCheck`

Controls whether the installCheck phase is executed. By default it is skipped, but if `doInstallCheck` is set to true, the installCheck phase is usually executed. Thus you should set

```
doInstallCheck = true;
```

in the derivation to enable install checks. The exception is cross compilation. Cross compiled builds never run tests, no matter how `doInstallCheck` is set, as the newly-built program won't run on the platform used to build it.

6.5.9.1.2. `installCheckTarget`

The make target that runs the install tests. Defaults to `installcheck`.

6.5.9.1.3. `installCheckFlags` / `installCheckFlagsArray`

A list of strings passed as additional flags to `make`. Like `makeFlags` and `makeFlagsArray`, but only used by the installCheck phase.

6.5.9.1.4. `installCheckInputs`

A list of dependencies used by the phase. This gets included in `nativeBuildInputs` when `doInstallCheck` is set.

6.5.9.1.5. `preInstallCheck`

Hook executed at the start of the installCheck phase.

6.5.9.1.6. `postInstallCheck`

Hook executed at the end of the installCheck phase.

6.5.10. The distribution phase

The distribution phase is intended to produce a source distribution of the package. The default `distPhase` first calls `make dist`, then it copies the resulting source tarballs to `$out/tarballs/`. This phase is only executed if the attribute `doDist` is set.

6.5.10.1. Variables controlling the distribution phase

6.5.10.1.1. `distTarget`

The make target that produces the distribution. Defaults to `dist`.

6.5.10.1.2. `distFlags` / `distFlagsArray`

Additional flags passed to `make`.

6.5.10.1.3. `tarballs`

The names of the source distribution files to be copied to `$out/tarballs/`. It can contain shell wildcards. The default is `*.tar.gz`.

6.5.10.1.4. `dontCopyDist`

If set, no files are copied to `$out/tarballs/`.

6.5.10.1.5. `preDist`

Hook executed at the start of the distribution phase.

6.5.10.1.6. `postDist`

Hook executed at the end of the distribution phase.

6.6. Shell functions

The standard environment provides a number of useful functions.

6.6.1. `makeWrapper` <executable> <wrapperfile> <args>

Constructs a wrapper for a program with various possible arguments. For example:

```
# adds `FOOBAR=baz` to `$out/bin/foo`'s environment
makeWrapper $out/bin/foo $wrapperfile --set FOOBAR baz

# prefixes the binary paths of `hello` and `git`
# Be advised that paths often should be patched in directly
# (via string replacements or in `configurePhase`).
makeWrapper $out/bin/foo $wrapperfile --prefix PATH : ${lib.makeBinPath [ h
```

There's many more kinds of arguments, they are documented in `nixpkgs/pkgs/build-support/setup-hooks/make-wrapper.sh`.

`wrapProgram` is a convenience function you probably want to use most of the time.

6.6.2. `substitute` <infile> <outfile> <subs>

Performs string substitution on the contents of <infile>, writing the result to <outfile>. The substitutions in <subs> are of the following form:

6.6.2.1. `--replace` <s1> <s2>

Replace every occurrence of the string <s1> by <s2>.

6.6.2.2. `--subst-var` <varName>

Replace every occurrence of `@varName@` by the contents of the environment variable <varName>. This is useful for generating files from templates, using `@...@` in the template as placeholders.

6.6.2.3. `--subst-var-by` <varName> <s>

Replace every occurrence of `@varName@` by the string <s>.

Example:

```
substitute ./foo.in ./foo.out \
    --replace /usr/bin/bar $bar/bin/bar \
    --replace "a string containing spaces" "some other text" \
    --subst-var someVar
```

6.6.3. `substituteInPlace` <file> <subs>

Like `substitute`, but performs the substitutions in place on the file <file>.

6.6.4. `substituteAll` <infile> <outfile>

Replaces every occurrence of `@varName@`, where <varName> is any environment variable, in <infile>, writing the result to <outfile>. For instance, if <infile> has the contents

```
#! @bash@/bin/sh
PATH=@coreutils@/bin
echo @foo@
```

and the environment contains `bash=/nix/store/bmwp0q28cf21...-bash-3.2-p39` and `coreutils=/nix/store/68afga4khv0w...-coreutils-6.12`, but does not contain the variable `foo`, then the output will be

```
#! /nix/store/bmwp0q28cf21...-bash-3.2-p39/bin/sh
PATH=/nix/store/68afga4khv0w...-coreutils-6.12/bin
echo @foo@
```

That is, no substitution is performed for undefined variables.

Environment variables that start with an uppercase letter or an underscore are filtered out, to prevent global variables (like `HOME`) or private variables (like `__ETC_PROFILE_DONE`) from accidentally getting substituted. The variables also have to be valid bash "names", as defined in the bash manpage (alphanumeric or _, must not start with a number).

6.6.5. substituteAllInPlace <file>

Like `substituteAll`, but performs the substitutions in place on the file <file>.

6.6.6. stripHash <path>

Strips the directory and hash part of a store path, outputting the name part to `stdout`. For example:

```
# prints coreutils-8.24
stripHash "/nix/store/9s9r019176g7cvn2nvcw41gsp862y6b4-coreutils-8.24"
```

If you wish to store the result in another variable, then the following idiom may be useful:

```
name="/nix/store/9s9r019176g7cvn2nvcw41gsp862y6b4-coreutils-8.24"
someVar=$(stripHash $name)
```

6.6.7. wrapProgram <executable> <makeWrapperArgs>

Convenience function for `makeWrapper` that automatically creates a sane wrapper file. It takes all the same arguments as `makeWrapper`, except for `--argv0`.

It cannot be applied multiple times, since it will overwrite the wrapper file.

6.7. Package setup hooks

Nix itself considers a build-time dependency as merely something that should previously be built and accessible at build time—packages themselves are on their own to perform any additional setup. In most cases, that is fine, and the downstream derivation can deal with its own dependencies. But for a few common tasks, that would result in almost every package doing the same sort of setup work—depending not on the package itself, but entirely on which dependencies were used.

In order to alleviate this burden, the setup hook mechanism was written, where any package can include a

shell script that [by convention rather than enforcement by Nix], any downstream reverse-dependency will source as part of its build process. That allows the downstream dependency to merely specify its dependencies, and lets those dependencies effectively initialize themselves. No boilerplate mirroring the list of dependencies is needed.

The setup hook mechanism is a bit of a sledgehammer though: a powerful feature with a broad and indiscriminate area of effect. The combination of its power and implicit use may be expedient, but isn't without costs. Nix itself is unchanged, but the spirit of added dependencies being effect-free is violated even if the letter isn't. For example, if a derivation path is mentioned more than once, Nix itself doesn't care and simply makes sure the dependency derivation is already built just the same—depending is just needing something to exist, and needing is idempotent. However, a dependency specified twice will have its setup hook run twice, and that could easily change the build environment (though a well-written setup hook will therefore strive to be idempotent so this is in fact not observable). More broadly, setup hooks are anti-modular in that multiple dependencies, whether the same or different, should not interfere and yet their setup hooks may well do so.

The most typical use of the setup hook is actually to add other hooks which are then run (i.e. after all the setup hooks) on each dependency. For example, the C compiler wrapper's setup hook feeds itself flags for each dependency that contains relevant libraries and headers. This is done by defining a bash function, and appending its name to one of `envBuildBuildHooks`, `envBuildHostHooks`, `envBuildTargetHooks`, `envHostHostHooks`, `envHostTargetHooks`, or `envTargetTargetHooks`. These 6 bash variables correspond to the 6 sorts of dependencies by platform (there's 12 total but we ignore the propagated/non-propagated axis).

Packages adding a hook should not hard code a specific hook, but rather choose a variable *relative* to how they are included. Returning to the C compiler wrapper example, if the wrapper itself is an `n` dependency, then it only wants to accumulate flags from `n + 1` dependencies, as only those ones match the compiler's target platform. The `hostOffset` variable is defined with the current dependency's host offset `targetOffset` with its target offset, before its setup hook is sourced. Additionally, since most environment hooks don't care about the target platform, that means the setup hook can append to the right bash array by doing something like

```
addEnvHooks "$hostOffset" myBashFunction
```

The *existence* of setups hooks has long been documented and packages inside Nixpkgs are free to use this mechanism. Other packages, however, should not rely on these mechanisms not changing between Nixpkgs versions. Because of the existing issues with this system, there's little benefit from mandating it be stable for any period of time.

First, let's cover some setup hooks that are part of Nixpkgs default stdenv. This means that they are run for every package built using `stdenv.mkDerivation`. Some of these are platform specific, so they may run on Linux but not Darwin or vice-versa.

6.7.1. `move-docs.sh`

This setup hook moves any installed documentation to the `/share` subdirectory directory. This includes the man, doc and info directories. This is needed for legacy programs that do not know how to use the `share` subdirectory.

6.7.2. `compress-man-pages.sh`

This setup hook compresses any man pages that have been installed. The compression is done using the gzip program. This helps to reduce the installed size of packages.

6.7.3. `strip.sh`

This runs the strip command on installed binaries and libraries. This removes unnecessary information like debug symbols when they are not needed. This also helps to reduce the installed size of packages.

6.7.4. `patch-shebangs.sh`

This setup hook patches installed scripts to use the full path to the shebang interpreter. A shebang interpreter is the first commented line of a script telling the operating system which program will run the script (e.g `#!/bin/bash`). In Nix, we want an exact path to that interpreter to be used. This often replaces `/bin/sh` with a path in the Nix store.

6.7.5. `audit-tmpdir.sh`

This verifies that no references are left from the install binaries to the directory used to build those binaries. This ensures that the binaries do not need things outside the Nix store. This is currently supported in Linux only.

6.7.6. `multiple-outputs.sh`

This setup hook adds configure flags that tell packages to install files into any one of the proper outputs listed in `outputs`. This behavior can be turned off by setting `setOutputFlags` to false in the derivation environment. See Chapter 8, *Multiple-output packages* for more information.

6.7.7. `move-sbin.sh`

This setup hook moves any binaries installed in the `sbin/` subdirectory into `bin/`. In addition, a link is provided from `sbin/` to `bin/` for compatibility.

6.7.8. `move-lib64.sh`

This setup hook moves any libraries installed in the `lib64/` subdirectory into `lib/`. In addition, a link is provided from `lib64/` to `lib/` for compatibility.

6.7.9. `move-systemd-user-units.sh`

This setup hook moves any systemd user units installed in the `lib/` subdirectory into `share/`. In addition, a link is provided from `share/` to `lib/` for compatibility. This is needed for systemd to find user services when installed into the user profile.

6.7.10. `set-source-date-epoch-to-latest.sh`

This sets `SOURCE_DATE_EPOCH` to the modification time of the most recent file.

6.7.11. Bintools Wrapper

The Bintools Wrapper wraps the binary utilities for a bunch of miscellaneous purposes. These are GNU Binutils when targetting Linux, and a mix of cctools and GNU binutils for Darwin. [The "Bintools" name is supposed to be a compromise between "Binutils" and "cctools" not denoting any specific implementation.] Specifically, the underlying bintools package, and a C standard library (glibc or Darwin's libSystem, just for the dynamic loader) are all fed in, and dependency finding, hardening (see below), and purity checks for each are handled by the Bintools Wrapper. Packages typically depend on CC Wrapper, which in turn (at run time) depends on the Bintools Wrapper.

The Bintools Wrapper was only just recently split off from CC Wrapper, so the division of labor is still being worked out. For example, it shouldn't care about the C standard library, but just take a derivation with the dynamic loader (which happens to be the glibc on linux). Dependency finding however is a task both wrappers will continue to need to share, and probably the most important to understand. It is currently accomplished by collecting directories of host-platform dependencies (i.e. `buildInputs` and `nativeBuildInputs`) in environment variables. The Bintools Wrapper's setup hook causes any `lib` and `lib64` subdirectories to be added to `NIX_LDFLAGS`. Since the CC Wrapper and the Bintools Wrapper use the same strategy, most of the Bintools Wrapper code is sparsely commented and refers to the CC Wrapper. But the CC Wrapper's code, by contrast, has quite lengthy comments. The Bintools Wrapper merely cites those, rather than repeating them, to avoid falling out of sync.

A final task of the setup hook is defining a number of standard environment variables to tell build systems which executables fulfill which purpose. They are defined to just be the base name of the tools, under the assumption that the Bintools Wrapper's binaries will be on the path. Firstly, this helps poorly-written packages, e.g. ones that look for just `gcc` when `CC` isn't defined yet `clang` is to be used. Secondly, this helps packages not get confused when cross-compiling, in which case multiple Bintools Wrappers may simultaneously be in use. [6] `BUILD_`- and `TARGET_`-prefixed versions of the normal environment variable are defined for additional Bintools Wrappers, properly disambiguating them.

A problem with this final task is that the Bintools Wrapper is honest and defines `LD` as `ld`. Most packages, however, firstly use the C compiler for linking, secondly use `LD` anyways, defining it as the C compiler, and thirdly, only so define `LD` when it is undefined as a fallback. This triple-threat means Bintools Wrapper will break those packages, as LD is already defined as the actual linker which the package won't override yet doesn't want to use. The workaround is to define, just for the problematic package, `LD` as the C compiler. A good way to do this would be `preConfigure = "LD=$CC"`.

6.7.12. CC Wrapper

The CC Wrapper wraps a C toolchain for a bunch of miscellaneous purposes. Specifically, a C compiler (GCC or Clang), wrapped binary tools, and a C standard library (glibc or Darwin's libSystem, just for the dynamic loader) are all fed in, and dependency finding, hardening (see below), and purity checks for each are handled by the CC Wrapper. Packages typically depend on the CC Wrapper, which in turn (at run-time) depends on the Bintools Wrapper.

Dependency finding is undoubtedly the main task of the CC Wrapper. This works just like the Bintools Wrapper, except that any `include` subdirectory of any relevant dependency is added to `NIX_CFLAGS_COMPILE`. The setup hook itself contains some lengthy comments describing the exact convoluted mechanism by which this is accomplished.

Similarly, the CC Wrapper follows the Bintools Wrapper in defining standard environment variables with the names of the tools it wraps, for the same reasons described above. Importantly, while it includes a `cc` symlink to the c compiler for portability, the `CC` will be defined using the compiler's "real name" (i.e. `gcc` or `clang`). This helps lousy build systems that inspect on the name of the compiler rather than run it.

Here are some more packages that provide a setup hook. Since the list of hooks is extensible, this is not an exhaustive list. The mechanism is only to be used as a last resort, so it might cover most uses.

6.7.13. Perl

Adds the `lib/site_perl` subdirectory of each build input to the `PERL5LIB` environment variable. For instance, if `buildInputs` contains Perl, then the `lib/site_perl` subdirectory of each input is added to the `PERL5LIB` environment variable.

6.7.14. Python

Adds the `lib/${python.libPrefix}/site-packages` subdirectory of each build input to the `PYTHONPATH` environment variable.

6.7.15. pkg-config

Adds the `lib/pkgconfig` and `share/pkgconfig` subdirectories of each build input to the `PKG_CONFIG_PATH` environment variable.

6.7.16. Automake

Adds the `share/aclocal` subdirectory of each build input to the `ACLOCAL_PATH` environment variable.

6.7.17. Autoconf

The `autoreconfHook` derivation adds `autoreconfPhase`, which runs autoreconf, libtoolize and automake, essentially preparing the configure script in autotools-based builds. Most autotools-based packages come with the configure script pre-generated, but this hook is necessary for a few packages

and when you need to patch the package's configure scripts.

6.7.18. libxml2

Adds every file named `catalog.xml` found under the `xml/dtd` and `xml/xsl` subdirectories of each build input to the `XML_CATALOG_FILES` environment variable.

6.7.19. teTeX / TeX Live

Adds the `share/texmf-nix` subdirectory of each build input to the `TEXINPUTS` environment variable.

6.7.20. Qt 4

Sets the `QTDIR` environment variable to Qt's path.

6.7.21. gdk-pixbuf

Exports `GDK_PIXBUF_MODULE_FILE` environment variable to the builder. Add librsvg package to `buildInputs` to get svg support. See also the setup hook description in GNOME platform docs.

6.7.22. GHC

Creates a temporary package database and registers every Haskell build input in it (TODO: how?).

6.7.23. GNOME platform

Hooks related to GNOME platform and related libraries like GLib, GTK and GStreamer are described in Section 15.9, "GNOME".

6.7.24. autoPatchelfHook

This is a special setup hook which helps in packaging proprietary software in that it automatically tries to find missing shared library dependencies of ELF files based on the given `buildInputs` and `nativeBuildInputs`.

You can also specify a `runtimeDependencies` variable which lists dependencies to be unconditionally added to rpath of all executables. This is useful for programs that use dlopen 3 to load libraries at runtime.

In certain situations you may want to run the main command (`autoPatchelf`) of the setup hook on a file or a set of directories instead of unconditionally patching all outputs. This can be done by setting the `dontAutoPatchelf` environment variable to a non-empty value.

By default `autoPatchelf` will fail as soon as any ELF file requires a dependency which cannot be resolved via the given build inputs. In some situations you might prefer to just leave missing dependencies unpatched and continue to patch the rest. This can be achieved by setting the `autoPatchelfIgnoreMissingDeps` environment variable to a non-empty value.

The `autoPatchelf` command also recognizes a `--no-recurse` command line flag, which prevents it from recursing into subdirectories.

6.7.25. breakpointHook

This hook will make a build pause instead of stopping when a failure happens. It prevents nix from cleaning up the build environment immediately and allows the user to attach to a build environment using the `cntr` command. Upon build error it will print instructions on how to use `cntr`, which can be used to enter the environment for debugging. Installing cntr and running the command will provide shell access to the build sandbox of failed build. At `/var/lib/cntr` the sandboxed filesystem is mounted. All commands and files of the system are still accessible within the shell. To execute commands from the sandbox use the cntr exec subcommand. `cntr` is only supported on Linux-based platforms. To use it first add `cntr` to your `environment.systemPackages` on NixOS or alternatively to the root user on non-NixOS systems. Then in the package that is supposed to be inspected, add `breakpointHook` to `nativeBuildInputs`.

```
nativeBuildInputs = [ breakpointHook ];
```

When a build failure happens there will be an instruction printed that shows how to attach with `cntr` to the build sandbox.

> **Note:** This won't work with remote builds as the build environment is on a different machine and can't be accessed by `cntr`. Remote builds can be turned off by setting `--option builders ''` for `nix-build` or `--builders ''` for `nix build`.

6.7.26. installShellFiles

This hook helps with installing manpages and shell completion files. It exposes 2 shell functions `installManPage` and `installShellCompletion` that can be used from your `postInstall` hook.

The `installManPage` function takes one or more paths to manpages to install. The manpages must have a section suffix, and may optionally be compressed (with `.gz` suffix). This function will place them into the correct directory.

The `installShellCompletion` function takes one or more paths to shell completion files. By default it will autodetect the shell type from the completion file extension, but you may also specify it by passing one of `--bash`, `--fish`, or `--zsh`. These flags apply to all paths listed after them (up until another shell flag is given). Each path may also have a custom installation name provided by providing a flag `--name NAME` before the path. If this flag is not provided, zsh completions will be renamed automatically such that `foobar.zsh` becomes `_foobar`. A root name may be provided for all paths using the flag `--cmd NAME`; this synthesizes the appropriate name depending on the shell (e.g. `--cmd foo` will synthesize the name `foo.bash` for bash and `_foo` for zsh). The path may also be a fifo or named fd (such as produced by `<(cmd)`), in which case the shell and name must be provided.

```
nativeBuildInputs = [ installShellFiles ];
postInstall = ''
  installManPage doc/foobar.1 doc/barfoo.3
  # explicit behavior
  installShellCompletion --bash --name foobar.bash share/completions.bash
  installShellCompletion --fish --name foobar.fish share/completions.fish
  installShellCompletion --zsh --name _foobar share/completions.zsh
  # implicit behavior
  installShellCompletion share/completions/foobar.{bash,fish,zsh}
  # using named fd
  installShellCompletion --cmd foobar \
    --bash <($out/bin/foobar --bash-completion) \
    --fish <($out/bin/foobar --fish-completion) \
    --zsh <($out/bin/foobar --zsh-completion)
'';
```

6.7.27. libiconv, libintl

A few libraries automatically add to `NIX_LDFLAGS` their library, making their symbols automatically available to the linker. This includes libiconv and libintl (gettext). This is done to provide compatibility between GNU Linux, where libiconv and libintl are bundled in, and other systems where that might not be the case. Sometimes, this behavior is not desired. To disable this behavior, set `dontAddExtraLibs`.

6.7.28. validatePkgConfig

The `validatePkgConfig` hook validates all pkg-config (`.pc`) files in a package. This helps catching some common errors in pkg-config files, such as undefined variables.

6.7.29. cmake

Overrides the default configure phase to run the CMake command. By default, we use the Make generator of CMake. In addition, dependencies are added automatically to CMAKE_PREFIX_PATH so that packages are correctly detected by CMake. Some additional flags are passed in to give similar behavior to configure-based packages. You can disable this hook's behavior by setting configurePhase to a custom value, or by setting dontUseCmakeConfigure. cmakeFlags controls flags passed only to CMake. By default, parallel building is enabled as CMake supports parallel building almost everywhere. When Ninja is also in use, CMake will detect that and use the ninja generator.

6.7.30. xcbuildHook

Overrides the build and install phases to run the "xcbuild" command. This hook is needed when a project only comes with build files for the XCode build system. You can disable this behavior by setting

buildPhase and configurePhase to a custom value. xcbuildFlags controls flags passed only to xcbuild.

6.7.31. Meson

Overrides the configure phase to run meson to generate Ninja files. To run these files, you should accompany Meson with ninja. By default, `enableParallelBuilding` is enabled as Meson supports parallel building almost everywhere.

6.7.31.1. Variables controlling Meson

6.7.31.1.1. `mesonFlags`

Controls the flags passed to meson.

6.7.31.1.2. `mesonBuildType`

Which `--buildtype` to pass to Meson. We default to `plain`.

6.7.31.1.3. `mesonAutoFeatures`

What value to set `-Dauto_features=` to. We default to `enabled`.

6.7.31.1.4. `mesonWrapMode`

What value to set `-Dwrap_mode=` to. We default to `nodownload` as we disallow network access.

6.7.31.1.5. `dontUseMesonConfigure`

Disables using Meson's `configurePhase`.

6.7.32. ninja

Overrides the build, install, and check phase to run ninja instead of make. You can disable this behavior with the `dontUseNinjaBuild`, `dontUseNinjaInstall`, and `dontUseNinjaCheck`, respectively. Parallel building is enabled by default in Ninja.

6.7.33. unzip

This setup hook will allow you to unzip .zip files specified in `$src`. There are many similar packages like `unrar`, `undmg`, etc.

6.7.34. wafHook

Overrides the configure, build, and install phases. This will run the "waf" script used by many projects. If `wafPath` (default `./waf`) doesn't exist, it will copy the version of waf available in Nixpkgs. `wafFlags`

can be used to pass flags to the waf script.

6.7.35. scons

Overrides the build, install, and check phases. This uses the scons build system as a replacement for make. scons does not provide a configure phase, so everything is managed at build and install time.

6.8. Purity in Nixpkgs

Measures taken to prevent dependencies on packages outside the store, and what you can do to prevent them.

GCC doesn't search in locations such as `/usr/include`. In fact, attempts to add such directories through the `-I` flag are filtered out. Likewise, the linker (from GNU binutils) doesn't search in standard locations such as `/usr/lib`. Programs built on Linux are linked against a GNU C Library that likewise doesn't search in the default system locations.

6.9. Hardening in Nixpkgs

There are flags available to harden packages at compile or link-time. These can be toggled using the `stdenv.mkDerivation` parameters `hardeningDisable` and `hardeningEnable`.

Both parameters take a list of flags as strings. The special `"all"` flag can be passed to `hardeningDisable` to turn off all hardening. These flags can also be used as environment variables for testing or development purposes.

The following flags are enabled by default and might require disabling with `hardeningDisable` if the program to package is incompatible.

6.9.1. format

Adds the `-Wformat -Wformat-security -Werror=format-security` compiler options. At present, this warns about calls to `printf` and `scanf` functions where the format string is not a string literal and there are no format arguments, as in `printf(foo);`. This may be a security hole if the format string came from untrusted input and contains `%n`.

This needs to be turned off or fixed for errors similar to:

```
/tmp/nix-build-zynaddsubfx-2.5.2.drv-0/zynaddsubfx-2.5.2/src/UI/guimain.cpp
         printf(help_message);
                              ^
cc1plus: some warnings being treated as errors
```

6.9.2. stackprotector

Adds the `-fstack-protector-strong --param ssp-buffer-size=4` compiler options. This adds safety checks against stack overwrites rendering many potential code injection attacks into aborting situations. In the best case this turns code injection vulnerabilities into denial of service or into non-issues (depending on the application).

This needs to be turned off or fixed for errors similar to:

```
bin/blib.a(bios_console.o): In function `bios_handle_cup':
/tmp/nix-build-ipxe-20141124-5cbdc41.drv-0/ipxe-5cbdc41/src/arch/i386/firmw
```

6.9.3. fortify

Adds the `-O2 -D_FORTIFY_SOURCE=2` compiler options. During code generation the compiler knows a great deal of information about buffer sizes (where possible), and attempts to replace insecure unlimited length buffer function calls with length-limited ones. This is especially useful for old, crufty code. Additionally, format strings in writable memory that contain `%n` are blocked. If an application depends on such a format string, it will need to be worked around.

Additionally, some warnings are enabled which might trigger build failures if compiler warnings are treated as errors in the package build. In this case, set `NIX_CFLAGS_COMPILE` to `-Wno-error=warning-type`.

This needs to be turned off or fixed for errors similar to:

```
malloc.c:404:15: error: return type is an incomplete type
malloc.c:410:19: error: storage size of 'ms' isn't known

strdup.h:22:1: error: expected identifier or '(' before '__extension__'

strsep.c:65:23: error: register name not specified for 'delim'

installwatch.c:3751:5: error: conflicting types for '__open_2'

fcntl2.h:50:4: error: call to '__open_missing_mode' declared with attribute
```

6.9.4. pic

Adds the `-fPIC` compiler options. This options adds support for position independent code in shared libraries and thus making ASLR possible.

Most notably, the Linux kernel, kernel modules and other code not running in an operating system environment like boot loaders won't build with PIC enabled. The compiler will is most cases complain that PIC is not supported for a specific build.

This needs to be turned off or fixed for assembler errors similar to:

```
ccbLfRgg.s: Assembler messages:
ccbLfRgg.s:33: Error: missing or invalid displacement expression `private_k
```

6.9.5. `strictoverflow`

Signed integer overflow is undefined behaviour according to the C standard. If it happens, it is an error in the program as it should check for overflow before it can happen, not afterwards. GCC provides built-in functions to perform arithmetic with overflow checking, which are correct and faster than any custom implementation. As a workaround, the option `-fno-strict-overflow` makes gcc behave as if signed integer overflows were defined.

This flag should not trigger any build or runtime errors.

6.9.6. `relro`

Adds the `-z relro` linker option. During program load, several ELF memory sections need to be written to by the linker, but can be turned read-only before turning over control to the program. This prevents some GOT (and .dtors) overwrite attacks, but at least the part of the GOT used by the dynamic linker (.got.plt) is still vulnerable.

This flag can break dynamic shared object loading. For instance, the module systems of Xorg and OpenCV are incompatible with this flag. In almost all cases the `bindnow` flag must also be disabled and incompatible programs typically fail with similar errors at runtime.

6.9.7. `bindnow`

Adds the `-z bindnow` linker option. During program load, all dynamic symbols are resolved, allowing for the complete GOT to be marked read-only (due to `relro`). This prevents GOT overwrite attacks. For very large applications, this can incur some performance loss during initial load while symbols are resolved, but this shouldn't be an issue for daemons.

This flag can break dynamic shared object loading. For instance, the module systems of Xorg and PHP are incompatible with this flag. Programs incompatible with this flag often fail at runtime due to missing symbols, like:

```
intel_drv.so: undefined symbol: vgaHWFreeHWRec
```

The following flags are disabled by default and should be enabled with `hardeningEnable` for packages that take untrusted input like network services.

6.9.8. pie

Adds the `-fPIE` compiler and `-pie` linker options. Position Independent Executables are needed to take advantage of Address Space Layout Randomization, supported by modern kernel versions. While ASLR can already be enforced for data areas in the stack and heap (brk and mmap), the code areas must be compiled as position-independent. Shared libraries already do this with the `pic` flag, so they gain ASLR automatically, but binary .text regions need to be build with `pie` to gain ASLR. When this happens, ROP attacks are much harder since there are no static locations to bounce off of during a memory corruption attack.

For more in-depth information on these hardening flags and hardening in general, refer to the Debian Wiki, Ubuntu Wiki, Gentoo Wiki, and the Arch Wiki.

---

[1] The build platform is ignored because it is a mere implementation detail of the package satisfying the dependency: As a general programming principle, dependencies are always *specified* as interfaces, not concrete implementation.

[2] Currently, this means for native builds all dependencies are put on the `PATH`. But in the future that may not be the case for sake of matching cross: the platforms would be assumed to be unique for native and cross builds alike, so only the `depsBuild*` and `nativeBuildInputs` would be added to the `PATH`.

[3] The `findInputs` function, currently residing in `pkgs/stdenv/generic/setup.sh`, implements the propagation logic.

[4] It clears the `sys_lib_*search_path` variables in the Libtool script to prevent Libtool from using libraries in `/usr/lib` and such.

[5] Eventually these will be passed building natively as well, to improve determinism: build-time guessing, as is done today, is a risk of impurity.

[6] Each wrapper targets a single platform, so if binaries for multiple platforms are needed, the underlying binaries must be wrapped multiple times. As this is a property of the wrapper itself, the multiple wrappings are needed whether or not the same underlying binaries can target multiple platforms.

Chapter 7. Meta-attributes

Table of Contents

7.1. Standard meta-attributes

7.2. Licenses

Nix packages can declare *meta-attributes* that contain information about a package such as a description, its homepage, its license, and so on. For instance, the GNU Hello package has a `meta` declaration like this:

```
meta = with lib; {
  description = "A program that produces a familiar, friendly greeting";
  longDescription = ''
    GNU Hello is a program that prints "Hello, world!" when you run it.
    It is fully customizable.
  '';
  homepage = "https://www.gnu.org/software/hello/manual/";
  license = licenses.gpl3Plus;
  maintainers = [ maintainers.eelco ];
  platforms = platforms.all;
};
```

Meta-attributes are not passed to the builder of the package. Thus, a change to a meta-attribute doesn't trigger a recompilation of the package. The value of a meta-attribute must be a string.

The meta-attributes of a package can be queried from the command-line using `nix-env`:

```
$ nix-env -qa hello --json
{
    "hello": {
        "meta": {
            "description": "A program that produces a familiar, friendly gr
            "homepage": "https://www.gnu.org/software/hello/manual/",
            "license": {
                "fullName": "GNU General Public License version 3 or later"
                "shortName": "GPLv3+",
                "url": "http://www.fsf.org/licensing/licenses/gpl.html"
            },
            "longDescription": "GNU Hello is a program that prints \"Hello,
            "maintainers": [
                "Ludovic Court\u00e8s <ludo@gnu.org>"
            ],
            "platforms": [
                "i686-linux",
                "x86_64-linux",
                "armv5tel-linux",
                "armv7l-linux",
                "mips32-linux",
                "x86_64-darwin",
                "i686-cygwin",
                "i686-freebsd",
                "x86_64-freebsd",
                "i686-openbsd",
```

```
                "x86_64-openbsd"
            ],
            "position": "/home/user/dev/nixpkgs/pkgs/applications/misc/hell
        },
        "name": "hello-2.9",
        "system": "x86_64-linux"
    }
}
```

`nix-env` knows about the `description` field specifically:

```
$ nix-env -qa hello --description
hello-2.3  A program that produces a familiar, friendly greeting
```

7.1. Standard meta-attributes

It is expected that each meta-attribute is one of the following:

7.1.1. `description`

A short (one-line) description of the package. This is shown by `nix-env -q --description` and also on the Nixpkgs release pages.

Don't include a period at the end. Don't include newline characters. Capitalise the first character. For brevity, don't repeat the name of package — just describe what it does.

Wrong: `"libpng is a library that allows you to decode PNG images."`

Right: `"A library for decoding PNG images"`

7.1.2. `longDescription`

An arbitrarily long description of the package.

7.1.3. `branch`

Release branch. Used to specify that a package is not going to receive updates that are not in this branch; for example, Linux kernel 3.0 is supposed to be updated to 3.0.X, not 3.1.

7.1.4. `homepage`

The package's homepage. Example: `https://www.gnu.org/software/hello/manual/`

7.1.5. downloadPage

The page where a link to the current version can be found. Example: `https://ftp.gnu.org/gnu/hello/`

7.1.6. changelog

A link or a list of links to the location of Changelog for a package. A link may use expansion to refer to the correct changelog version. Example: `"https://git.savannah.gnu.org/cgit/hello.git/plain/NEWS?h=v${version}"`

7.1.7. license

The license, or licenses, for the package. One from the attribute set defined in `nixpkgs/lib/licenses.nix`. At this moment using both a list of licenses and a single license is valid. If the license field is in the form of a list representation, then it means that parts of the package are licensed differently. Each license should preferably be referenced by their attribute. The non-list attribute value can also be a space delimited string representation of the contained attribute `shortNames` or `spdxIds`. The following are all valid examples:

- Single license referenced by attribute (preferred) `lib.licenses.gpl3Only`.
- Single license referenced by its attribute shortName (frowned upon) `"gpl3Only"`.
- Single license referenced by its attribute spdxId (frowned upon) `"GPL-3.0-only"`.
- Multiple licenses referenced by attribute (preferred) `with lib.licenses; [ asl20 free ofl ]`.
- Multiple licenses referenced as a space delimited string of attribute shortNames (frowned upon) `"asl20 free ofl"`.

For details, see Licenses.

7.1.8. maintainers

A list of the maintainers of this Nix expression. Maintainers are defined in `nixpkgs/maintainers/maintainer-list.nix`. There is no restriction to becoming a maintainer, just add yourself to that list in a separate commit titled “maintainers: add alice”, and reference maintainers with `maintainers = with lib.maintainers; [ alice bob ]`.

7.1.9. priority

The *priority* of the package, used by `nix-env` to resolve file name conflicts between packages. See the Nix manual page for `nix-env` for details. Example: `"10"` (a low-priority package).

7.1.10. platforms

The list of Nix platform types on which the package is supported. Hydra builds packages according to the platform specified. If no platform is specified, the package does not have prebuilt binaries. An example is:

```
meta.platforms = lib.platforms.linux;
```

Attribute Set `lib.platforms` defines various common lists of platforms types.

7.1.11. `tests`

> **Warning:** This attribute is special in that it is not actually under the `meta` attribute set but rather under the `passthru` attribute set. This is due to how `meta` attributes work, and the fact that they are supposed to contain only metadata, not derivations.

An attribute set with as values tests. A test is a derivation, which builds successfully when the test passes, and fails to build otherwise. A derivation that is a test needs to have `meta.timeout` defined.

The NixOS tests are available as `nixosTests` in parameters of derivations. For instance, the OpenSMTPD derivation includes lines similar to:

```
{ /* ... */, nixosTests }:
{
  # ...
  passthru.tests = {
    basic-functionality-and-dovecot-integration = nixosTests.opensmtpd;
  };
}
```

7.1.12. `timeout`

A timeout (in seconds) for building the derivation. If the derivation takes longer than this time to build, it can fail due to breaking the timeout. However, all computers do not have the same computing power, hence some builders may decide to apply a multiplicative factor to this value. When filling this value in, try to keep it approximately consistent with other values already present in `nixpkgs`.

7.1.13. `hydraPlatforms`

The list of Nix platform types for which the Hydra instance at `hydra.nixos.org` will build the package. (Hydra is the Nix-based continuous build system.) It defaults to the value of `meta.platforms`. Thus, the only reason to set `meta.hydraPlatforms` is if you want `hydra.nixos.org` to build the package on a subset of `meta.platforms`, or not at all, e.g.

```
meta.platforms = lib.platforms.linux;
meta.hydraPlatforms = [];
```

7.1.14. broken

If set to `true`, the package is marked as "broken", meaning that it won't show up in `nix-env -qa`, and cannot be built or installed. Such packages should be removed from Nixpkgs eventually unless they are fixed.

7.1.15. updateWalker

If set to `true`, the package is tested to be updated correctly by the `update-walker.sh` script without additional settings. Such packages have `meta.version` set and their homepage (or the page specified by `meta.downloadPage`) contains a direct link to the package tarball.

7.2. Licenses

The `meta.license` attribute should preferably contain a value from `lib.licenses` defined in `nixpkgs/lib/licenses.nix`, or in-place license description of the same format if the license is unlikely to be useful in another expression.

Although it's typically better to indicate the specific license, a few generic options are available:

7.2.1. lib.licenses.free, "free"

Catch-all for free software licenses not listed above.

7.2.2. lib.licenses.unfreeRedistributable, "unfree-redistributable"

Unfree package that can be redistributed in binary form. That is, it's legal to redistribute the *output* of the derivation. This means that the package can be included in the Nixpkgs channel.

Sometimes proprietary software can only be redistributed unmodified. Make sure the builder doesn't actually modify the original binaries; otherwise we're breaking the license. For instance, the NVIDIA X11 drivers can be redistributed unmodified, but our builder applies `patchelf` to make them work. Thus, its license is `"unfree"` and it cannot be included in the Nixpkgs channel.

7.2.3. lib.licenses.unfree, "unfree"

Unfree package that cannot be redistributed. You can build it yourself, but you cannot redistribute the output of the derivation. Thus it cannot be included in the Nixpkgs channel.

7.2.4. lib.licenses.unfreeRedistributableFirmware,

"unfree-redistributable-firmware"

This package supplies unfree, redistributable firmware. This is a separate value from `unfree-redistributable` because not everybody cares whether firmware is free.

Chapter 8. Multiple-output packages

Table of Contents

8.1. Introduction

8.2. Installing a split package

8.3. Using a split package

8.4. Writing a split derivation

8.1. Introduction

The Nix language allows a derivation to produce multiple outputs, which is similar to what is utilized by other Linux distribution packaging systems. The outputs reside in separate Nix store paths, so they can be mostly handled independently of each other, including passing to build inputs, garbage collection or binary substitution. The exception is that building from source always produces all the outputs.

The main motivation is to save disk space by reducing runtime closure sizes; consequently also sizes of substituted binaries get reduced. Splitting can be used to have more granular runtime dependencies, for example the typical reduction is to split away development-only files, as those are typically not needed during runtime. As a result, closure sizes of many packages can get reduced to a half or even much less.

> **Note:** The reduction effects could be instead achieved by building the parts in completely separate derivations. That would often additionally reduce build-time closures, but it tends to be much harder to write such derivations, as build systems typically assume all parts are being built at once. This compromise approach of single source package producing multiple binary packages is also utilized often by rpm and deb.

A number of attributes can be used to work with a derivation with multiple outputs. The attribute `outputs` is a list of strings, which are the names of the outputs. For each of these names, an identically named attribute is created, corresponding to that output. The attribute `meta.outputsToInstall` is used to determine the default set of outputs to install when using the derivation name unqualified.

8.2. Installing a split package

When installing a package with multiple outputs, the package's `meta.outputsToInstall` attribute determines which outputs are actually installed. `meta.outputsToInstall` is a list whose default

installs binaries and the associated man pages. The following sections describe ways to install different outputs.

8.2.1. Selecting outputs to install via NixOS

NixOS provides two ways to select the outputs to install for packages listed in `environment.systemPackages`:

- The configuration option `environment.extraOutputsToInstall` is appended to each package's `meta.outputsToInstall` attribute to determine the outputs to install. It can for example be used to install `info` documentation or debug symbols for all packages.
- The outputs can be listed as packages in `environment.systemPackages`. For example, the `"out"` and `"info"` outputs for the `coreutils` package can be installed by including `coreutils` and `coreutils.info` in `environment.systemPackages`.

8.2.2. Selecting outputs to install via `nix-env`

`nix-env` lacks an easy way to select the outputs to install. When installing a package, `nix-env` always installs the outputs listed in `meta.outputsToInstall`, even when the user explicitly selects an output.

`nix-env` silenty disregards the outputs selected by the user, and instead installs the outputs from `meta.outputsToInstall`. For example,

```
$ nix-env -iA nixpkgs.coreutils.info
```

installs the `"out"` output (`coreutils.meta.outputsToInstall` is `[ "out" ]`) instead of the requested `"info"`.

The only recourse to select an output with `nix-env` is to override the package's `meta.outputsToInstall`, using the functions described in Chapter 4, *Overriding*. For example, the following overlay adds the `"info"` output for the `coreutils` package:

```
self: super:
{
  coreutils = super.coreutils.overrideAttrs (oldAttrs: {
    meta = oldAttrs.meta // { outputsToInstall = oldAttrs.meta.outputsToIns
  });
}
```

8.3. Using a split package

In the Nix language the individual outputs can be reached explicitly as attributes, e.g. `coreutils.info`, but the typical case is just using packages as build inputs.

When a multiple-output derivation gets into a build input of another derivation, the `dev` output is added if it exists, otherwise the first output is added. In addition to that, `propagatedBuildOutputs` of that package which by default contain `$outputBin` and `$outputLib` are also added. (See Section 8.4.2, "File type groups".)

In some cases it may be desirable to combine different outputs under a single store path. A function `symlinkJoin` can be used to do this. (Note that it may negate some closure size benefits of using a multiple-output package.)

8.4. Writing a split derivation

Here you find how to write a derivation that produces multiple outputs.

In nixpkgs there is a framework supporting multiple-output derivations. It tries to cover most cases by default behavior. You can find the source separated in `<nixpkgs/pkgs/build-support/setup-hooks/multiple-outputs.sh>`; it's relatively well-readable. The whole machinery is triggered by defining the `outputs` attribute to contain the list of desired output names (strings).

```
outputs = [ "bin" "dev" "out" "doc" ];
```

Often such a single line is enough. For each output an equally named environment variable is passed to the builder and contains the path in nix store for that output. Typically you also want to have the main `out` output, as it catches any files that didn't get elsewhere.

> **Note:** There is a special handling of the `debug` output, described at Section 6.5.8.1.17, "`separateDebugInfo`".

8.4.1. "Binaries first"

A commonly adopted convention in `nixpkgs` is that executables provided by the package are contained within its first output. This convention allows the dependent packages to reference the executables provided by packages in a uniform manner. For instance, provided with the knowledge that the `perl` package contains a `perl` executable it can be referenced as `${pkgs.perl}/bin/perl` within a Nix derivation that needs to execute a Perl script.

The `glibc` package is a deliberate single exception to the "binaries first" convention. The `glibc` has `libs` as its first output allowing the libraries provided by `glibc` to be referenced directly (e.g. `${stdenv.glibc}/lib/ld-linux-x86-64.so.2`). The executables provided by `glibc` can be accessed via its `bin` attribute (e.g. `${stdenv.glibc.bin}/bin/ldd`).

The reason for why `glibc` deviates from the convention is because referencing a library provided by

`glibc` is a very common operation among Nix packages. For instance, third-party executables packaged by Nix are typically patched and relinked with the relevant version of `glibc` libraries from Nix packages (please see the documentation on patchelf for more details).

8.4.2. File type groups

The support code currently recognizes some particular kinds of outputs and either instructs the build system of the package to put files into their desired outputs or it moves the files during the fixup phase. Each group of file types has an `outputFoo` variable specifying the output name where they should go. If that variable isn't defined by the derivation writer, it is guessed – a default output name is defined, falling back to other possibilities if the output isn't defined.

8.4.2.1. `$outputDev`

is for development-only files. These include C(++) headers (`include/`), pkg-config (`lib/pkgconfig/`), cmake (`lib/cmake/`) and aclocal files (`share/aclocal/`). They go to `dev` or `out` by default.

8.4.2.2. `$outputBin`

is meant for user-facing binaries, typically residing in `bin/`. They go to `bin` or `out` by default.

8.4.2.3. `$outputLib`

is meant for libraries, typically residing in `lib/` and `libexec/`. They go to `lib` or `out` by default.

8.4.2.4. `$outputDoc`

is for user documentation, typically residing in `share/doc/`. It goes to `doc` or `out` by default.

8.4.2.5. `$outputDevdoc`

is for *developer* documentation. Currently we count gtk-doc and devhelp books, typically residing in `share/gtk-doc/` and `share/devhelp/`, in there. It goes to `devdoc` or is removed (!) by default. This is because e.g. gtk-doc tends to be rather large and completely unused by nixpkgs users.

8.4.2.6. `$outputMan`

is for man pages (except for section 3), typically residing in `share/man/man[0-9]/`. They go to `man` or `$outputBin` by default.

8.4.2.7. `$outputDevman`

is for section 3 man pages, typically residing in `share/man/man[0-9]/`. They go to `devman` or `$outputMan` by default.

8.4.2.8. `$outputInfo`

is for info pages, typically residing in `share/info/`. They go to `info` or `$outputBin` by default.

8.4.3. Common caveats

- Some configure scripts don't like some of the parameters passed by default by the framework, e.g. `--docdir=/foo/bar`. You can disable this by setting `setOutputFlags = false;`.
- The outputs of a single derivation can retain references to each other, but note that circular references are not allowed. (And each strongly-connected component would act as a single output anyway.)
- Most of split packages contain their core functionality in libraries. These libraries tend to refer to various kind of data that typically gets into `out`, e.g. locale strings, so there is often no advantage in separating the libraries into `lib`, as keeping them in `out` is easier.
- Some packages have hidden assumptions on install paths, which complicates splitting.

Chapter 9. Cross-compilation

Table of Contents

9.1. Introduction

9.2. Packaging in a cross-friendly manner

9.3. Cross-building packages

9.4. Cross-compilation infrastructure

9.1. Introduction

"Cross-compilation" means compiling a program on one machine for another type of machine. For example, a typical use of cross-compilation is to compile programs for embedded devices. These devices often don't have the computing power and memory to compile their own programs. One might think that cross-compilation is a fairly niche concern. However, there are significant advantages to rigorously distinguishing between build-time and run-time environments! Significant, because the benefits apply even when one is developing and deploying on the same machine. Nixpkgs is increasingly adopting the opinion that packages should be written with cross-compilation in mind, and Nixpkgs should evaluate in a similar way (by minimizing cross-compilation-specific special cases) whether or not one is cross-compiling.

This chapter will be organized in three parts. First, it will describe the basics of how to package software in a way that supports cross-compilation. Second, it will describe how to use Nixpkgs when cross-compiling. Third, it will describe the internal infrastructure supporting cross-compilation.

9.2. Packaging in a cross-friendly manner

9.2.1. Platform parameters

Nixpkgs follows the conventions of GNU autoconf. We distinguish between 3 types of platforms when building a derivation: *build*, *host*, and *target*. In summary, *build* is the platform on which a package is being built, *host* is the platform on which it will run. The third attribute, *target*, is relevant only for certain specific compilers and build tools.

In Nixpkgs, these three platforms are defined as attribute sets under the names `buildPlatform`, `hostPlatform`, and `targetPlatform`. They are always defined as attributes in the standard environment. That means one can access them like:

```
{ stdenv, fooDep, barDep, ... }: ...stdenv.buildPlatform...
```

`buildPlatform`

The "build platform" is the platform on which a package is built. Once someone has a built package, or pre-built binary package, the build platform should not matter and can be ignored.

`hostPlatform`

The "host platform" is the platform on which a package will be run. This is the simplest platform to understand, but also the one with the worst name.

`targetPlatform`

The "target platform" attribute is, unlike the other two attributes, not actually fundamental to the process of building software. Instead, it is only relevant for compatibility with building certain specific compilers and build tools. It can be safely ignored for all other packages.

The build process of certain compilers is written in such a way that the compiler resulting from a single build can itself only produce binaries for a single platform. The task of specifying this single "target platform" is thus pushed to build time of the compiler. The root cause of this is that the compiler (which will be run on the host) and the standard library/runtime (which will be run on the target) are built by a single build process.

There is no fundamental need to think about a single target ahead of time like this. If the tool supports modular or pluggable backends, both the need to specify the target at build time and the constraint of having only a single target disappear. An example of such a tool is LLVM.

Although the existence of a "target platform" is arguably a historical mistake, it is a common one: examples of tools that suffer from it are GCC, Binutils, GHC and Autoconf. Nixpkgs tries to avoid sharing in the mistake where possible. Still, because the concept of a target platform is so ingrained, it is best to support it as is.

The exact schema these fields follow is a bit ill-defined due to a long and convoluted evolution, but this is slowly being cleaned up. You can see examples of ones used in practice in `lib.systems.examples`; note how they are not all very consistent. For now, here are few fields can count on them containing:

`system`

This is a two-component shorthand for the platform. Examples of this would be "x86_64-darwin" and "i686-linux"; see `lib.systems.doubles` for more. The first component corresponds to the CPU architecture of the platform and the second to the operating system of the platform (`[cpu]-[os]`). This format has built-in support in Nix, such as the `builtins.currentSystem` impure string.

`config`

This is a 3- or 4- component shorthand for the platform. Examples of this would be `x86_64-unknown-linux-gnu` and `aarch64-apple-darwin14`. This is a standard format called the "LLVM target triple", as they are pioneered by LLVM. In the 4-part form, this corresponds to `[cpu]-[vendor]-[os]-[abi]`. This format is strictly more informative than the "Nix host double", as the previous format could analogously be termed. This needs a better name than `config`!

`parsed`

This is a Nix representation of a parsed LLVM target triple with white-listed components. This can be specified directly, or actually parsed from the `config`. See `lib.systems.parse` for the exact representation.

`libc`

This is a string identifying the standard C library used. Valid identifiers include "glibc" for GNU libc, "libSystem" for Darwin's Libsystem, and "uclibc" for µClibc. It should probably be refactored to use the module system, like `parse`.

`is*`

These predicates are defined in `lib.systems.inspect`, and slapped onto every platform. They are superior to the ones in `stdenv` as they force the user to be explicit about which platform they are inspecting. Please use these instead of those.

`platform`

This is, quite frankly, a dumping ground of ad-hoc settings (it's an attribute set). See `lib.systems.platforms` for examples—there's hopefully one in there that will work verbatim for each platform that is working. Please help us triage these flags and give them better homes!

9.2.2. Theory of dependency categorization

Note: This is a rather philosophical description that isn't very Nixpkgs-specific. For an overview of all the relevant attributes given to `mkDerivation`, see Section 6.3, "Specifying dependencies". For a description of how everything is implemented, see Section 9.4.1, "Implementation of dependencies".

In this section we explore the relationship between both runtime and build-time dependencies and the 3 Autoconf platforms.

A run time dependency between two packages requires that their host platforms match. This is directly implied by the meaning of "host platform" and "runtime dependency": The package dependency exists while both packages are running on a single host platform.

A build time dependency, however, has a shift in platforms between the depending package and the depended-on package. "build time dependency" means that to build the depending package we need to be able to run the depended-on's package. The depending package's build platform is therefore equal to the depended-on package's host platform.

If both the dependency and depending packages aren't compilers or other machine-code-producing tools, we're done. And indeed `buildInputs` and `nativeBuildInputs` have covered these simpler cases for many years. But if the dependency does produce machine code, we might need to worry about its target platform too. In principle, that target platform might be any of the depending package's build, host, or target platforms, but we prohibit dependencies from a "later" platform to an earlier platform to limit confusion because we've never seen a legitimate use for them.

Finally, if the depending package is a compiler or other machine-code-producing tool, it might need dependencies that run at "emit time". This is for compilers that (regrettably) insist on being built together with their source languages' standard libraries. Assuming build != host != target, a run-time dependency of the standard library cannot be run at the compiler's build time or run time, but only at the run time of code emitted by the compiler.

Putting this all together, that means we have dependencies in the form "host → target", in at most the following six combinations:

9.2.2.1. Possible dependency types

Dependency's host platform	Dependency's target platform
build	build
build	host
build	target
host	host
host	target
target	target

Some examples will make this table clearer. Suppose there's some package that is being built with a `(build, host, target)` platform triple of `(foo, bar, baz)`. If it has a build-time library dependency, that would be a "host → build" dependency with a triple of `(foo, foo, *)` (the target platform is irrelevant). If it needs a compiler to be built, that would be a "build → host" dependency with a triple of `(foo, foo, *)` (the target platform is irrelevant). That compiler, would be built with another compiler, also "build → host" dependency, with a triple of `(foo, foo, foo)`.

9.2.3. Cross packaging cookbook

Some frequently encountered problems when packaging for cross-compilation should be answered here. Ideally, the information above is exhaustive, so this section cannot provide any new information, but it is ludicrous and cruel to expect everyone to spend effort working through the interaction of many features just to figure out the same answer to the same common problem. Feel free to add to this list!

9.2.3.1. My package fails to find a binutils command (`cc/ar/ld` etc.)

Many packages assume that an unprefixed binutils (`cc/ar/ld` etc.) is available, but Nix doesn't provide one. It only provides a prefixed one, just as it only does for all the other binutils programs. It may be necessary to patch the package to fix the build system to use a prefix. For instance, instead of `cc`, use `${stdenv.cc.targetPrefix}cc`.

```
makeFlags = [ "CC=${stdenv.cc.targetPrefix}cc" ];
```

9.2.3.2. How do I avoid compiling a GCC cross-compiler from source?

On less powerful machines, it can be inconvenient to cross-compile a package only to find out that GCC has to be compiled from source, which could take up to several hours. Nixpkgs maintains a limited cross-related jobset on Hydra, which tests cross-compilation to various platforms from build platforms "x86_64-darwin", "x86_64-linux", and "aarch64-linux". See `pkgs/top-level/release-cross.nix` for the full list of target platforms and packages. For instance, the following invocation fetches the pre-built cross-compiled GCC for `armv6l-unknown-linux-gnueabihf` and builds GNU Hello from source.

```
$ nix-build '<nixpkgs>' -A pkgsCross.raspberryPi.hello
```

9.2.3.3. What if my package's build system needs to build a C program to be run under the build environment?

Add the following to your `mkDerivation` invocation.

```
depsBuildBuild = [ buildPackages.stdenv.cc ];
```

9.2.3.4. My package's testsuite needs to run host platform code.

Add the following to your `mkDerivation` invocation.

```
doCheck = stdenv.hostPlatform == stdenv.buildPlatform;
```

9.3. Cross-building packages

Nixpkgs can be instantiated with `localSystem` alone, in which case there is no cross-compiling and

everything is built by and for that system, or also with `crossSystem`, in which case packages run on the latter, but all building happens on the former. Both parameters take the same schema as the 3 (build, host, and target) platforms defined in the previous section. As mentioned above, `lib.systems.examples` has some platforms which are used as arguments for these parameters in practice. You can use them programmatically, or on the command line:

```
$ nix-build '<nixpkgs>' --arg crossSystem '(import <nixpkgs/lib>).systems.e
```

Note

Eventually we would like to make these platform examples an unnecessary convenience so that

```
$ nix-build '<nixpkgs>' --arg crossSystem '{ config = "<arch>-<os>-<vendor>
```

works in the vast majority of cases. The problem today is dependencies on other sorts of configuration which aren't given proper defaults. We rely on the examples to crudely to set those configuration parameters in some vaguely sane manner on the users behalf. Issue #34274 tracks this inconvenience along with its root cause in crufty configuration options.

While one is free to pass both parameters in full, there's a lot of logic to fill in missing fields. As discussed in the previous section, only one of `system`, `config`, and `parsed` is needed to infer the other two. Additionally, `libc` will be inferred from `parse`. Finally, `localSystem.system` is also *impurely* inferred based on the platform evaluation occurs. This means it is often not necessary to pass `localSystem` at all, as in the command-line example in the previous paragraph.

> **Note:** Many sources (manual, wiki, etc) probably mention passing `system`, `platform`, along with the optional `crossSystem` to Nixpkgs:
> `import <nixpkgs> { system = ..; platform = ..; crossSystem = ..; }`.
> Passing those two instead of `localSystem` is still supported for compatibility, but is discouraged. Indeed, much of the inference we do for these parameters is motivated by compatibility as much as convenience.

One would think that `localSystem` and `crossSystem` overlap horribly with the three `*Platforms` (`buildPlatform`, `hostPlatform`, and `targetPlatform`; see `stage.nix` or the manual). Actually, those identifiers are purposefully not used here to draw a subtle but important distinction: While the granularity of having 3 platforms is necessary to properly *build* packages, it is overkill for specifying the user's *intent* when making a build plan or package set. A simple "build vs deploy" dichotomy is adequate: the sliding window principle described in the previous section shows how to interpolate between the these two "end points" to get the 3 platform triple for each bootstrapping stage. That means for any package a given package set, even those not bound on the top level but only reachable via dependencies or `buildPackages`, the three platforms will be defined as one of `localSystem` or `crossSystem`, with

the former replacing the latter as one traverses build-time dependencies. A last simple difference is that `crossSystem` should be null when one doesn't want to cross-compile, while the `*Platform`s are always non-null. `localSystem` is always non-null.

9.4. Cross-compilation infrastructure

9.4.1. Implementation of dependencies

The categories of dependencies developed in Section 9.2.2, "Theory of dependency categorization" are specified as lists of derivations given to `mkDerivation`, as documented in Section 6.3, "Specifying dependencies". In short, each list of dependencies for "host → target" of "foo → bar" is called `depsFooBar`, with exceptions for backwards compatibility that `depsBuildHost` is instead called `nativeBuildInputs` and `depsHostTarget` is instead called `buildInputs`. Nixpkgs is now structured so that each `depsFooBar` is automatically taken from `pkgsFooBar`. (These `pkgsFooBar`s are quite new, so there is no special case for `nativeBuildInputs` and `buildInputs`.) For example, `pkgsBuildHost.gcc` should be used at build-time, while `pkgsHostTarget.gcc` should be used at run-time.

Now, for most of Nixpkgs's history, there were no `pkgsFooBar` attributes, and most packages have not been refactored to use it explicitly. Prior to those, there were just `buildPackages`, `pkgs`, and `targetPackages`. Those are now redefined as aliases to `pkgsBuildHost`, `pkgsHostTarget`, and `pkgsTargetTarget`. It is acceptable, even recommended, to use them for libraries to show that the host platform is irrelevant.

But before that, there was just `pkgs`, even though both `buildInputs` and `nativeBuildInputs` existed. [Cross barely worked, and those were implemented with some hacks on `mkDerivation` to override dependencies.] What this means is the vast majority of packages do not use any explicit package set to populate their dependencies, just using whatever `callPackage` gives them even if they do correctly sort their dependencies into the multiple lists described above. And indeed, asking that users both sort their dependencies, *and* take them from the right attribute set, is both too onerous and redundant, so the recommended approach (for now) is to continue just categorizing by list and not using an explicit package set.

To make this work, we "splice" together the six `pkgsFooBar` package sets and have `callPackage` actually take its arguments from that. This is currently implemented in `pkgs/top-level/splice.nix`. `mkDerivation` then, for each dependency attribute, pulls the right derivation out from the splice. This splicing can be skipped when not cross-compiling as the package sets are the same, but still is a bit slow for cross-compiling. We'd like to do something better, but haven't come up with anything yet.

9.4.2. Bootstrapping

Each of the package sets described above come from a single bootstrapping stage. While `pkgs/top-level/default.nix`, coordinates the composition of stages at a high level, `pkgs/top-level/stage.nix` "ties the knot" (creates the fixed point) of each stage. The package sets are defined per-stage however, so they can be thought of as edges between stages (the nodes) in a graph.

Compositions like `pkgsBuildTarget.targetPackages` can be thought of as paths to this graph.

While there are many package sets, and thus many edges, the stages can also be arranged in a linear chain. In other words, many of the edges are redundant as far as connectivity is concerned. This hinges on the type of bootstrapping we do. Currently for cross it is:

1. `(native, native, native)`
2. `(native, native, foreign)`
3. `(native, foreign, foreign)`

In each stage, `pkgsBuildHost` refers to the previous stage, `pkgsBuildBuild` refers to the one before that, and `pkgsHostTarget` refers to the current one, and `pkgsTargetTarget` refers to the next one. When there is no previous or next stage, they instead refer to the current stage. Note how all the invariants regarding the mapping between dependency and depending packages' build host and target platforms are preserved. `pkgsBuildTarget` and `pkgsHostHost` are more complex in that the stage fitting the requirements isn't always a fixed chain of "prevs" and "nexts" away (modulo the "saturating" self-references at the ends). We just special case each instead. All the primary edges are implemented is in `pkgs/stdenv/booter.nix`, and secondarily aliases in `pkgs/top-level/stage.nix`.

> **Note:** The native stages are bootstrapped in legacy ways that predate the current cross implementation. This is why the bootstrapping stages leading up to the final stages are ignored in the previous paragraph.

If one looks at the 3 platform triples, one can see that they overlap such that one could put them together into a chain like:

```
(native, native, native, foreign, foreign)
```

If one imagines the saturating self references at the end being replaced with infinite stages, and then overlays those platform triples, one ends up with the infinite tuple:

```
(native..., native, native, native, foreign, foreign, foreign...)
```

One can then imagine any sequence of platforms such that there are bootstrap stages with their 3 platforms determined by "sliding a window" that is the 3 tuple through the sequence. This was the original model for bootstrapping. Without a target platform (assume a better world where all compilers are multi-target and all standard libraries are built in their own derivation), this is sufficient. Conversely if one wishes to cross compile "faster", with a "Canadian Cross" bootstrapping stage where `build != host != target`, more bootstrapping stages are needed since no sliding window provides the pesky `pkgsBuildTarget` package set since it skips the Canadian cross stage's "host".

Note

It is much better to refer to `buildPackages` than `targetPackages`, or more broadly package sets that do not mention "target". There are three reasons for this.

First, it is because bootstrapping stages do not have a unique `targetPackages`. For example a `(x86-linux, x86-linux, arm-linux)` and `(x86-linux, x86-linux, x86-windows)` package set both have a `(x86-linux, x86-linux, x86-linux)` package set. Because there is no canonical `targetPackages` for such a native (`build == host == target`) package set, we set their `targetPackages`

Second, it is because this is a frequent source of hard-to-follow "infinite recursions" / cycles. When only package sets that don't mention target are used, the package set forms a directed acyclic graph. This means that all cycles that exist are confined to one stage. This means they are a lot smaller, and easier to follow in the code or a backtrace. It also means they are present in native and cross builds alike, and so more likely to be caught by CI and other users.

Thirdly, it is because everything target-mentioning only exists to accommodate compilers with lousy build systems that insist on the compiler itself and standard library being built together. Of course that is bad because bigger derivations means longer rebuilds. It is also problematic because it tends to make the standard libraries less like other libraries than they could be, complicating code and build systems alike. Because of the other problems, and because of these innate disadvantages, compilers ought to be packaged another way where possible.

> **Note:** If one explores Nixpkgs, they will see derivations with names like `gccCross`. Such `*Cross` derivations is a holdover from before we properly distinguished between the host and target platforms—the derivation with "Cross" in the name covered the `build = host != target` case, while the other covered the `host = target`, with build platform the same or not based on whether one was using its `.nativeDrv` or `.crossDrv`. This ugliness will disappear soon.

Chapter 10. Platform Notes

Table of Contents

10.1. Darwin (macOS)

10.1. Darwin (macOS)

Some common issues when packaging software for Darwin:

- The Darwin `stdenv` uses clang instead of gcc. When referring to the compiler `$CC` or `cc` will work in both cases. Some builds hardcode gcc/g++ in their build scripts, that can usually be fixed with using something like `makeFlags = [ "CC=cc" ];` or by patching the build scripts.

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
```

```
  # ...
  buildPhase = ''
    $CC -o hello hello.c
  '';
}
```

- On Darwin, libraries are linked using absolute paths, libraries are resolved by their `install_name` at link time. Sometimes packages won't set this correctly causing the library lookups to fail at runtime. This can be fixed by adding extra linker flags or by running `install_name_tool -id` during the `fixupPhase`.

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  # ...
  makeFlags = lib.optional stdenv.isDarwin "LDFLAGS=-Wl,-install_name,$(ou
}
```

- Even if the libraries are linked using absolute paths and resolved via their `install_name` correctly, tests can sometimes fail to run binaries. This happens because the `checkPhase` runs before the libraries are installed.

  This can usually be solved by running the tests after the `installPhase` or alternatively by using `DYLD_LIBRARY_PATH`. More information about this variable can be found in the *dyld(1)* manpage.

```
dyld: Library not loaded: /nix/store/7hnmbscpayxzxrixrgxvvlifzlxdsdir-jq-1
Referenced from: /private/tmp/nix-build-jq-1.5.drv-0/jq-1.5/tests/../jq
Reason: image not found
./tests/jqtest: line 5: 75779 Abort trap: 6
```

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  # ...
  doInstallCheck = true;
  installCheckTarget = "check";
}
```

- Some packages assume xcode is available and use `xcrun` to resolve build tools like `clang`, etc. This causes errors like
  `xcode-select: error: no developer tools were found at '/Applications/Xcode.app'`
  while the build doesn't actually depend on xcode.

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  # ...
  prePatch = ''
    substituteInPlace Makefile \
        --replace '/usr/bin/xcrun clang' clang
  '';
}
```

The package `xcbuild` can be used to build projects that really depend on Xcode. However, this replacement is not 100% compatible with Xcode and can occasionally cause issues.

Part III. Builders

Table of Contents

11. Fetchers

12. Trivial builders

13. Special builders

14. Images

15. Languages and frameworks

16. Packages

Chapter 11. Fetchers

Table of Contents

11.1. `fetchsvn`

11.2. `fetchgit`

11.3. `fetchfossil`

11.4. `fetchcvs`

11.5. `fetchhg`

11.6. `fetchFromGitHub`

11.7. `fetchFromGitLab`

11.8. fetchFromGitiles

11.9. fetchFromBitbucket

11.10. fetchFromSavannah

11.11. fetchFromRepoOrCz

11.12. fetchFromSourcehut

When using Nix, you will frequently need to download source code and other files from the internet. Nixpkgs comes with a few helper functions that allow you to fetch fixed-output derivations in a structured way.

The two fetcher primitives are `fetchurl` and `fetchzip`. Both of these have two required arguments, a URL and a hash. The hash is typically `sha256`, although many more hash algorithms are supported. Nixpkgs contributors are currently recommended to use `sha256`. This hash will be used by Nix to identify your source. A typical usage of fetchurl is provided below.

```
{ stdenv, fetchurl }:

stdenv.mkDerivation {
  name = "hello";
  src = fetchurl {
    url = "http://www.example.org/hello.tar.gz";
    sha256 = "1111111111111111111111111111111111111111111111111111";
  };
}
```

The main difference between `fetchurl` and `fetchzip` is in how they store the contents. `fetchurl` will store the unaltered contents of the URL within the Nix store. `fetchzip` on the other hand will decompress the archive for you, making files and directories directly accessible in the future. `fetchzip` can only be used with archives. Despite the name, `fetchzip` is not limited to .zip files and can also be used with any tarball.

`fetchpatch` works very similarly to `fetchurl` with the same arguments expected. It expects patch files as a source and performs normalization on them before computing the checksum. For example it will remove comments or other unstable parts that are sometimes added by version control systems and can change over time.

Other fetcher functions allow you to add source code directly from a VCS such as subversion or git. These are mostly straightforward nambes based on the name of the command used with the VCS system. Because they give you a working repository, they act most like `fetchzip`.

11.1. fetchsvn

Used with Subversion. Expects `url` to a Subversion directory, `rev`, and `sha256`.

11.2. fetchgit

Used with Git. Expects `url` to a Git repo, `rev`, and `sha256`. `rev` in this case can be full the git commit id (SHA1 hash) or a tag name like `refs/tags/v1.0`.

Additionally the following optional arguments can be given: `fetchSubmodules = true` makes `fetchgit` also fetch the submodules of a repository. If `deepClone` is set to true, the entire repository is cloned as opposing to just creating a shallow clone. `deepClone = true` also implies `leaveDotGit = true` which means that the `.git` directory of the clone won't be removed after checkout.

11.3. fetchfossil

Used with Fossil. Expects `url` to a Fossil archive, `rev`, and `sha256`.

11.4. fetchcvs

Used with CVS. Expects `cvsRoot`, `tag`, and `sha256`.

11.5. fetchhg

Used with Mercurial. Expects `url`, `rev`, and `sha256`.

A number of fetcher functions wrap part of `fetchurl` and `fetchzip`. They are mainly convenience functions intended for commonly used destinations of source code in Nixpkgs. These wrapper fetchers are listed below.

11.6. fetchFromGitHub

`fetchFromGitHub` expects four arguments. `owner` is a string corresponding to the GitHub user or organization that controls this repository. `repo` corresponds to the name of the software repository. These are located at the top of every GitHub HTML page as `owner/repo`. `rev` corresponds to the Git commit hash or tag (e.g `v1.0`) that will be downloaded from Git. Finally, `sha256` corresponds to the hash of the extracted directory. Again, other hash algorithms are also available but `sha256` is currently preferred.

`fetchFromGitHub` uses `fetchzip` to download the source archive generated by GitHub for the specified revision. If `leaveDotGit`, `deepClone` or `fetchSubmodules` are set to `true`, `fetchFromGitHub` will use `fetchgit` instead. Refer to its section for documentation of these options.

11.7. `fetchFromGitLab`

This is used with GitLab repositories. The arguments expected are very similar to fetchFromGitHub above.

11.8. `fetchFromGitiles`

This is used with Gitiles repositories. The arguments expected are similar to fetchgit.

11.9. `fetchFromBitbucket`

This is used with BitBucket repositories. The arguments expected are very similar to fetchFromGitHub above.

11.10. `fetchFromSavannah`

This is used with Savannah repositories. The arguments expected are very similar to fetchFromGitHub above.

11.11. `fetchFromRepoOrCz`

This is used with repo.or.cz repositories. The arguments expected are very similar to fetchFromGitHub above.

11.12. `fetchFromSourcehut`

This is used with sourcehut repositories. The arguments expected are very similar to fetchFromGitHub above. Don't forget the tilde (~) in front of the user name!

Chapter 12. Trivial builders

Table of Contents

12.1. `runCommand`

12.2. `runCommandCC`

12.3. `runCommandLocal`

12.4. `writeTextFile`, `writeText`, `writeTextDir`, `writeScript`, `writeScriptBin`

12.5. `symlinkJoin`

12.6. `writeReferencesToFile`

12.7. `writeDirectReferencesToFile`

Nixpkgs provides a couple of functions that help with building derivations. The most important one, `stdenv.mkDerivation`, has already been documented above. The following functions wrap `stdenv.mkDerivation`, making it easier to use in certain cases.

12.1. `runCommand`

This takes three arguments, `name`, `env`, and `buildCommand`. `name` is just the name that Nix will append to the store path in the same way that `stdenv.mkDerivation` uses its `name` attribute. `env` is an attribute set specifying environment variables that will be set for this derivation. These attributes are then passed to the wrapped `stdenv.mkDerivation`. `buildCommand` specifies the commands that will be run to create this derivation. Note that you will need to create `$out` for Nix to register the command as successful.

An example of using `runCommand` is provided below.

```
(import <nixpkgs> {}).runCommand "my-example" {} ''
  echo My example command is running

  mkdir $out

  echo I can write data to the Nix store > $out/message

  echo I can also run basic commands like:

  echo ls
  ls

  echo whoami
  whoami

  echo date
  date
''
```

12.2. `runCommandCC`

This works just like `runCommand`. The only difference is that it also provides a C compiler in `buildCommand`'s environment. To minimize your dependencies, you should only use this if you are sure you will need a C compiler as part of running your command.

12.3. `runCommandLocal`

Variant of `runCommand` that forces the derivation to be built locally, it is not substituted. This is intended for very cheap commands (<1s execution time). It saves on the network roundrip and can speed up a build.

> **Note:** This sets `allowSubstitutes` to `false`, so only use `runCommandLocal` if you are certain the user will always have a builder for the `system` of the derivation. This should be true for most trivial use cases (e.g. just copying some files to a different location or adding symlinks), because there the `system` is usually the same as `builtins.currentSystem`.

12.4. `writeTextFile`, `writeText`, `writeTextDir`, `writeScript`, `writeScriptBin`

These functions write `text` to the Nix store. This is useful for creating scripts from Nix expressions. `writeTextFile` takes an attribute set and expects two arguments, `name` and `text`. `name` corresponds to the name used in the Nix store path. `text` will be the contents of the file. You can also set `executable` to true to make this file have the executable bit set.

Many more commands wrap `writeTextFile` including `writeText`, `writeTextDir`, `writeScript`, and `writeScriptBin`. These are convenience functions over `writeTextFile`.

12.5. `symlinkJoin`

This can be used to put many derivations into the same directory structure. It works by creating a new derivation and adding symlinks to each of the paths listed. It expects two arguments, `name`, and `paths`. `name` is the name used in the Nix store path for the created derivation. `paths` is a list of paths that will be symlinked. These paths can be to Nix store derivations or any other subdirectory contained within.

12.6. `writeReferencesToFile`

Writes the closure of transitive dependencies to a file.

This produces the equivalent of `nix-store -q --requisites`.

For example,

```
writeReferencesToFile (writeScriptBin "hi" ''${hello}/bin/hello'')
```

produces an output path `/nix/store/<hash>-runtime-deps` containing

```
/nix/store/<hash>-hello-2.10
/nix/store/<hash>-hi
/nix/store/<hash>-libidn2-2.3.0
```

```
/nix/store/<hash>-libunistring-0.9.10
/nix/store/<hash>-glibc-2.32-40
```

You can see that this includes `hi`, the original input path, `hello`, which is a direct reference, but also the other paths that are indirectly required to run `hello`.

12.7. `writeDirectReferencesToFile`

Writes the set of references to the output file, that is, their immediate dependencies.

This produces the equivalent of `nix-store -q --references`.

For example,

```
writeDirectReferencesToFile (writeScriptBin "hi" ''${hello}/bin/hello'')
```

produces an output path `/nix/store/<hash>-runtime-references` containing

```
/nix/store/<hash>-hello-2.10
```

but none of `hello`'s dependencies, because those are not referenced directly by `hi`'s output.

Chapter 13. Special builders

Table of Contents

13.1. buildFHSUserEnv

13.2. pkgs.mkShell

This chapter describes several special builders.

13.1. buildFHSUserEnv

`buildFHSUserEnv` provides a way to build and run FHS-compatible lightweight sandboxes. It creates an isolated root with bound `/nix/store`, so its footprint in terms of disk space needed is quite small. This allows one to run software which is hard or unfeasible to patch for NixOS – 3rd-party source trees with FHS assumptions, games distributed as tarballs, software with integrity checking and/or external self-updated binaries. It uses Linux namespaces feature to create temporary lightweight environments which are destroyed after all child processes exit, without root user rights requirement. Accepted arguments are:

- `name` Environment name.
- `targetPkgs` Packages to be installed for the main host's architecture (i.e. x86_64 on x86_64

installations). Along with libraries binaries are also installed.

- `multiPkgs` Packages to be installed for all architectures supported by a host (i.e. i686 and x86_64 on x86_64 installations). Only libraries are installed by default.
- `extraBuildCommands` Additional commands to be executed for finalizing the directory structure.
- `extraBuildCommandsMulti` Like `extraBuildCommands`, but executed only on multilib architectures.
- `extraOutputsToInstall` Additional derivation outputs to be linked for both target and multi-architecture packages.
- `extraInstallCommands` Additional commands to be executed for finalizing the derivation with runner script.
- `runScript` A command that would be executed inside the sandbox and passed all the command line arguments. It defaults to `bash`.

One can create a simple environment using a `shell.nix` like that:

```
{ pkgs ? import <nixpkgs> {} }:

(pkgs.buildFHSUserEnv {
  name = "simple-x11-env";
  targetPkgs = pkgs: (with pkgs;
    [ udev
      alsaLib
    ]) ++ (with pkgs.xorg;
    [ libX11
      libXcursor
      libXrandr
    ]);
  multiPkgs = pkgs: (with pkgs;
    [ udev
      alsaLib
    ]);
  runScript = "bash";
}).env
```

Running `nix-shell` would then drop you into a shell with these libraries and binaries available. You can use this to run closed-source applications which expect FHS structure without hassles: simply change `runScript` to the application path, e.g. `./bin/start.sh` – relative paths are supported.

13.2. pkgs.mkShell

`pkgs.mkShell` is a special kind of derivation that is only useful when using it combined with `nix-shell`. It will in fact fail to instantiate when invoked with `nix-build`.

13.2.1. Usage

```
{ pkgs ? import <nixpkgs> {} }:
pkgs.mkShell {
  # specify which packages to add to the shell environment
  packages = [ pkgs.gnumake ];
  # add all the dependencies, of the given packages, to the shell environme
  inputsFrom = with pkgs; [ hello gnutar ];
}
```

Chapter 14. Images

Table of Contents

14.1. pkgs.appimageTools

14.2. pkgs.dockerTools

14.3. pkgs.ociTools

14.4. pkgs.snapTools

This chapter describes tools for creating various types of images.

14.1. pkgs.appimageTools

`pkgs.appimageTools` is a set of functions for extracting and wrapping AppImage files. They are meant to be used if traditional packaging from source is infeasible, or it would take too long. To quickly run an AppImage file, `pkgs.appimage-run` can be used as well.

> **Warning:** The `appimageTools` API is unstable and may be subject to backwards-incompatible changes in the future.

14.1.1. AppImage formats

There are different formats for AppImages, see the specification for details.

- Type 1 images are ISO 9660 files that are also ELF executables.
- Type 2 images are ELF executables with an appended filesystem.

They can be told apart with `file -k`:

```
$ file -k type1.AppImage
type1.AppImage: ELF 64-bit LSB executable, x86-64, version 1 (SYSV) ISO 966
spot sensor temperature 0.000000, unit celsius, color scheme 0, calibration

$ file -k type2.AppImage
type2.AppImage: ELF 64-bit LSB executable, x86-64, version 1 (SYSV) (Lepton
```

Note how the type 1 AppImage is described as an `ISO 9660 CD-ROM filesystem`, and the type 2 AppImage is not.

14.1.2. Wrapping

Depending on the type of AppImage you're wrapping, you'll have to use `wrapType1` or `wrapType2`.

```
appimageTools.wrapType2 { # or wrapType1
  name = "patchwork";
  src = fetchurl {
    url = "https://github.com/ssbc/patchwork/releases/download/v3.11.4/Patc
    sha256 = "1blsprpkvm0ws9b96gb36f0rbf8f5jgmw4x6dsb1kswr4ysf591s";
  };
  extraPkgs = pkgs: with pkgs; [ ];
}
```

- `name` specifies the name of the resulting image.
- `src` specifies the AppImage file to extract.
- `extraPkgs` allows you to pass a function to include additional packages inside the FHS environment your AppImage is going to run in. There are a few ways to learn which dependencies an application needs:
  - Looking through the extracted AppImage files, reading its scripts and running `patchelf` and `ldd` on its executables. This can also be done in `appimage-run`, by setting `APPIMAGE_DEBUG_EXEC=bash`.
  - Running `strace -vfefile` on the wrapped executable, looking for libraries that can't be found.

14.2. pkgs.dockerTools

`pkgs.dockerTools` is a set of functions for creating and manipulating Docker images according to the Docker Image Specification v1.2.0. Docker itself is not used to perform any of the operations done by

these functions.

14.2.1. buildImage

This function is analogous to the `docker build` command, in that it can be used to build a Docker-compatible repository tarball containing a single image with one or multiple layers. As such, the result is suitable for being loaded in Docker with `docker load`.

The parameters of `buildImage` with relative example values are described below:

```
buildImage {
  name = "redis";
  tag = "latest";

  fromImage = someBaseImage;
  fromImageName = null;
  fromImageTag = "latest";

  contents = pkgs.redis;
  runAsRoot = ''
    #!${pkgs.runtimeShell}
    mkdir -p /data
  '';

  config = {
    Cmd = [ "/bin/redis-server" ];
    WorkingDir = "/data";
    Volumes = { "/data" = { }; };
  };
}
```

The above example will build a Docker image `redis/latest` from the given base image. Loading and running this image in Docker results in `redis-server` being started automatically.

- `name` specifies the name of the resulting image. This is the only required argument for `buildImage`.
- `tag` specifies the tag of the resulting image. By default it's `null`, which indicates that the nix output hash will be used as tag.
- `fromImage` is the repository tarball containing the base image. It must be a valid Docker image, such as exported by `docker save`. By default it's `null`, which can be seen as equivalent to `FROM scratch` of a `Dockerfile`.
- `fromImageName` can be used to further specify the base image within the repository, in case it contains multiple images. By default it's `null`, in which case `buildImage` will peek the first image available in the repository.

- `fromImageTag` can be used to further specify the tag of the base image within the repository, in case an image contains multiple tags. By default it's `null`, in which case `buildImage` will peek the first tag available for the base image.

- `contents` is a derivation that will be copied in the new layer of the resulting image. This can be similarly seen as `ADD contents/ /` in a `Dockerfile`. By default it's `null`.

- `runAsRoot` is a bash script that will run as root in an environment that overlays the existing layers of the base image with the new resulting layer, including the previously copied `contents` derivation. This can be similarly seen as `RUN ...` in a `Dockerfile`.

  NOTE: Using this parameter requires the `kvm` device to be available.

- `config` is used to specify the configuration of the containers that will be started off the built image in Docker. The available options are listed in the Docker Image Specification v1.2.0.

After the new layer has been created, its closure (to which `contents`, `config` and `runAsRoot` contribute) will be copied in the layer itself. Only new dependencies that are not already in the existing layers will be copied.

At the end of the process, only one new single layer will be produced and added to the resulting image.

The resulting repository will only list the single image `image/tag`. In the case of the `buildImage` example it would be `redis/latest`.

It is possible to inspect the arguments with which an image was built using its `buildArgs` attribute.

> ***NOTE:*** If you see errors similar to `getProtocolByName: does not exist (no such protocol name: tcp)` you may need to add `pkgs.iana-etc` to `contents`.

> ***NOTE:*** If you see errors similar to `Error_Protocol ("certificate has unknown CA",True,UnknownCa)` you may need to add `pkgs.cacert` to `contents`.

By default `buildImage` will use a static date of one second past the UNIX Epoch. This allows `buildImage` to produce binary reproducible images. When listing images with `docker images`, the newly created images will be listed like this:

```
$ docker images
REPOSITORY   TAG      IMAGE ID       CREATED        SIZE
hello        latest   08c791c7846e   48 years ago   25.2MB
```

You can break binary reproducibility but have a sorted, meaningful `CREATED` column by setting `created` to `now`.

```
pkgs.dockerTools.buildImage {
```

```
  name = "hello";
  tag = "latest";
  created = "now";
  contents = pkgs.hello;

  config.Cmd = [ "/bin/hello" ];
}
```

and now the Docker CLI will display a reasonable date and sort the images as expected:

```
$ docker images
REPOSITORY   TAG      IMAGE ID       CREATED              SIZE
hello        latest   de2bf4786de6   About a minute ago   25.2MB
```

however, the produced images will not be binary reproducible.

14.2.2. buildLayeredImage

Create a Docker image with many of the store paths being on their own layer to improve sharing between images. The image is realized into the Nix store as a gzipped tarball. Depending on the intended usage, many users might prefer to use `streamLayeredImage` instead, which this function uses internally.

`name`

The name of the resulting image.

`tag` *optional*

Tag of the generated image.

Default: the output path's hash

`fromImage` *optional*

The repository tarball containing the base image. It must be a valid Docker image, such as one exported by `docker save`.

Default: `null`, which can be seen as equivalent to `FROM scratch` of a `Dockerfile`.

`contents` *optional*

Top level paths in the container. Either a single derivation, or a list of derivations.

Default: `[]`

`config` *optional*

Run-time configuration of the container. A full list of the options are available at in the Docker Image Specification v1.2.0.

Default: `{}`

`created` *optional*

Date and time the layers were created. Follows the same `now` exception supported by `buildImage`.

Default: `1970-01-01T00:00:01Z`

`maxLayers` *optional*

Maximum number of layers to create.

Default: `100`

Maximum: `125`

`extraCommands` *optional*

Shell commands to run while building the final layer, without access to most of the layer contents. Changes to this layer are "on top" of all the other layers, so can create additional directories and files.

`fakeRootCommands` *optional*

Shell commands to run while creating the archive for the final layer in a fakeroot environment. Unlike `extraCommands`, you can run `chown` to change the owners of the files in the archive, changing fakeroot's state instead of the real filesystem. The latter would require privileges that the build user does not have. Static binaries do not interact with the fakeroot environment. By default all files in the archive will be owned by root.

14.2.2.1. Behavior of `contents` in the final image

Each path directly listed in `contents` will have a symlink in the root of the image.

For example:

```
pkgs.dockerTools.buildLayeredImage {
  name = "hello";
  contents = [ pkgs.hello ];
}
```

will create symlinks for all the paths in the `hello` package:

```
/bin/hello -> /nix/store/h1zb1padqbbb7jicsvkmrym3r6snphxg-hello-2.10/bin/he
/share/info/hello.info -> /nix/store/h1zb1padqbbb7jicsvkmrym3r6snphxg-hello
/share/locale/bg/LC_MESSAGES/hello.mo -> /nix/store/h1zb1padqbbb7jicsvkmrym
```

14.2.2.2. Automatic inclusion of `config` references

The closure of `config` is automatically included in the closure of the final image.

This allows you to make very simple Docker images with very little code. This container will start up and run `hello`:

```
pkgs.dockerTools.buildLayeredImage {
  name = "hello";
  config.Cmd = [ "${pkgs.hello}/bin/hello" ];
}
```

14.2.2.3. Adjusting `maxLayers`

Increasing the `maxLayers` increases the number of layers which have a chance to be shared between different images.

Modern Docker installations support up to 128 layers, however older versions support as few as 42.

If the produced image will not be extended by other Docker builds, it is safe to set `maxLayers` to `128`. However it will be impossible to extend the image further.

The first (`maxLayers-2`) most "popular" paths will have their own individual layers, then layer # `maxLayers-1` will contain all the remaining "unpopular" paths, and finally layer #`maxLayers` will contain the Image configuration.

Docker's Layers are not inherently ordered, they are content-addressable and are not explicitly layered until they are composed in to an Image.

14.2.3. streamLayeredImage

Builds a script which, when run, will stream an uncompressed tarball of a Docker image to stdout. The arguments to this function are as for `buildLayeredImage`. This method of constructing an image does not realize the image into the Nix store, so it saves on IO and disk/cache space, particularly with large images.

The image produced by running the output script can be piped directly into `docker load`, to load it into the local docker daemon:

```
$(nix-build) | docker load
```

Alternatively, the image be piped via `gzip` into `skopeo`, e.g. to copy it into a registry:

```
$(nix-build) | gzip --fast | skopeo copy docker-archive:/dev/stdin docker:/
```

14.2.4. pullImage

This function is analogous to the `docker pull` command, in that it can be used to pull a Docker image

from a Docker registry. By default Docker Hub is used to pull images.

Its parameters are described in the example below:

```
pullImage {
  imageName = "nixos/nix";
  imageDigest =
    "sha256:20d9485b25ecfd89204e843a962c1bd70e9cc6858d65d7f5fadc340246e2116
  finalImageName = "nix";
  finalImageTag = "1.11";
  sha256 = "0mqjy3zq2v6rrhizgb9nvhczl87lcfphq9601wcprdika2jz7qh8";
  os = "linux";
  arch = "x86_64";
}
```

- `imageName` specifies the name of the image to be downloaded, which can also include the registry namespace (e.g. `nixos`). This argument is required.
- `imageDigest` specifies the digest of the image to be downloaded. This argument is required.
- `finalImageName`, if specified, this is the name of the image to be created. Note it is never used to fetch the image since we prefer to rely on the immutable digest ID. By default it's equal to `imageName`.
- `finalImageTag`, if specified, this is the tag of the image to be created. Note it is never used to fetch the image since we prefer to rely on the immutable digest ID. By default it's `latest`.
- `sha256` is the checksum of the whole fetched image. This argument is required.
- `os`, if specified, is the operating system of the fetched image. By default it's `linux`.
- `arch`, if specified, is the cpu architecture of the fetched image. By default it's `x86_64`.

`nix-prefetch-docker` command can be used to get required image parameters:

```
$ nix run nixpkgs.nix-prefetch-docker -c nix-prefetch-docker --image-name m
```

Since a given `imageName` may transparently refer to a manifest list of images which support multiple architectures and/or operating systems, you can supply the `--os` and `--arch` arguments to specify exactly which image you want. By default it will match the OS and architecture of the host the command is run on.

```
$ nix-prefetch-docker --image-name mysql --image-tag 5 --arch x86_64 --os l
```

Desired image name and tag can be set using `--final-image-name` and `--final-image-tag`

arguments:

```
$ nix-prefetch-docker --image-name mysql --image-tag 5 --final-image-name e
```

14.2.5. exportImage

This function is analogous to the `docker export` command, in that it can be used to flatten a Docker image that contains multiple layers. It is in fact the result of the merge of all the layers of the image. As such, the result is suitable for being imported in Docker with `docker import`.

> ***NOTE:*** Using this function requires the `kvm` device to be available.

The parameters of `exportImage` are the following:

```
exportImage {
  fromImage = someLayeredImage;
  fromImageName = null;
  fromImageTag = null;

  name = someLayeredImage.name;
}
```

The parameters relative to the base image have the same synopsis as described in buildImage, except that `fromImage` is the only required argument in this case.

The `name` argument is the name of the derivation output, which defaults to `fromImage.name`.

14.2.6. shadowSetup

This constant string is a helper for setting up the base files for managing users and groups, only if such files don't exist already. It is suitable for being used in a `buildImage runAsRoot` script for cases like in the example below:

```
buildImage {
  name = "shadow-basic";

  runAsRoot = ''
    #!${pkgs.runtimeShell}
    ${shadowSetup}
    groupadd -r redis
    useradd -r -g redis redis
    mkdir /data
    chown redis:redis /data
  '';
```

```
}
```

Creating base files like `/etc/passwd` or `/etc/login.defs` is necessary for shadow-utils to manipulate users and groups.

14.3. pkgs.ociTools

`pkgs.ociTools` is a set of functions for creating containers according to the OCI container specification v1.0.0. Beyond that it makes no assumptions about the container runner you choose to use to run the created container.

14.3.1. buildContainer

This function creates a simple OCI container that runs a single command inside of it. An OCI container consists of a `config.json` and a rootfs directory.The nix store of the container will contain all referenced dependencies of the given command.

The parameters of `buildContainer` with an example value are described below:

```
buildContainer {
  args = [
    (with pkgs;
      writeScript "run.sh" ''
        #!${bash}/bin/bash
        exec ${bash}/bin/bash
      '').outPath
  ];

  mounts = {
    "/data" = {
      type = "none";
      source = "/var/lib/mydata";
      options = [ "bind" ];
    };
  };

  readonly = false;
}
```

- `args` specifies a set of arguments to run inside the container. This is the only required argument for `buildContainer`. All referenced packages inside the derivation will be made available inside the container
- `mounts` specifies additional mount points chosen by the user. By default only a minimal set of

necessary filesystems are mounted into the container (e.g procfs, cgroupfs)

- `readonly` makes the container's rootfs read-only if it is set to true. The default value is false `false`.

14.4. pkgs.snapTools

`pkgs.snapTools` is a set of functions for creating Snapcraft images. Snap and Snapcraft is not used to perform these operations.

14.4.1. The makeSnap Function

`makeSnap` takes a single named argument, `meta`. This argument mirrors the upstream `snap.yaml` format exactly.

The `base` should not be specified, as `makeSnap` will force set it.

Currently, `makeSnap` does not support creating GUI stubs.

14.4.2. Build a Hello World Snap

The following expression packages GNU Hello as a Snapcraft snap.

```
let
  inherit (import <nixpkgs> { }) snapTools hello;
in snapTools.makeSnap {
  meta = {
    name = "hello";
    summary = hello.meta.description;
    description = hello.meta.longDescription;
    architectures = [ "amd64" ];
    confinement = "strict";
    apps.hello.command = "${hello}/bin/hello";
  };
}
```

`nix-build` this expression and install it with `snap install ./result --dangerous`. `hello` will now be the Snapcraft version of the package.

14.4.3. Build a Graphical Snap

Graphical programs require many more integrations with the host. This example uses Firefox as an example, because it is one of the most complicated programs we could package.

```
let
  inherit (import <nixpkgs> { }) snapTools firefox;
```

```
in snapTools.makeSnap {
  meta = {
    name = "nix-example-firefox";
    summary = firefox.meta.description;
    architectures = [ "amd64" ];
    apps.nix-example-firefox = {
      command = "${firefox}/bin/firefox";
      plugs = [
        "pulseaudio"
        "camera"
        "browser-support"
        "avahi-observe"
        "cups-control"
        "desktop"
        "desktop-legacy"
        "gsettings"
        "home"
        "network"
        "mount-observe"
        "removable-media"
        "x11"
      ];
    };
    confinement = "strict";
  };
}
```

`nix-build` this expression and install it with `snap install ./result --dangerous`. `nix-example-firefox` will now be the Snapcraft version of the Firefox package.

The specific meaning behind plugs can be looked up in the Snapcraft interface documentation.

Chapter 15. Languages and frameworks

Table of Contents

15.1. Agda

15.2. Android

15.3. BEAM Languages (Erlang, Elixir & LFE)

15.4. Bower

15.5. Coq and coq packages

15.6. Crystal

15.7. Dhall

15.8. Emscripten

15.9. GNOME

15.10. Go

15.11. Haskell

15.12. Idris

15.13. iOS

15.14. Java

15.15. User's Guide to Lua Infrastructure

15.16. Maven

15.17. Node.js

15.18. OCaml

15.19. Perl

15.20. PHP

15.21. Python

15.22. Qt

15.23. R

15.24. Ruby

15.25. Rust

15.26. TeX Live

15.27. Titanium

15.28. Vim

The standard build environment makes it easy to build typical Autotools-based packages with very little code. Any other kind of package can be accomodated by overriding the appropriate phases of `stdenv`. However, there are specialised functions in Nixpkgs to easily build packages for other programming languages, such as Perl or Haskell. These are described in this chapter.

15.1. Agda

15.1.1. How to use Agda

Agda is available as the agda package.

The `agda` package installs an Agda-wrapper, which calls `agda` with `--library-file` set to a generated library-file within the nix store, this means your library-file in `$HOME/.agda/libraries` will be ignored. By default the agda package installs Agda with no libraries, i.e. the generated library-file is empty. To use Agda with libraries, the `agda.withPackages` function can be used. This function either takes:

- A list of packages,
- or a function which returns a list of packages when given the `agdaPackages` attribute set,
- or an attribute set containing a list of packages and a GHC derivation for compilation (see below).
- or an attribute set containing a function which returns a list of packages when given the `agdaPackages` attribute set and a GHC derivation for compilation (see below).

For example, suppose we wanted a version of Agda which has access to the standard library. This can be obtained with the expressions:

```
agda.withPackages [ agdaPackages.standard-library ]
```

or

```
agda.withPackages (p: [ p.standard-library ])
```

or can be called as in the Compiling Agda section.

If you want to use a different version of a library (for instance a development version) override the `src` attribute of the package to point to your local repository

```
agda.withPackages (p: [
  (p.standard-library.overrideAttrs (oldAttrs: {
    version = "local version";
    src = /path/to/local/repo/agda-stdlib;
  }))
])
```

You can also reference a GitHub repository

```
agda.withPackages (p: [
  (p.standard-library.overrideAttrs (oldAttrs: {
    version = "1.5";
    src =  fetchFromGitHub {
```

```
      repo = "agda-stdlib";
      owner = "agda";
      rev = "v1.5";
      sha256 = "16fcb7ssj6kj687a042afaa2gq48rc8abihpm14k684ncihb2k4w";
    };
  }))
])
```

If you want to use a library not added to Nixpkgs, you can add a dependency to a local library by calling `agdaPackages.mkDerivation`.

```
agda.withPackages (p: [
  (p.mkDerivation {
    pname = "your-agda-lib";
    version = "1.0.0";
    src = /path/to/your-agda-lib;
  })
])
```

Again you can reference GitHub

```
agda.withPackages (p: [
  (p.mkDerivation {
    pname = "your-agda-lib";
    version = "1.0.0";
    src = fetchFromGitHub {
      repo = "repo";
      owner = "owner";
      version = "...";
      rev = "...";
      sha256 = "...";
    };
  })
])
```

See Building Agda Packages for more information on `mkDerivation`.

Agda will not by default use these libraries. To tell Agda to use a library we have some options:

- Call `agda` with the library flag:

```
$ agda -l standard-library -i . MyFile.agda
```

- Write a `my-library.agda-lib` file for the project you are working on which may look like:

```
name: my-library
include: .
depend: standard-library
```

- Create the file `~/.agda/defaults` and add any libraries you want to use by default.

More information can be found in the [official Agda documentation on library management](#).

15.1.2. Compiling Agda

Agda modules can be compiled using the GHC backend with the `--compile` flag. A version of `ghc` with `ieee754` is made available to the Agda program via the `--with-compiler` flag. This can be overridden by a different version of `ghc` as follows:

```
agda.withPackages {
  pkgs = [ ... ];
  ghc = haskell.compiler.ghcHEAD;
}
```

15.1.3. Writing Agda packages

To write a nix derivation for an Agda library, first check that the library has a `*.agda-lib` file.

A derivation can then be written using `agdaPackages.mkDerivation`. This has similar arguments to `stdenv.mkDerivation` with the following additions:

- `everythingFile` can be used to specify the location of the `Everything.agda` file, defaulting to `./Everything.agda`. If this file does not exist then either it should be patched in or the `buildPhase` should be overridden (see below).
- `libraryName` should be the name that appears in the `*.agda-lib` file, defaulting to `pname`.
- `libraryFile` should be the file name of the `*.agda-lib` file, defaulting to `${libraryName}.agda-lib`.

Here is an example `default.nix`

```
{ nixpkgs ?  <nixpkgs> }:
with (import nixpkgs {});
agdaPackages.mkDerivation {
  version = "1.0";
  pname = "my-agda-lib";
  src = ./.;
```

```
  buildInputs = [
    agdaPackages.standard-library
  ];
}
```

15.1.3.1. Building Agda packages

The default build phase for `agdaPackages.mkDerivation` simply runs `agda` on the `Everything.agda` file. If something else is needed to build the package (e.g. `make`) then the `buildPhase` should be overridden. Additionally, a `preBuild` or `configurePhase` can be used if there are steps that need to be done prior to checking the `Everything.agda` file. `agda` and the Agda libraries contained in `buildInputs` are made available during the build phase.

15.1.3.2. Installing Agda packages

The default install phase copies Agda source files, Agda interface files (`*.agdai`) and `*.agda-lib` files to the output directory. This can be overridden.

By default, Agda sources are files ending on `.agda`, or literate Agda files ending on `.lagda`, `.lagda.tex`, `.lagda.org`, `.lagda.md`, `.lagda.rst`. The list of recognised Agda source extensions can be extended by setting the `extraExtensions` config variable.

15.1.4. Adding Agda packages to Nixpkgs

To add an Agda package to `nixpkgs`, the derivation should be written to `pkgs/development/libraries/agda/${library-name}/` and an entry should be added to `pkgs/top-level/agda-packages.nix`. Here it is called in a scope with access to all other Agda libraries, so the top line of the `default.nix` can look like:

```
{ mkDerivation, standard-library, fetchFromGitHub }:
```

Note that the derivation function is called with `mkDerivation` set to `agdaPackages.mkDerivation`, therefore you could use a similar set as in your `default.nix` from Writing Agda Packages with `agdaPackages.mkDerivation` replaced with `mkDerivation`.

Here is an example skeleton derivation for iowa-stdlib:

```
mkDerivation {
  version = "1.5.0";
  pname = "iowa-stdlib";

  src = ...

  libraryFile = "";
  libraryName = "IAL-1.3";
```

```
  buildPhase = ''
    patchShebangs find-deps.sh
    make
  '';
}
```

This library has a file called `.agda-lib`, and so we give an empty string to `libraryFile` as nothing precedes `.agda-lib` in the filename. This file contains `name: IAL-1.3`, and so we let `libraryName = "IAL-1.3"`. This library does not use an `Everything.agda` file and instead has a Makefile, so there is no need to set `everythingFile` and we set a custom `buildPhase`.

When writing an Agda package it is essential to make sure that no `.agda-lib` file gets added to the store as a single file (for example by using `writeText`). This causes Agda to think that the nix store is a Agda library and it will attempt to write to it whenever it typechecks something. See https://github.com/agda/agda/issues/4613.

15.2. Android

The Android build environment provides three major features and a number of supporting features.

15.2.1. Deploying an Android SDK installation with plugins

The first use case is deploying the SDK with a desired set of plugins or subsets of an SDK.

```
with import <nixpkgs> {};

let
  androidComposition = androidenv.composeAndroidPackages {
    toolsVersion = "26.1.1";
    platformToolsVersion = "30.0.5";
    buildToolsVersions = [ "30.0.3" ];
    includeEmulator = false;
    emulatorVersion = "30.3.4";
    platformVersions = [ "28" "29" "30" ];
    includeSources = false;
    includeSystemImages = false;
    systemImageTypes = [ "google_apis_playstore" ];
    abiVersions = [ "armeabi-v7a" "arm64-v8a" ];
    cmakeVersions = [ "3.10.2" ];
    includeNDK = true;
    ndkVersions = ["22.0.7026061"];
    useGoogleAPIs = false;
    useGoogleTVAddOns = false;
```

```
    includeExtras = [
      "extras;google;gcm"
    ];
  };
in
androidComposition.androidsdk
```

The above function invocation states that we want an Android SDK with the above specified plugin versions. By default, most plugins are disabled. Notable exceptions are the tools, platform-tools and build-tools sub packages.

The following parameters are supported:

- `toolsVersion`, specifies the version of the tools package to use
- `platformsToolsVersion` specifies the version of the `platform-tools` plugin
- `buildToolsVersions` specifies the versions of the `build-tools` plugins to use.
- `includeEmulator` specifies whether to deploy the emulator package (`false` by default). When enabled, the version of the emulator to deploy can be specified by setting the `emulatorVersion` parameter.
- `cmakeVersions` specifies which CMake versions should be deployed.
- `includeNDK` specifies that the Android NDK bundle should be included. Defaults to: `false`.
- `ndkVersions` specifies the NDK versions that we want to use. These are linked under the `ndk` directory of the SDK root, and the first is linked under the `ndk-bundle` directory.
- `ndkVersion` is equivalent to specifying one entry in `ndkVersions`, and `ndkVersions` overrides this parameter if provided.
- `includeExtras` is an array of identifier strings referring to arbitrary add-on packages that should be installed.
- `platformVersions` specifies which platform SDK versions should be included.

For each platform version that has been specified, we can apply the following options:

- `includeSystemImages` specifies whether a system image for each platform SDK should be included.
- `includeSources` specifies whether the sources for each SDK version should be included.
- `useGoogleAPIs` specifies that for each selected platform version the Google API should be included.
- `useGoogleTVAddOns` specifies that for each selected platform version the Google TV add-on should be included.

For each requested system image we can specify the following options:

- `systemImageTypes` specifies what kind of system images should be included. Defaults to: `default`.
- `abiVersions` specifies what kind of ABI version of each system image should be included. Defaults to: `armeabi-v7a`.

Most of the function arguments have reasonable default settings.

You can specify license names:

- `extraLicenses` is a list of license names. You can get these names from repo.json or `querypackages.sh licenses`. The SDK license (`android-sdk-license`) is accepted for you if you set accept_license to true. If you are doing something like working with preview SDKs, you will want to add `android-sdk-preview-license` or whichever license applies here.

Additionally, you can override the repositories that composeAndroidPackages will pull from:

- `repoJson` specifies a path to a generated repo.json file. You can generate this by running `generate.sh`, which in turn will call into `mkrepo.rb`.
- `repoXmls` is an attribute set containing paths to repo XML files. If specified, it takes priority over `repoJson`, and will trigger a local build writing out a repo.json to the Nix store based on the given repository XMLs.

```
repoXmls = {
  packages = [ ./xml/repository2-1.xml ];
  images = [
    ./xml/android-sys-img2-1.xml
    ./xml/android-tv-sys-img2-1.xml
    ./xml/android-wear-sys-img2-1.xml
    ./xml/android-wear-cn-sys-img2-1.xml
    ./xml/google_apis-sys-img2-1.xml
    ./xml/google_apis_playstore-sys-img2-1.xml
  ];
  addons = [ ./xml/addon2-1.xml ];
};
```

When building the above expression with:

```
$ nix-build
```

The Android SDK gets deployed with all desired plugin versions.

We can also deploy subsets of the Android SDK. For example, to only the `platform-tools` package, you can evaluate the following expression:

```
with import <nixpkgs> {};

let
  androidComposition = androidenv.composeAndroidPackages {
    # ...
  };
in
androidComposition.platform-tools
```

15.2.2. Using predefined Android package compositions

In addition to composing an Android package set manually, it is also possible to use a predefined composition that contains all basic packages for a specific Android version, such as version 9.0 (API-level 28).

The following Nix expression can be used to deploy the entire SDK with all basic plugins:

```
with import <nixpkgs> {};

androidenv.androidPkgs_9_0.androidsdk
```

It is also possible to use one plugin only:

```
with import <nixpkgs> {};

androidenv.androidPkgs_9_0.platform-tools
```

15.2.3. Building an Android application

In addition to the SDK, it is also possible to build an Ant-based Android project and automatically deploy all the Android plugins that a project requires.

```
with import <nixpkgs> {};

androidenv.buildApp {
  name = "MyAndroidApp";
  src = ./myappsources;
  release = true;

  # If release is set to true, you need to specify the following parameters
  keyStore = ./keystore;
  keyAlias = "myfirstapp";
  keyStorePassword = "mykeystore";
```

```
  keyAliasPassword = "myfirstapp";

  # Any Android SDK parameters that install all the relevant plugins that a
  # build requires
  platformVersions = [ "24" ];

  # When we include the NDK, then ndk-build is invoked before Ant gets invo
  includeNDK = true;
}
```

Aside from the app-specific build parameters (`name`, `src`, `release` and keystore parameters), the `buildApp {}` function supports all the function parameters that the SDK composition function (the function shown in the previous section) supports.

This build function is particularly useful when it is desired to use Hydra: the Nix-based continuous integration solution to build Android apps. An Android APK gets exposed as a build product and can be installed on any Android device with a web browser by navigating to the build result page.

15.2.4. Spawning emulator instances

For testing purposes, it can also be quite convenient to automatically generate scripts that spawn emulator instances with all desired configuration settings.

An emulator spawn script can be configured by invoking the `emulateApp {}` function:

```
with import <nixpkgs> {};

androidenv.emulateApp {
  name = "emulate-MyAndroidApp";
  platformVersion = "28";
  abiVersion = "x86"; # armeabi-v7a, mips, x86_64
  systemImageType = "google_apis_playstore";
}
```

Additional flags may be applied to the Android SDK's emulator through the runtime environment variable `$NIX_ANDROID_EMULATOR_FLAGS`.

It is also possible to specify an APK to deploy inside the emulator and the package and activity names to launch it:

```
with import <nixpkgs> {};

androidenv.emulateApp {
  name = "emulate-MyAndroidApp";
```

```
  platformVersion = "24";
  abiVersion = "armeabi-v7a"; # mips, x86, x86_64
  systemImageType = "default";
  useGoogleAPIs = false;
  app = ./MyApp.apk;
  package = "MyApp";
  activity = "MainActivity";
}
```

In addition to prebuilt APKs, you can also bind the APK parameter to a `buildApp {}` function invocation shown in the previous example.

15.2.5. Notes on environment variables in Android projects

- `ANDROID_SDK_ROOT` should point to the Android SDK. In your Nix expressions, this should be `${androidComposition.androidsdk}/libexec/android-sdk`. Note that `ANDROID_HOME` is deprecated, but if you rely on tools that need it, you can export it too.
- `ANDROID_NDK_ROOT` should point to the Android NDK, if you're doing NDK development. In your Nix expressions, this should be `${ANDROID_SDK_ROOT}/ndk-bundle`.

If you are running the Android Gradle plugin, you need to export GRADLE_OPTS to override aapt2 to point to the aapt2 binary in the Nix store as well, or use a FHS environment so the packaged aapt2 can run. If you don't want to use a FHS environment, something like this should work:

```
let
  buildToolsVersion = "30.0.3";

  # Use buildToolsVersion when you define androidComposition
  androidComposition = <...>;
in
pkgs.mkShell rec {
  ANDROID_SDK_ROOT = "${androidComposition.androidsdk}/libexec/android-sdk"
  ANDROID_NDK_ROOT = "${ANDROID_SDK_ROOT}/ndk-bundle";

  # Use the same buildToolsVersion here
  GRADLE_OPTS = "-Dorg.gradle.project.android.aapt2FromMavenOverride=${ANDR
}
```

If you are using cmake, you need to add it to PATH in a shell hook or FHS env profile. The path is suffixed with a build number, but properly prefixed with the version. So, something like this should suffice:

```
let
```

```
  cmakeVersion = "3.10.2";

  # Use cmakeVersion when you define androidComposition
  androidComposition = <...>;
in
pkgs.mkShell rec {
  ANDROID_SDK_ROOT = "${androidComposition.androidsdk}/libexec/android-sdk"
  ANDROID_NDK_ROOT = "${ANDROID_SDK_ROOT}/ndk-bundle";

  # Use the same cmakeVersion here
  shellHook = ''
    export PATH="$(echo "$ANDROID_SDK_ROOT/cmake/${cmakeVersion}".*/bin):$P
  '';
}
```

Note that running Android Studio with ANDROID_SDK_ROOT set will automatically write a `local.properties` file with `sdk.dir` set to $ANDROID_SDK_ROOT if one does not already exist. If you are using the NDK as well, you may have to add `ndk.dir` to this file.

An example shell.nix that does all this for you is provided in examples/shell.nix. This shell.nix includes a shell hook that overwrites local.properties with the correct sdk.dir and ndk.dir values. This will ensure that the SDK and NDK directories will both be correct when you run Android Studio inside nix-shell.

15.2.6. Notes on improving build.gradle compatibility

Ensure that your buildToolsVersion and ndkVersion match what is declared in androidenv. If you are using cmake, make sure its declared version is correct too.

Otherwise, you may get cryptic errors from aapt2 and the Android Gradle plugin warning that it cannot install the build tools because the SDK directory is not writeable.

```
android {
    buildToolsVersion "30.0.3"
    ndkVersion = "22.0.7026061"
    externalNativeBuild {
        cmake {
            version "3.10.2"
        }
    }
}
```

15.2.7. Querying the available versions of each plugin

repo.json provides all the options in one file now.

A shell script in the `pkgs/development/mobile/androidenv/` subdirectory can be used to retrieve all possible options:

```
./querypackages.sh packages
```

The above command-line instruction queries all package versions in repo.json.

15.2.8. Updating the generated expressions

repo.json is generated from XML files that the Android Studio package manager uses. To update the expressions run the `generate.sh` script that is stored in the `pkgs/development/mobile/androidenv/` subdirectory:

```
./generate.sh
```

15.3. BEAM Languages (Erlang, Elixir & LFE)

15.3.1. Introduction

In this document and related Nix expressions, we use the term, *BEAM*, to describe the environment. BEAM is the name of the Erlang Virtual Machine and, as far as we're concerned, from a packaging perspective, all languages that run on the BEAM are interchangeable. That which varies, like the build system, is transparent to users of any given BEAM package, so we make no distinction.

15.3.2. Structure

All BEAM-related expressions are available via the top-level `beam` attribute, which includes:

- `interpreters`: a set of compilers running on the BEAM, including multiple Erlang/OTP versions (`beam.interpreters.erlangR22`, etc), Elixir (`beam.interpreters.elixir`) and LFE (Lisp Flavoured Erlang) (`beam.interpreters.lfe`).
- `packages`: a set of package builders (Mix and rebar3), each compiled with a specific Erlang/OTP version, e.g. `beam.packages.erlang22`.

The default Erlang compiler, defined by `beam.interpreters.erlang`, is aliased as `erlang`. The default BEAM package set is defined by `beam.packages.erlang` and aliased at the top level as `beamPackages`.

To create a package builder built with a custom Erlang version, use the lambda, `beam.packagesWith`, which accepts an Erlang/OTP derivation and produces a package builder similar to `beam.packages.erlang`.

Many Erlang/OTP distributions available in `beam.interpreters` have versions with ODBC and/or Java

enabled or without wx (no observer support). For example, there's `beam.interpreters.erlangR22_odbc_javac`, which corresponds to `beam.interpreters.erlangR22` and `beam.interpreters.erlangR22_nox`, which corresponds to `beam.interpreters.erlangR22`.

15.3.3. Build Tools

15.3.3.1. Rebar3

We provide a version of Rebar3, under `rebar3`. We also provide a helper to fetch Rebar3 dependencies from a lockfile under `fetchRebar3Deps`.

We also provide a version on Rebar3 with plugins included, under `rebar3WithPlugins`. This package is a function which takes two arguments: `plugins`, a list of nix derivations to include as plugins (loaded only when specified in `rebar.config`), and `globalPlugins`, which should always be loaded by rebar3. Example: `rebar3WithPlugins { globalPlugins = [beamPackages.pc]; }`.

When adding a new plugin it is important that the `packageName` attribute is the same as the atom used by rebar3 to refer to the plugin.

15.3.3.2. Mix & Erlang.mk

Erlang.mk works exactly as expected. There is a bootstrap process that needs to be run, which is supported by the `buildErlangMk` derivation.

For Elixir applications use `mixRelease` to make a release. See examples for more details.

There is also a `buildMix` helper, whose behavior is closer to that of `buildErlangMk` and `buildRebar3`. The primary difference is that mixRelease makes a release, while buildMix only builds the package, making it useful for libraries and other dependencies.

15.3.4. How to Install BEAM Packages

BEAM builders are not registered at the top level, simply because they are not relevant to the vast majority of Nix users. To install any of those builders into your profile, refer to them by their attribute path `beamPackages.rebar3`:

```
$ nix-env -f "<nixpkgs>" -iA beamPackages.rebar3
```

15.3.5. Packaging BEAM Applications

15.3.5.1. Erlang Applications

15.3.5.1.1. Rebar3 Packages

The Nix function, `buildRebar3`, defined in `beam.packages.erlang.buildRebar3` and aliased at

the top level, can be used to build a derivation that understands how to build a Rebar3 project.

If a package needs to compile native code via Rebar3's port compilation mechanism, add `compilePort = true;` to the derivation.

15.3.5.1.2. Erlang.mk Packages

Erlang.mk functions similarly to Rebar3, except we use `buildErlangMk` instead of `buildRebar3`.

15.3.5.1.3. Mix Packages

`mixRelease` is used to make a release in the mix sense. Dependencies will need to be fetched with `fetchMixDeps` and passed to it.

15.3.5.1.4. mixRelease - Elixir Phoenix example

Here is how your `default.nix` file would look.

```
with import <nixpkgs> { };

let
  packages = beam.packagesWith beam.interpreters.erlang;
  src = builtins.fetchgit {
    url = "ssh://git@github.com/your_id/your_repo";
    rev = "replace_with_your_commit";
  };

  pname = "your_project";
  version = "0.0.1";
  mixEnv = "prod";

  mixDeps = packages.fetchMixDeps {
    pname = "mix-deps-${pname}";
    inherit src mixEnv version;
    # nix will complain and tell you the right value to replace this with
    sha256 = lib.fakeSha256;
    # if you have build time environment variables add them here
    MY_ENV_VAR="my_value";
  };

  nodeDependencies = (pkgs.callPackage ./assets/default.nix { }).shell.node

  frontEndFiles = stdenvNoCC.mkDerivation {
    pname = "frontend-${pname}";
```

```
    nativeBuildInputs = [ nodejs ];

    inherit version src;

    buildPhase = ''
      cp -r ./assets $TEMPDIR

      mkdir -p $TEMPDIR/assets/node_modules/.cache
      cp -r ${nodeDependencies}/lib/node_modules $TEMPDIR/assets
      export PATH="${nodeDependencies}/bin:$PATH"

      cd $TEMPDIR/assets
      webpack --config ./webpack.config.js
      cd ..
    '';

    installPhase = ''
      cp -r ./priv/static $out/
    '';

    outputHashAlgo = "sha256";
    outputHashMode = "recursive";
    # nix will complain and tell you the right value to replace this with
    outputHash = lib.fakeSha256;

    impureEnvVars = lib.fetchers.proxyImpureEnvVars;
  };


in packages.mixRelease {
  inherit src pname version mixEnv mixDeps;
  # if you have build time environment variables add them here
  MY_ENV_VAR="my_value";
  preInstall = ''
    mkdir -p ./priv/static
    cp -r ${frontEndFiles} ./priv/static
  '';
}
```

Setup will require the following steps:

- Move your secrets to runtime environment variables. For more information refer to the runtime.exs docs. On a fresh Phoenix build that would mean that both `DATABASE_URL` and `SECRET_KEY` need to

be moved to `runtime.exs`.

- `cd assets` and `nix-shell -p node2nix --run node2nix --development` will generate a Nix expression containing your frontend dependencies
- commit and push those changes
- you can now `nix-build .`
- To run the release, set the `RELEASE_TMP` environment variable to a directory that your program has write access to. It will be used to store the BEAM settings.

15.3.5.1.5. Example of creating a service for an Elixir - Phoenix project

In order to create a service with your release, you could add a `service.nix` in your project with the following

```
{config, pkgs, lib, ...}:

let
  release = pkgs.callPackage ./default.nix;
  release_name = "app";
  working_directory = "/home/app";
in
{
  systemd.services.${release_name} = {
    wantedBy = [ "multi-user.target" ];
    after = [ "network.target" "postgresql.service" ];
    requires = [ "network-online.target" "postgresql.service" ];
    description = "my app";
    environment = {
      # RELEASE_TMP is used to write the state of the
      # VM configuration when the system is running
      # it needs to be a writable directory
      RELEASE_TMP = working_directory;
      # can be generated in an elixir console with
      # Base.encode32(:crypto.strong_rand_bytes(32))
      RELEASE_COOKIE = "my_cookie";
      MY_VAR = "my_var";
    };
    serviceConfig = {
      Type = "exec";
      DynamicUser = true;
      WorkingDirectory = working_directory;
      # Implied by DynamicUser, but just to emphasize due to RELEASE_TMP
```

```
      PrivateTmp = true;
      ExecStart = ''
        ${release}/bin/${release_name} start
      '';
      ExecStop = ''
        ${release}/bin/${release_name} stop
      '';
      ExecReload = ''
        ${release}/bin/${release_name} restart
      '';
      Restart = "on-failure";
      RestartSec = 5;
      StartLimitBurst = 3;
      StartLimitInterval = 10;
    };
    # disksup requires bash
    path = [ pkgs.bash ];
  };

  environment.systemPackages = [ release ];
}
```

15.3.6. How to Develop

15.3.6.1. Creating a Shell

Usually, we need to create a `shell.nix` file and do our development inside of the environment specified therein. Just install your version of Erlang and any other interpreters, and then use your normal build tools. As an example with Elixir:

```
{ pkgs ? import "<nixpkgs"> {} }:

with pkgs;

let

  elixir = beam.packages.erlangR22.elixir_1_9;

in
mkShell {
  buildInputs = [ elixir ];

  ERL_INCLUDE_PATH="${erlang}/lib/erlang/usr/include";
```

```
}
```

15.3.6.1.1. Elixir - Phoenix project

Here is an example `shell.nix`.

```
with import <nixpkgs> { };

let
  # define packages to install
  basePackages = [
    git
    # replace with beam.packages.erlang.elixir_1_11 if you need
    beam.packages.erlang.elixir
    nodejs
    postgresql_13
    # only used for frontend dependencies
    # you are free to use yarn2nix as well
    nodePackages.node2nix
    # formatting js file
    nodePackages.prettier
  ];

  inputs = basePackages ++ lib.optionals stdenv.isLinux [ inotify-tools ]
    ++ lib.optionals stdenv.isDarwin
    (with darwin.apple_sdk.frameworks; [ CoreFoundation CoreServices ]);

  # define shell startup command
  hooks = ''
    # this allows mix to work on the local directory
    mkdir -p .nix-mix .nix-hex
    export MIX_HOME=$PWD/.nix-mix
    export HEX_HOME=$PWD/.nix-mix
    export PATH=$MIX_HOME/bin:$HEX_HOME/bin:$PATH
    # TODO: not sure how to make hex available without installing it afterw
    mix local.hex --if-missing
    export LANG=en_US.UTF-8
    export ERL_AFLAGS="-kernel shell_history enabled"

    # postges related
    # keep all your db data in a folder inside the project
    export PGDATA="$PWD/db"
```

```
    # phoenix related env vars
    export POOL_SIZE=15
    export DB_URL="postgresql://postgres:postgres@localhost:5432/db"
    export PORT=4000
    export MIX_ENV=dev
    # add your project env vars here, word readable in the nix store.
    export ENV_VAR="your_env_var"
  '';

in mkShell {
  buildInputs = inputs;
  shellHook = hooks;
}
```

Initializing the project will require the following steps:

- create the db directory `initdb ./db` (inside your mix project folder)
- create the postgres user `createuser postgres -ds`
- create the db `createdb db`
- start the postgres instance `pg_ctl -l "$PGDATA/server.log" start`
- add the `/db` folder to your `.gitignore`
- you can start your phoenix server and get a shell with `iex -S mix phx.server`

15.4. Bower

Bower is a package manager for web site front-end components. Bower packages (comprising of build artefacts and sometimes sources) are stored in `git` repositories, typically on Github. The package registry is run by the Bower team with package metadata coming from the `bower.json` file within each package.

The end result of running Bower is a `bower_components` directory which can be included in the web app's build process.

Bower can be run interactively, by installing `nodePackages.bower`. More interestingly, the Bower components can be declared in a Nix derivation, with the help of `nodePackages.bower2nix`.

15.4.1. bower2nix usage

Suppose you have a `bower.json` with the following contents:

15.4.1.1. Example bower.json

```
  "name": "my-web-app",
  "dependencies": {
    "angular": "~1.5.0",
    "bootstrap": "~3.3.6"
  }
```

Running `bower2nix` will produce something like the following output:

```
{ fetchbower, buildEnv }:
buildEnv { name = "bower-env"; ignoreCollisions = true; paths = [
  (fetchbower "angular" "1.5.3" "~1.5.0" "1749xb0firxdra4rzadm4q9x90v6pzkbd
  (fetchbower "bootstrap" "3.3.6" "~3.3.6" "1vvqlpbfcy0k5pncfjaiskj3y6scwif
  (fetchbower "jquery" "2.2.2" "1.9.1 - 2" "10sp5h98sqwk90y4k6hbdviwqzvzwqf
];
```

Using the `bower2nix` command line arguments, the output can be redirected to a file. A name like `bower-packages.nix` would be fine.

The resulting derivation is a union of all the downloaded Bower packages (and their dependencies). To use it, they still need to be linked together by Bower, which is where `buildBowerComponents` is useful.

15.4.2. buildBowerComponents function

The function is implemented in pkgs/development/bower-modules/generic/default.nix.

15.4.2.1. Example buildBowerComponents

```
      bowerComponents = buildBowerComponents {
        name = "my-web-app";
        generated = ./bower-packages.nix; ❶
        src = myWebApp; ❷
      };
```

In "buildBowerComponents" example the following arguments are of special significance to the function:

❶ `generated` specifies the file which was created by **`bower2nix`**.

❷ `src` is your project's sources. It needs to contain a `bower.json` file.

`buildBowerComponents` will run Bower to link together the output of `bower2nix`, resulting in a `bower_components` directory which can be used.

Here is an example of a web frontend build process using `gulp`. You might use `grunt`, or anything else.

15.4.2.2. Example build script (gulpfile.js)

```
var gulp = require('gulp');

gulp.task('default', [], function () {
  gulp.start('build');
});

gulp.task('build', [], function () {
  console.log("Just a dummy gulp build");
  gulp
    .src(["./bower_components/**/*"])
    .pipe(gulp.dest("./gulpdist/"));
});
```

15.4.2.3. Example Full example — default.nix

```
      { myWebApp ? { outPath = ./.; name = "myWebApp"; }
      , pkgs ? import <nixpkgs> {}
      }:

      pkgs.stdenv.mkDerivation {
        name = "my-web-app-frontend";
        src = myWebApp;

        buildInputs = [ pkgs.nodePackages.gulp ];

        bowerComponents = pkgs.buildBowerComponents { ❶
          name = "my-web-app";
          generated = ./bower-packages.nix;
          src = myWebApp;
        };

        buildPhase = ''
          cp --reflink=auto --no-preserve=mode -R $bowerComponents/bower_co
          export HOME=$PWD ❸
          ${pkgs.nodePackages.gulp}/bin/gulp build ❹
        '';

        installPhase = "mv gulpdist $out";
      }
```

A few notes about Full example — `default.nix`:

❶ The result of `buildBowerComponents` is an input to the frontend build.

❷ Whether to symlink or copy the `bower_components` directory depends on the build tool in use. In this case a copy is used to avoid **`gulp`** silliness with permissions.

❸ **`gulp`** requires `HOME` to refer to a writeable directory.

❹ The actual build command. Other tools could be used.

15.4.3. Troubleshooting

15.4.3.1. ENOCACHE errors from buildBowerComponents

This means that Bower was looking for a package version which doesn't exist in the generated `bower-packages.nix`.

If `bower.json` has been updated, then run `bower2nix` again.

It could also be a bug in `bower2nix` or `fetchbower`. If possible, try reformulating the version specification in `bower.json`.

15.5. Coq and coq packages

15.5.1. Coq derivation: coq

The Coq derivation is overridable through the `coq.override overrides`, where overrides is an attribute set which contains the arguments to override. We recommend overriding either of the following

- `version` (optional, defaults to the latest version of Coq selected for nixpkgs, see `pkgs/top-level/coq-packages` to witness this choice), which follows the conventions explained in the `coqPackages` section below,
- `customOCamlPackage` (optional, defaults to `null`, which lets Coq choose a version automatically), which can be set to any of the ocaml packages attribute of `ocaml-ng` (such as `ocaml-ng.ocamlPackages_4_10` which is the default for Coq 8.11 for example).
- `coq-version` (optional, defaults to the short version e.g. "8.10"), is a version number of the form "x.y" that indicates which Coq's version build behavior to mimic when using a source which is not a release. E.g.

```
coq.override { version = "d370a9d1328a4e1cdb9d02ee032f605a9d94ec7a"; coq-version = "8.10"; }
```

.

15.5.2. Coq packages attribute sets: coqPackages

The recommended way of defining a derivation for a Coq library, is to use the `coqPackages.mkCoqDerivation` function, which is essentially a specialization of `mkDerivation` taking into account most of the specifics of Coq libraries. The following attributes are supported:

- `pname` (required) is the name of the package,
- `version` (optional, defaults to `null`), is the version to fetch and build, this attribute is interpreted in several ways depending on its type and pattern:
  - if it is a known released version string, i.e. from the `release` attribute below, the according release is picked, and the `version` attribute of the resulting derivation is set to this release string,
  - if it is a majorMinor `"x.y"` prefix of a known released version (as defined above), then the latest `"x.y.z"` known released version is selected (for the ordering given by `versionAtLeast`),
  - if it is a path or a string representing an absolute path (i.e. starting with `"/"`), the provided path is selected as a source, and the `version` attribute of the resulting derivation is set to `"dev"`,
  - if it is a string of the form `owner:branch` then it tries to download the `branch` of owner `owner` for a project of the same name using the same vcs, and the `version` attribute of the resulting derivation is set to `"dev"`, additionally if the owner is not provided (i.e. if the `owner:` prefix is missing), it defaults to the original owner of the package (see below),
  - if it is a string of the form `"#N"`, and the domain is github, then it tries to download the current head of the pull request `#N` from github,
- `defaultVersion` (optional). Coq libraries may be compatible with some specific versions of Coq only. The `defaultVersion` attribute is used when no `version` is provided (or if `version = null`) to select the version of the library to use by default, depending on the context. This selection will mainly depend on a `coq` version number but also possibly on other packages versions (e.g. `mathcomp`). If its value ends up to be `null`, the package is marked for removal in end-user `coqPackages` attribute set.
- `release` (optional, defaults to `{}`), lists all the known releases of the library and for each of them provides an attribute set with at least a `sha256` attribute (you may put the empty string `""` in order to automatically insert a fake sha256, this will trigger an error which will allow you to find the correct sha256), each attribute set of the list of releases also takes optional overloading arguments for the fetcher as below (i.e.`domain`, `owner`, `repo`, `rev` assuming the default fetcher is used) and optional overrides for the result of the fetcher (i.e. `version` and `src`).
- `fetcher` (optional, defaults to a generic fetching mechanism supporting github or gitlab based infrastructures), is a function that takes at least an `owner`, a `repo`, a `rev`, and a `sha256` and returns an attribute set with a `version` and `src`.

- `repo` (optional, defaults to the value of `pname`),
- `owner` (optional, defaults to `"coq-community"`).
- `domain` (optional, defaults to `"github.com"`), domains including the strings `"github"` or `"gitlab"` in their names are automatically supported, otherwise, one must change the `fetcher` argument to support them (cf `pkgs/development/coq-modules/heq/default.nix` for an example),
- `releaseRev` (optional, defaults to `(v: v)`), provides a default mapping from release names to revision hashes/branch names/tags,
- `displayVersion` (optional), provides a way to alter the computation of `name` from `pname`, by explaining how to display version numbers,
- `namePrefix` (optional), provides a way to alter the computation of `name` from `pname`, by explaining which dependencies must occur in `name`,
- `extraBuildInputs` (optional), by default `buildInputs` just contains `coq`, this allows to add more build inputs,
- `mlPlugin` (optional, defaults to `false`). Some extensions (plugins) might require OCaml and sometimes other OCaml packages. Standard dependencies can be added by setting the current option to `true`. For a finer grain control, the `coq.ocamlPackages` attribute can be used in `extraBuildInputs` to depend on the same package set Coq was built against.
- `useDune2ifVersion` (optional, default to `(x: false)` uses Dune2 to build the package if the provided predicate evaluates to true on the version, e.g. `useDune2if = versions.isGe "1.1"` will use dune if the version of the package is greater or equal to `"1.1"`,
- `useDune2` (optional, defaults to `false`) uses Dune2 to build the package if set to true, the presence of this attribute overrides the behavior of the previous one.
- `enableParallelBuilding` (optional, defaults to `true`), since it is activated by default, we provide a way to disable it.
- `extraInstallFlags` (optional), allows to extend `installFlags` which initializes the variable `COQMF_COQLIB` so as to install in the proper subdirectory. Indeed Coq libraries should be installed in `$(out)/lib/coq/${coq.coq-version}/user-contrib/`. Such directories are automatically added to the `$COQPATH` environment variable by the hook defined in the Coq derivation.
- `setCOQBIN` (optional, defaults to `true`), by default, the environment variable `$COQBIN` is set to the current Coq's binary, but one can disable this behavior by setting it to `false`,
- `useMelquiondRemake` (optional, default to `null`) is an attribute set, which, if given, overloads the `preConfigurePhases`, `configureFlags`, `buildPhase`, and `installPhase` attributes of the derivation for a specific use in libraries using `remake` as set up by Guillaume Melquiond for `flocq`, `gappalib`, `interval`, and `coquelicot` (see the corresponding derivation for concrete examples of use of this option). For backward compatibility, the attribute `useMelquiondRemake.logpath` must

be set to the logical root of the library (otherwise, one can pass `useMelquiondRemake = {}` to activate this without backward compatibility).

- `dropAttrs`, `keepAttrs`, `dropDerivationAttrs` are all optional and allow to tune which attribute is added or removed from the final call to `mkDerivation`.

It also takes other standard `mkDerivation` attributes, they are added as such, except for `meta` which extends an automatically computed `meta` (where the `platform` is the same as `coq` and the homepage is automatically computed).

Here is a simple package example. It is a pure Coq library, thus it depends on Coq. It builds on the Mathematical Components library, thus it also takes some `mathcomp` derivations as `extraBuildInputs`.

```
{ lib, mkCoqDerivation, version ? null
, coq, mathcomp, mathcomp-finmap, mathcomp-bigenough }:
with lib; mkCoqDerivation {
  /* namePrefix leads to e.g. `name = coq8.11-mathcomp1.11-multinomials-1.5
  namePrefix = [ "coq" "mathcomp" ];
  pname = "multinomials";
  owner = "math-comp";
  inherit version;
  defaultVersion =  with versions; switch [ coq.version mathcomp.version ]
      { cases = [ (range "8.7" "8.12")  "1.11.0" ];             out = "1.5.
      { cases = [ (range "8.7" "8.11")  (range "1.8" "1.10") ]; out = "1.5.
      { cases = [ (range "8.7" "8.10")  (range "1.8" "1.10") ]; out = "1.4"
      { cases = [ "8.6"                 (range "1.6" "1.7") ];  out = "1.1"
    ] null;
  release = {
    "1.5.2".sha256 = "15aspf3jfykp1xgsxf8knqkxv8aav2p39c2fyirw7pwsfbsv2c4s"
    "1.5.1".sha256 = "13nlfm2wqripaq671gakz5mn4r0xwm0646araxv0nh455p9ndjs3"
    "1.5.0".sha256 = "064rvc0x5g7y1a0nip6ic91vzmq52alf6in2bc2dmss6dmzv90hw"
    "1.5.0".rev    = "1.5";
    "1.4".sha256   = "0vnkirs8iqsv8s59yx1fvg1nkwnzydl42z3scya1xp1b48qkgn0p"
    "1.3".sha256   = "0l3vi5n094nx3qmy66hsv867fnqm196r8v605kpk24gl0aa57wh4"
    "1.2".sha256   = "1mh1w339dslgv4f810xr1b8v2w7rpx6fgk9pz96q0fyq49fw2xcq"
    "1.1".sha256   = "1q8alsm89wkc0lhcvxlyn0pd8rbl2nnxg81zyrabpz610qqjqc3s"
    "1.0".sha256   = "1qmbxp1h81cy3imh627pznmng0kvv37k4hrwi2faa101s6bcx55m"
  };

  propagatedBuildInputs =
    [ mathcomp.ssreflect mathcomp.algebra mathcomp-finmap mathcomp-bigenoug

  meta = {
```

```
    description = "A Coq/SSReflect Library for Monoidal Rings and Multinomi
    license = licenses.cecill-c;
  };
}
```

15.6. Crystal

15.6.1. Building a Crystal package

This section uses Mint as an example for how to build a Crystal package.

If the Crystal project has any dependencies, the first step is to get a `shards.nix` file encoding those. Get a copy of the project and go to its root directory such that its `shard.lock` file is in the current directory, then run `crystal2nix` in it

```
$ git clone https://github.com/mint-lang/mint
$ cd mint
$ git checkout 0.5.0
$ nix-shell -p crystal2nix --run crystal2nix
```

This should have generated a `shards.nix` file.

Next create a Nix file for your derivation and use `pkgs.crystal.buildCrystalPackage` as follows:

```
with import <nixpkgs> {};
crystal.buildCrystalPackage rec {
  pname = "mint";
  version = "0.5.0";

  src = fetchFromGitHub {
    owner = "mint-lang";
    repo = "mint";
    rev = version;
    sha256 = "0vxbx38c390rd2ysvbwgh89v2232sh5rbsp3nk9wzb70jybpslvl";
  };

  # Insert the path to your shards.nix file here
  shardsFile = ./shards.nix;

  ...
}
```

This won't build anything yet, because we haven't told it what files build. We can specify a mapping from

binary names to source files with the `crystalBinaries` attribute. The project's compilation instructions should show this. For Mint, the binary is called "mint", which is compiled from the source file `src/mint.cr`, so we'll specify this as follows:

```
crystalBinaries.mint.src = "src/mint.cr";

# ...
```

Additionally you can override the default `crystal build` options (which are currently `--release --progress --no-debug --verbose`) with

```
crystalBinaries.mint.options = [ "--release" "--verbose" ];
```

Depending on the project, you might need additional steps to get it to compile successfully. In Mint's case, we need to link against openssl, so in the end the Nix file looks as follows:

```
with import <nixpkgs> {};
crystal.buildCrystalPackage rec {
  version = "0.5.0";
  pname = "mint";
  src = fetchFromGitHub {
    owner = "mint-lang";
    repo = "mint";
    rev = version;
    sha256 = "0vxbx38c390rd2ysvbwgh89v2232sh5rbsp3nk9wzb70jybpslvl";
  };

  shardsFile = ./shards.nix;
  crystalBinaries.mint.src = "src/mint.cr";

  buildInputs = [ openssl ];
}
```

15.7. Dhall

The Nixpkgs support for Dhall assumes some familiarity with Dhall's language support for importing Dhall expressions, which is documented here:

- `dhall-lang.org` - Installing packages

15.7.1. Remote imports

Nixpkgs bypasses Dhall's support for remote imports using Dhall's semantic integrity checks. Specifically,

any Dhall import can be protected by an integrity check like:

```
https://prelude.dhall-lang.org/v20.1.0/package.dhall
  sha256:26b0ef498663d269e4dc6a82b0ee289ec565d683ef4c00d0ebdd25333a5a3c98
```

… and if the import is cached then the interpreter will load the import from cache instead of fetching the URL.

Nixpkgs uses this trick to add all of a Dhall expression's dependencies into the cache so that the Dhall interpreter never needs to resolve any remote URLs. In fact, Nixpkgs uses a Dhall interpreter with remote imports disabled when packaging Dhall expressions to enforce that the interpreter never resolves a remote import. This means that Nixpkgs only supports building Dhall expressions if all of their remote imports are protected by semantic integrity checks.

Instead of remote imports, Nixpkgs uses Nix to fetch remote Dhall code. For example, the Prelude Dhall package uses `pkgs.fetchFromGitHub` to fetch the `dhall-lang` repository containing the Prelude. Relying exclusively on Nix to fetch Dhall code ensures that Dhall packages built using Nix remain pure and also behave well when built within a sandbox.

15.7.2. Packaging a Dhall expression from scratch

We can illustrate how Nixpkgs integrates Dhall by beginning from the following trivial Dhall expression with one dependency (the Prelude):

```
-- ./true.dhall

let Prelude = https://prelude.dhall-lang.org/v20.1.0/package.dhall

in  Prelude.Bool.not False
```

As written, this expression cannot be built using Nixpkgs because the expression does not protect the Prelude import with a semantic integrity check, so the first step is to freeze the expression using `dhall freeze`, like this:

```
$ dhall freeze --inplace ./true.dhall
```

… which gives us:

```
-- ./true.dhall

let Prelude =
      https://prelude.dhall-lang.org/v20.1.0/package.dhall
        sha256:26b0ef498663d269e4dc6a82b0ee289ec565d683ef4c00d0ebdd25333a5a
```

```
in  Prelude.Bool.not False
```

To package that expression, we create a `./true.nix` file containing the following specification for the Dhall package:

```
# ./true.nix

{ buildDhallPackage, Prelude }:

buildDhallPackage {
  name = "true";
  code = ./true.dhall;
  dependencies = [ Prelude ];
  source = true;
}
```

… and we complete the build by incorporating that Dhall package into the `pkgs.dhallPackages` hierarchy using an overlay, like this:

```
# ./example.nix

let
  nixpkgs = builtins.fetchTarball {
    url    = "https://github.com/NixOS/nixpkgs/archive/94b2848559b12a8ed1fe
    sha256 = "1pbl4c2dsaz2lximgd31m96jwbps6apn3anx8cvvhk1gl9rkg107";
  };

  dhallOverlay = self: super: {
    true = self.callPackage ./true.nix { };
  };

  overlay = self: super: {
    dhallPackages = super.dhallPackages.override (old: {
      overrides =
        self.lib.composeExtensions (old.overrides or (_: _: {})) dhallOverl
    });
  };

  pkgs = import nixpkgs { config = {}; overlays = [ overlay ]; };

in
```

```
  pkgs
```

... which we can then build using this command:

```
$ nix build --file ./example.nix dhallPackages.true
```

15.7.3. Contents of a Dhall package

The above package produces the following directory tree:

```
$ tree -a ./result
result
├── .cache
│   └── dhall
│       └── 122027abdeddfe8503496adeb623466caa47da5f63abd2bc6fa19f6cfcb73ec
├── binary.dhall
└── source.dhall
```

... where:

- `source.dhall` contains the result of interpreting our Dhall package:

  ```
  $ cat ./result/source.dhall
  True
  ```

- The `.cache` subdirectory contains one binary cache product encoding the same result as `source.dhall`:

  ```
  $ dhall decode < ./result/.cache/dhall/122027abdeddfe8503496adeb623466caa
  True
  ```

- `binary.dhall` contains a Dhall expression which handles fetching and decoding the same cache product:

  ```
  $ cat ./result/binary.dhall
  missing sha256:27abdeddfe8503496adeb623466caa47da5f63abd2bc6fa19f6cfcb73e
  $ cp -r ./result/.cache .cache

  $ chmod -R u+w .cache

  $ XDG_CACHE_HOME=.cache dhall --file ./result/binary.dhall
  ```

```
True
```

The `source.dhall` file is only present for packages that specify `source = true;`. By default, Dhall packages omit the `source.dhall` in order to conserve disk space when they are used exclusively as dependencies. For example, if we build the Prelude package it will only contain the binary encoding of the expression:

```
$ nix build --file ./example.nix dhallPackages.Prelude

$ tree -a result
result
├── .cache
│   └── dhall
│       └── 122026b0ef498663d269e4dc6a82b0ee289ec565d683ef4c00d0ebdd25333a5
└── binary.dhall

2 directories, 2 files
```

Typically, you only specify `source = true;` for the top-level Dhall expression of interest (such as our example `true.nix` Dhall package). However, if you wish to specify `source = true` for all Dhall packages, then you can amend the Dhall overlay like this:

```
dhallOverrides = self: super: {
  # Enable source for all Dhall packages
  buildDhallPackage =
    args: super.buildDhallPackage (args // { source = true; });

  true = self.callPackage ./true.nix { };
};
```

… and now the Prelude will contain the fully decoded result of interpreting the Prelude:

```
$ nix build --file ./example.nix dhallPackages.Prelude

$ tree -a result
result
├── .cache
│   └── dhall
│       └── 122026b0ef498663d269e4dc6a82b0ee289ec565d683ef4c00d0ebdd25333a5
├── binary.dhall
└── source.dhall
```

```
$ cat ./result/source.dhall
{ Bool =
  { and =
      \(_ : List Bool) ->
        List/fold Bool _ Bool (\(_ : Bool) -> \(_ : Bool) -> _@1 && _) True
  , build = \(_ : Type -> _ -> _@1 -> _@2) -> _ Bool True False
  , even =
      \(_ : List Bool) ->
        List/fold Bool _ Bool (\(_ : Bool) -> \(_ : Bool) -> _@1 == _) True
  , fold =
      \(_ : Bool) ->
…
```

15.7.4. Packaging functions

We already saw an example of using `buildDhallPackage` to create a Dhall package from a single file, but most Dhall packages consist of more than one file and there are two derived utilities that you may find more useful when packaging multiple files:

- `buildDhallDirectoryPackage` - build a Dhall package from a local directory
- `buildDhallGitHubPackage` - build a Dhall package from a GitHub repository

The `buildDhallPackage` is the lowest-level function and accepts the following arguments:

- `name`: The name of the derivation
- `dependencies`: Dhall dependencies to build and cache ahead of time
- `code`: The top-level expression to build for this package

  Note that the `code` field accepts an arbitrary Dhall expression. You're not limited to just a file.
- `source`: Set to `true` to include the decoded result as `source.dhall` in the build product, at the expense of requiring more disk space
- `documentationRoot`: Set to the root directory of the package if you want `dhall-docs` to generate documentation underneath the `docs` subdirectory of the build product

The `buildDhallDirectoryPackage` is a higher-level function implemented in terms of `buildDhallPackage` that accepts the following arguments:

- `name`: Same as `buildDhallPackage`
- `dependencies`: Same as `buildDhallPackage`
- `source`: Same as `buildDhallPackage`

- `src`: The directory containing Dhall code that you want to turn into a Dhall package
- `file`: The top-level file (`package.dhall` by default) that is the entrypoint to the rest of the package
- `document`: Set to `true` to generate documentation for the package

The `buildDhallGitHubPackage` is another higher-level function implemented in terms of `buildDhallPackage` that accepts the following arguments:

- `name`: Same as `buildDhallPackage`
- `dependencies`: Same as `buildDhallPackage`
- `source`: Same as `buildDhallPackage`
- `owner`: The owner of the repository
- `repo`: The repository name
- `rev`: The desired revision (or branch, or tag)
- `directory`: The subdirectory of the Git repository to package (if a directory other than the root of the repository)
- `file`: The top-level file (`${directory}/package.dhall` by default) that is the entrypoint to the rest of the package
- `document`: Set to `true` to generate documentation for the package

Additionally, `buildDhallGitHubPackage` accepts the same arguments as `fetchFromGitHub`, such as `sha256` or `fetchSubmodules`.

15.7.5. `dhall-to-nixpkgs`

You can use the `dhall-to-nixpkgs` command-line utility to automate packaging Dhall code. For example:

```
$ nix-env --install --attr haskellPackages.dhall-nixpkgs

$ nix-env --install --attr nix-prefetch-git  # Used by dhall-to-nixpkgs

$ dhall-to-nixpkgs github https://github.com/Gabriel439/dhall-semver.git
{ buildDhallGitHubPackage, Prelude }:
  buildDhallGitHubPackage {
    name = "dhall-semver";
    githubBase = "github.com";
    owner = "Gabriel439";
    repo = "dhall-semver";
    rev = "2d44ae605302ce5dc6c657a1216887fbb96392a4";
```

```
    fetchSubmodules = false;
    sha256 = "0y8shvp8srzbjjpmnsvz9c12ciihnx1szs0yzyi9ashmrjvd0jcz";
    directory = "";
    file = "package.dhall";
    source = false;
    document = false;
    dependencies = [ (Prelude.overridePackage { file = "package.dhall"; })
    }
```

The utility takes care of automatically detecting remote imports and converting them to package dependencies. You can also use the utility on local Dhall directories, too:

```
$ dhall-to-nixpkgs directory ~/proj/dhall-semver
{ buildDhallDirectoryPackage, Prelude }:
  buildDhallDirectoryPackage {
    name = "proj";
    src = /Users/gabriel/proj/dhall-semver;
    file = "package.dhall";
    source = false;
    document = false;
    dependencies = [ (Prelude.overridePackage { file = "package.dhall"; })
    }
```

15.7.6. Overriding dependency versions

Suppose that we change our `true.dhall` example expression to depend on an older version of the Prelude (19.0.0):

```
-- ./true.dhall

let Prelude =
      https://prelude.dhall-lang.org/v19.0.0/package.dhall
        sha256:eb693342eb769f782174157eba9b5924cf8ac6793897fc36a31ccbd6f56d

in  Prelude.Bool.not False
```

If we try to rebuild that expression the build will fail:

```
$ nix build --file ./example.nix dhallPackages.true
builder for '/nix/store/0f1hla7ff1wiaqyk1r2ky4wnhnw114fi-true.drv' failed w
```

```
  Dhall was compiled without the 'with-http' flag.

  The requested URL was: https://prelude.dhall-lang.org/v19.0.0/package.dha


  4│       https://prelude.dhall-lang.org/v19.0.0/package.dhall
  5│         sha256:eb693342eb769f782174157eba9b5924cf8ac6793897fc36a31ccbd

  /nix/store/rsab4y99h14912h4zplqx2iizr5n4rc2-true.dhall:4:7
[1 built (1 failed), 0.0 MiB DL]
error: build of '/nix/store/0f1hla7ff1wiaqyk1r2ky4wnhnw114fi-true.drv' fail
```

… because the default Prelude selected by Nixpkgs revision `94b2848559b12a8ed1fe433084686b2a81123c99is` is version 20.1.0, which doesn't have the same integrity check as version 19.0.0. This means that version 19.0.0 is not cached and the interpreter is not allowed to fall back to importing the URL.

However, we can override the default Prelude version by using `dhall-to-nixpkgs` to create a Dhall package for our desired Prelude:

```
$ dhall-to-nixpkgs github https://github.com/dhall-lang/dhall-lang.git \
    --name Prelude \
    --directory Prelude \
    --rev v19.0.0 \
    > Prelude.nix
```

… and then referencing that package in our Dhall overlay, by either overriding the Prelude globally for all packages, like this:

```
  dhallOverrides = self: super: {
    true = self.callPackage ./true.nix { };

    Prelude = self.callPackage ./Prelude.nix { };
  };
```

… or selectively overriding the Prelude dependency for just the `true` package, like this:

```
  dhallOverrides = self: super: {
    true = self.callPackage ./true.nix {
      Prelude = self.callPackage ./Prelude.nix { };
    };
  };
```

15.7.7. Overrides

You can override any of the arguments to `buildDhallGitHubPackage` or `buildDhallDirectoryPackage` using the `overridePackage` attribute of a package. For example, suppose we wanted to selectively enable `source = true` just for the Prelude. We can do that like this:

```
dhallOverrides = self: super: {
  Prelude = super.Prelude.overridePackage { source = true; };

  …
};
```

15.8. Emscripten

Emscripten: An LLVM-to-JavaScript Compiler

This section of the manual covers how to use `emscripten` in nixpkgs.

Minimal requirements:

- nix
- nixpkgs

Modes of use of `emscripten`:

- **Imperative usage** (on the command line):

  If you want to work with `emcc`, `emconfigure` and `emmake` as you are used to from Ubuntu and similar distributions you can use these commands:

  - `nix-env -i emscripten`
  - `nix-shell -p emscripten`

- **Declarative usage**:

  This mode is far more power full since this makes use of `nix` for dependency management of emscripten libraries and targets by using the `mkDerivation` which is implemented by `pkgs.emscriptenStdenv` and `pkgs.buildEmscriptenPackage`. The source for the packages is in `pkgs/top-level/emscripten-packages.nix` and the abstraction behind it in `pkgs/development/em-modules/generic/default.nix`.

  - build and install all packages:
    - `nix-env -iA emscriptenPackages`

- dev-shell for zlib implementation hacking:
  - `nix-shell -A emscriptenPackages.zlib`

15.8.1. Imperative usage

A few things to note:

- `export EMCC_DEBUG=2` is nice for debugging
- `~/.emscripten`, the build artifact cache sometimes creates issues and needs to be removed from time to time

15.8.2. Declarative usage

Let's see two different examples from `pkgs/top-level/emscripten-packages.nix`:

- `pkgs.zlib.override`
- `pkgs.buildEmscriptenPackage`

Both are interesting concepts.

A special requirement of the `pkgs.buildEmscriptenPackage` is the `doCheck = true` is a default meaning that each emscriptenPackage requires a `checkPhase` implemented.

- Use `export EMCC_DEBUG=2` from within a emscriptenPackage's `phase` to get more detailed debug output what is going wrong.
- ~/.emscripten cache is requiring us to set `HOME=$TMPDIR` in individual phases. This makes compilation slower but also makes it more deterministic.

15.8.2.1. Usage 1: pkgs.zlib.override

This example uses `zlib` from nixpkgs but instead of compiling **C** to **ELF** it compiles **C** to **JS** since we were using `pkgs.zlib.override` and changed stdenv to `pkgs.emscriptenStdenv`. A few adaptions and hacks were set in place to make it working. One advantage is that when `pkgs.zlib` is updated, it will automatically update this package as well. However, this can also be the downside…

See the `zlib` example:

```
zlib = (pkgs.zlib.override {
  stdenv = pkgs.emscriptenStdenv;
}).overrideDerivation
(old: rec {
  buildInputs = old.buildInputs ++ [ pkg-config ];
  # we need to reset this setting!
  NIX_CFLAGS_COMPILE="";
  configurePhase = ''
```

```
    # FIXME: Some tests require writing at $HOME
    HOME=$TMPDIR
    runHook preConfigure

    #export EMCC_DEBUG=2
    emconfigure ./configure --prefix=$out --shared

    runHook postConfigure
  '';
  dontStrip = true;
  outputs = [ "out" ];
  buildPhase = ''
    emmake make
  '';
  installPhase = ''
    emmake make install
  '';
  checkPhase = ''
    echo "================= testing zlib using node =================="

    echo "Compiling a custom test"
    set -x
    emcc -O2 -s EMULATE_FUNCTION_POINTER_CASTS=1 test/example.c -DZ_SOLO \
    libz.so.${old.version} -I . -o example.js

    echo "Using node to execute the test"
    ${pkgs.nodejs}/bin/node ./example.js

    set +x
    if [ $? -ne 0 ]; then
      echo "test failed for some reason"
      exit 1;
    else
      echo "it seems to work! very good."
    fi
    echo "================= /testing zlib using node =================="
  '';

  postPatch = pkgs.lib.optionalString pkgs.stdenv.isDarwin ''
    substituteInPlace configure \
      --replace '/usr/bin/libtool' 'ar' \
      --replace 'AR="libtool"' 'AR="ar"' \
      --replace 'ARFLAGS="-o"' 'ARFLAGS="-r"'
```

```
  '';
});
```

15.8.2.2. Usage 2: pkgs.buildEmscriptenPackage

This `xmlmirror` example features a emscriptenPackage which is defined completely from this context and no `pkgs.zlib.override` is used.

```
xmlmirror = pkgs.buildEmscriptenPackage rec {
  name = "xmlmirror";

  buildInputs = [ pkg-config autoconf automake libtool gnumake libxml2 node
  nativeBuildInputs = [ pkg-config zlib ];

  src = pkgs.fetchgit {
    url = "https://gitlab.com/odfplugfest/xmlmirror.git";
    rev = "4fd7e86f7c9526b8f4c1733e5c8b45175860a8fd";
    sha256 = "1jasdqnbdnb83wbcnyrp32f36w3xwhwp0wq8lwwmhqagxrij1r4b";
  };

  configurePhase = ''
    rm -f fastXmlLint.js*
    # a fix for ERROR:root:For asm.js, TOTAL_MEMORY must be a multiple of 1
    # https://gitlab.com/odfplugfest/xmlmirror/issues/8
    sed -e "s/TOTAL_MEMORY=234217728/TOTAL_MEMORY=268435456/g" -i Makefile.
    # https://github.com/kripken/emscripten/issues/6344
    # https://gitlab.com/odfplugfest/xmlmirror/issues/9
    sed -e "s/\$(JSONC_LDFLAGS) \$(ZLIB_LDFLAGS) \$(LIBXML20_LDFLAGS)/\$(JS
    # https://gitlab.com/odfplugfest/xmlmirror/issues/11
    sed -e "s/-o fastXmlLint.js/-s EXTRA_EXPORTED_RUNTIME_METHODS='[\"ccall
  '';

  buildPhase = ''
    HOME=$TMPDIR
    make -f Makefile.emEnv
  '';

  outputs = [ "out" "doc" ];

  installPhase = ''
    mkdir -p $out/share
    mkdir -p $doc/share/${name}
```

```
    cp Demo* $out/share
    cp -R codemirror-5.12 $out/share
    cp fastXmlLint.js* $out/share
    cp *.xsd $out/share
    cp *.js $out/share
    cp *.xhtml $out/share
    cp *.html $out/share
    cp *.json $out/share
    cp *.rng $out/share
    cp README.md $doc/share/${name}
  '';
  checkPhase = ''

  '';
};
```

15.8.2.3. Declarative debugging

Use `nix-shell -I nixpkgs=/some/dir/nixpkgs -A emscriptenPackages.libz` and from there you can go trough the individual steps. This makes it easy to build a good `unit test` or list the files of the project.

1. `nix-shell -I nixpkgs=/some/dir/nixpkgs -A emscriptenPackages.libz`
2. `cd /tmp/`
3. `unpackPhase`
4. cd libz-1.2.3
5. `configurePhase`
6. `buildPhase`
7. ... happy hacking...

15.8.3. Summary

Using this toolchain makes it easy to leverage `nix` from NixOS, MacOSX or even Windows (WSL+ubuntu+nix). This toolchain is reproducible, behaves like the rest of the packages from nixpkgs and contains a set of well working examples to learn and adapt from.

If in trouble, ask the maintainers.

15.9. GNOME

15.9.1. Packaging GNOME applications

Programs in the GNOME universe are written in various languages but they all use GObject-based libraries like GLib, GTK or GStreamer. These libraries are often modular, relying on looking into certain directories to find their modules. However, due to Nix's specific file system organization, this will fail without our intervention. Fortunately, the libraries usually allow overriding the directories through environment variables, either natively or thanks to a patch in nixpkgs. Wrapping the executables to ensure correct paths are available to the application constitutes a significant part of packaging a modern desktop application. In this section, we will describe various modules needed by such applications, environment variables needed to make the modules load, and finally a script that will do the work for us.

15.9.1.1. Settings

GSettings API is often used for storing settings. GSettings schemas are required, to know the type and other metadata of the stored values. GLib looks for `glib-2.0/schemas/gschemas.compiled` files inside the directories of `XDG_DATA_DIRS`.

On Linux, GSettings API is implemented using dconf backend. You will need to add `dconf` GIO module to `GIO_EXTRA_MODULES` variable, otherwise the `memory` backend will be used and the saved settings will not be persistent.

Last you will need the dconf database D-Bus service itself. You can enable it using `programs.dconf.enable`.

Some applications will also require `gsettings-desktop-schemas` for things like reading proxy configuration or user interface customization. This dependency is often not mentioned by upstream, you should grep for `org.gnome.desktop` and `org.gnome.system` to see if the schemas are needed.

15.9.1.2. GIO modules

GLib's GIO library supports several extension points. Notably, they allow:

- implementing settings backends (already mentioned)
- adding TLS support
- proxy settings
- virtual file systems

The modules are typically installed to `lib/gio/modules/` directory of a package and you need to add them to `GIO_EXTRA_MODULES` if you need any of those features.

In particular, we recommend:

- adding `dconf.lib` for any software on Linux that reads GSettings (even transitivily through e.g. GTK's file manager)
- adding `glib-networking` for any software that accesses network using GIO or libsoup – glib-networking contains a module that implements TLS support and loads system-wide proxy settings

To allow software to use various virtual file systems, `gvfs` package can be also added. But that is usually an optional feature so we typically use `gvfs` from the system (e.g. installed globally using NixOS module).

15.9.1.3. GdkPixbuf loaders

GTK applications typically use GdkPixbuf to load images. But `gdk-pixbuf` package only supports basic bitmap formats like JPEG, PNG or TIFF, requiring to use third-party loader modules for other formats. This is especially painful since GTK itself includes SVG icons, which cannot be rendered without a loader provided by `librsvg`.

Unlike other libraries mentioned in this section, GdkPixbuf only supports a single value in its controlling environment variable `GDK_PIXBUF_MODULE_FILE`. It is supposed to point to a cache file containing information about the available loaders. Each loader package will contain a `lib/gdk-pixbuf-2.0/2.10.0/loaders.cache` file describing the default loaders in `gdk-pixbuf` package plus the loader contained in the package itself. If you want to use multiple third-party loaders, you will need to create your own cache file manually. Fortunately, this is pretty rare as not many loaders exist.

`gdk-pixbuf` contains a setup hook that sets `GDK_PIXBUF_MODULE_FILE` from dependencies but as mentioned in further section, it is pretty limited. Loaders should propagate this setup hook.

15.9.1.4. Icons

When an application uses icons, an icon theme should be available in `XDG_DATA_DIRS` during runtime. The package for the default, icon-less hicolor-icon-theme (should be propagated by every icon theme) contains a setup hook that will pick up icon themes from `buildInputs` and pass it to our wrapper. Unfortunately, relying on that would mean every user has to download the theme included in the package expression no matter their preference. For that reason, we leave the installation of icon theme on the user. If you use one of the desktop environments, you probably already have an icon theme installed.

To avoid costly file system access when locating icons, GTK, as well as Qt, can rely on `icon-theme.cache` files from the themes' top-level directories. These files are generated using `gtk-update-icon-cache`, which is expected to be run whenever an icon is added or removed to an icon theme (typically an application icon into `hicolor` theme) and some programs do indeed run this after icon installation. However, since packages are installed into their own prefix by Nix, this would lead to conflicts. For that reason, `gtk3` provides a setup hook that will clean the file from installation. Since most applications only ship their own icon that will be loaded on start-up, it should not affect them too much. On the other hand, icon themes are much larger and more widely used so we need to cache them. Because we recommend installing icon themes globally, we will generate the cache files from all packages in a profile using a NixOS module. You can enable the cache generation using `gtk.iconCache.enable` option if your desktop environment does not already do that.

15.9.1.5. Packaging icon themes

Icon themes may inherit from other icon themes. The inheritance is specified using the `Inherits` key in

the `index.theme` file distributed with the icon theme. According to the icon theme specification, icons not provided by the theme are looked for in its parent icon themes. Therefore the parent themes should be installed as dependencies for a more complete experience regarding the icon sets used.

The package `hicolor-icon-theme` provides a setup hook which makes symbolic links for the parent themes into the directory `share/icons` of the current theme directory in the nix store, making sure they can be found at runtime. For that to work the packages providing parent icon themes should be listed as propagated build dependencies, together with `hicolor-icon-theme`.

Also make sure that `icon-theme.cache` is installed for each theme provided by the package, and set `dontDropIconThemeCache` to `true` so that the cache file is not removed by the `gtk3` setup hook.

15.9.1.6. GTK Themes

Previously, a GTK theme needed to be in `XDG_DATA_DIRS`. This is no longer necessary for most programs since GTK incorporated Adwaita theme. Some programs (for example, those designed for elementary HIG) might require a special theme like `pantheon.elementary-gtk-theme`.

15.9.1.7. GObject introspection typelibs

GObject introspection allows applications to use C libraries in other languages easily. It does this through `typelib` files searched in `GI_TYPELIB_PATH`.

15.9.1.8. Various plug-ins

If your application uses GStreamer or Grilo, you should set `GST_PLUGIN_SYSTEM_PATH_1_0` and `GRL_PLUGIN_PATH`, respectively.

15.9.2. Onto `wrapGAppsHook`

Given the requirements above, the package expression would become messy quickly:

```
preFixup = ''
  for f in $(find $out/bin/ $out/libexec/ -type f -executable); do
    wrapProgram "$f" \
      --prefix GIO_EXTRA_MODULES : "${getLib dconf}/lib/gio/modules" \
      --prefix XDG_DATA_DIRS : "$out/share" \
      --prefix XDG_DATA_DIRS : "$out/share/gsettings-schemas/${name}" \
      --prefix XDG_DATA_DIRS : "${gsettings-desktop-schemas}/share/gsetting
      --prefix XDG_DATA_DIRS : "${hicolor-icon-theme}/share" \
      --prefix GI_TYPELIB_PATH : "${lib.makeSearchPath "lib/girepository-1.
  done
'';
```

Fortunately, there is `wrapGAppsHook`. It works in conjunction with other setup hooks that populate

environment variables, and it will then wrap all executables in `bin` and `libexec` directories using said variables.

For convenience, it also adds `dconf.lib` for a GIO module implementing a GSettings backend using `dconf`, `gtk3` for GSettings schemas, and `librsvg` for GdkPixbuf loader to the closure. There is also `wrapGAppsHook4`, which replaces GTK 3 with GTK 4. And in case you are packaging a program without a graphical interface, you might want to use `wrapGAppsNoGuiHook`, which runs the same script as `wrapGAppsHook` but does not bring `gtk3` and `librsvg` into the closure.

- `wrapGAppsHook` itself will add the package's `share` directory to `XDG_DATA_DIRS`.
- `glib` setup hook will populate `GSETTINGS_SCHEMAS_PATH` and then `wrapGAppsHook` will prepend it to `XDG_DATA_DIRS`.
- `gdk-pixbuf` setup hook will populate `GDK_PIXBUF_MODULE_FILE` with the path to biggest `loaders.cache` file from the dependencies containing GdkPixbuf loaders. This works fine when there are only two packages containing loaders (`gdk-pixbuf` and e.g. `librsvg`) – it will choose the second one, reasonably expecting that it will be bigger since it describes extra loader in addition to the default ones. But when there are more than two loader packages, this logic will break. One possible solution would be constructing a custom cache file for each package containing a program like `services/x11/gdk-pixbuf.nix` NixOS module does. `wrapGAppsHook` copies the `GDK_PIXBUF_MODULE_FILE` environment variable into the produced wrapper.
- One of `gtk3`'s setup hooks will remove `icon-theme.cache` files from package's icon theme directories to avoid conflicts. Icon theme packages should prevent this with `dontDropIconThemeCache = true;`.
- `dconf.lib` is a dependency of `wrapGAppsHook`, which then also adds it to the `GIO_EXTRA_MODULES` variable.
- `hicolor-icon-theme`'s setup hook will add icon themes to `XDG_ICON_DIRS` which is prepended to `XDG_DATA_DIRS` by `wrapGAppsHook`.
- `gobject-introspection` setup hook populates `GI_TYPELIB_PATH` variable with `lib/girepository-1.0` directories of dependencies, which is then added to wrapper by `wrapGAppsHook`. It also adds `share` directories of dependencies to `XDG_DATA_DIRS`, which is intended to promote GIR files but it also pollutes the closures of packages using `wrapGAppsHook`.

  > **Warning:** The setup hook currently does not work in expressions with `strictDeps` enabled, like Python packages. In those cases, you will need to disable it with `strictDeps = false;`.

- Setup hooks of `gst_all_1.gstreamer` and `grilo` will populate the `GST_PLUGIN_SYSTEM_PATH_1_0` and `GRL_PLUGIN_PATH` variables, respectively, which will then be added to the wrapper by `wrapGAppsHook`.

You can also pass additional arguments to `makeWrapper` using `gappsWrapperArgs` in `preFixup`

hook:

```
preFixup = ''
  gappsWrapperArgs+=(
    # Thumbnailers
    --prefix XDG_DATA_DIRS : "${gdk-pixbuf}/share"
    --prefix XDG_DATA_DIRS : "${librsvg}/share"
    --prefix XDG_DATA_DIRS : "${shared-mime-info}/share"
  )
'';
```

15.9.3. Updating GNOME packages

Most GNOME package offer `updateScript`, it is therefore possible to update to latest source tarball by running `nix-shell maintainers/scripts/update.nix --argstr package gnome.nautilus` or even en masse with `nix-shell maintainers/scripts/update.nix --argstr path gnome`. Read the package's `NEWS` file to see what changed.

15.9.4. Frequently encountered issues

15.9.4.1. `GLib-GIO-ERROR **: 06:04:50.903: No GSettings schemas are installed on the system`

There are no schemas available in `XDG_DATA_DIRS`. Temporarily add a random package containing schemas like `gsettings-desktop-schemas` to `buildInputs`. `glib` and `wrapGAppsHook` setup hooks will take care of making the schemas available to application and you will see the actual missing schemas with the next error. Or you can try looking through the source code for the actual schemas used.

15.9.4.2. `GLib-GIO-ERROR **: 06:04:50.903: Settings schema 'org.gnome.foo' is not installed`

Package is missing some GSettings schemas. You can find out the package containing the schema with `nix-locate org.gnome.foo.gschema.xml` and let the hooks handle the wrapping as above.

15.9.4.3. When using `wrapGAppsHook` with special derivers you can end up with double wrapped binaries.

This is because derivers like `python.pkgs.buildPythonApplication` or `qt5.mkDerivation` have setup-hooks automatically added that produce wrappers with makeWrapper. The simplest way to workaround that is to disable the `wrapGAppsHook` automatic wrapping with `dontWrapGApps = true;` and pass the arguments it intended to pass to makeWrapper to another.

In the case of a Python application it could look like:

```
python3.pkgs.buildPythonApplication {
  pname = "gnome-music";
  version = "3.32.2";

  nativeBuildInputs = [
    wrapGAppsHook
    gobject-introspection
    ...
  ];

  dontWrapGApps = true;

  # Arguments to be passed to `makeWrapper`, only used by buildPython*
  preFixup = ''
    makeWrapperArgs+=("'${gappsWrapperArgs[@]}")
  '';
}
```

And for a QT app like:

```
mkDerivation {
  pname = "calibre";
  version = "3.47.0";

  nativeBuildInputs = [
    wrapGAppsHook
    qmake
    ...
  ];

  dontWrapGApps = true;

  # Arguments to be passed to `makeWrapper`, only used by qt5's mkDerivatio
  preFixup = ''
    qtWrapperArgs+=("'${gappsWrapperArgs[@]}")
  '';
}
```

15.9.4.4. I am packaging a project that cannot be wrapped, like a library or

GNOME Shell extension.

You can rely on applications depending on the library setting the necessary environment variables but that is often easy to miss. Instead we recommend to patch the paths in the source code whenever possible. Here are some examples:

- Replacing a `GI_TYPELIB_PATH` in GNOME Shell extension – we are using `substituteAll` to include the path to a typelib into a patch.
- The following examples are hardcoding GSettings schema paths. To get the schema paths we use the functions
  - `glib.getSchemaPath` Takes a nix package attribute as an argument.
  - `glib.makeSchemaPath` Takes a package output like `$out` and a derivation name. You should use this if the schemas you need to hardcode are in the same derivation.

  Hard-coding GSettings schema path in Vala plug-in (dynamically loaded library) – here, `substituteAll` cannot be used since the schema comes from the same package preventing us from pass its path to the function, probably due to a Nix bug.

  Hard-coding GSettings schema path in C library – nothing special other than using Coccinelle patch to generate the patch itself.

15.9.4.5. I need to wrap a binary outside `bin` and `libexec` directories.

You can manually trigger the wrapping with `wrapGApp` in `preFixup` phase. It takes a path to a program as a first argument; the remaining arguments are passed directly to `wrapProgram` function.

15.10. Go

15.10.1. Go modules

The function `buildGoModule` builds Go programs managed with Go modules. It builds a Go Modules through a two phase build:

- An intermediate fetcher derivation. This derivation will be used to fetch all of the dependencies of the Go module.
- A final derivation will use the output of the intermediate derivation to build the binaries and produce the final output.

15.10.1.1. Example for `buildGoModule`

In the following is an example expression using `buildGoModule`, the following arguments are of special significance to the function:

- `vendorSha256`: is the hash of the output of the intermediate fetcher derivation. `vendorSha256` can

also take `null` as an input. When `null` is used as a value, rather than fetching the dependencies and vendoring them, we use the vendoring included within the source repo. If you'd like to not have to update this field on dependency changes, run `go mod vendor` in your source repo and set `vendorSha256 = null;`

- `runVend`: runs the vend command to generate the vendor directory. This is useful if your code depends on c code and go mod tidy does not include the needed sources to build.

```
pet = buildGoModule rec {
  pname = "pet";
  version = "0.3.4";

  src = fetchFromGitHub {
    owner = "knqyf263";
    repo = "pet";
    rev = "v${version}";
    sha256 = "0m2fzpqxk7hrbxsgqplkg7h2p7gv6s1miymv3gvw0cz039skag0s";
  };

  vendorSha256 = "1879j77k96684wi554rkjxydrj8g3hpp0kvxz03sd8dmwr3lh83j";

  runVend = true;

  meta = with lib; {
    description = "Simple command-line snippet manager, written in Go";
    homepage = "https://github.com/knqyf263/pet";
    license = licenses.mit;
    maintainers = with maintainers; [ kalbasit ];
    platforms = platforms.linux ++ platforms.darwin;
  };
}
```

15.10.2. `buildGoPackage` (legacy)

The function `buildGoPackage` builds legacy Go programs, not supporting Go modules.

15.10.2.1. Example for `buildGoPackage`

In the following is an example expression using buildGoPackage, the following arguments are of special significance to the function:

- `goPackagePath` specifies the package's canonical Go import path.
- `goDeps` is where the Go dependencies of a Go program are listed as a list of package source identified

by Go import path. It could be imported as a separate `deps.nix` file for readability. The dependency data structure is described below.

```
deis = buildGoPackage rec {
  pname = "deis";
  version = "1.13.0";

  goPackagePath = "github.com/deis/deis";

  src = fetchFromGitHub {
    owner = "deis";
    repo = "deis";
    rev = "v${version}";
    sha256 = "1qv9lxqx7m18029lj8cw3k7jngvxs4iciwrypdy0gd2nnghc68sw";
  };

  goDeps = ./deps.nix;
}
```

The `goDeps` attribute can be imported from a separate `nix` file that defines which Go libraries are needed and should be included in `GOPATH` for `buildPhase`:

```
# deps.nix
[ # goDeps is a list of Go dependencies.
  {
    # goPackagePath specifies Go package import path.
    goPackagePath = "gopkg.in/yaml.v2";
    fetch = {
      # `fetch type` that needs to be used to get package source.
      # If `git` is used there should be `url`, `rev` and `sha256` defined
      type = "git";
      url = "https://gopkg.in/yaml.v2";
      rev = "a83829b6f1293c91addabc89d0571c246397bbf4";
      sha256 = "1m4dsmk90sbi17571h6pld44zxz7jc4lrnl4f27dpd1l8g5xvjhh";
    };
  }
  {
    goPackagePath = "github.com/docopt/docopt-go";
    fetch = {
      type = "git";
      url = "https://github.com/docopt/docopt-go";
      rev = "784ddc588536785e7299f7272f39101f7faccc3f";
      sha256 = "0wwz48jl9fvl1iknvn9dqr4gfy1qs03gxaikrxxp9gry6773v3sj";
```

```
    };
  }
]
```

To extract dependency information from a Go package in automated way use go2nix. It can produce complete derivation and `goDeps` file for Go programs.

You may use Go packages installed into the active Nix profiles by adding the following to your ~/.bashrc:

```
for p in $NIX_PROFILES; do
    GOPATH="$p/share/go:$GOPATH"
done
```

15.10.3. Attributes used by the builders

Both `buildGoModule` and `buildGoPackage` can be tweaked to behave slightly differently, if the following attributes are used:

15.10.3.1. `buildFlagsArray` and `buildFlags`:

These attributes set build flags supported by `go build`. We recommend using `buildFlagsArray`. The most common use case of these attributes is to make the resulting executable aware of its own version. For example:

```
buildFlagsArray = [
  # Note: single quotes are not needed.
  "-ldflags=-X main.Version=${version} -X main.Commit=${version}"
];
```

```
buildFlagsArray = ''
  -ldflags=
  -X main.Version=${version}
  -X main.Commit=${version}
'';
```

15.10.3.2. `deleteVendor`

Removes the pre-existing vendor directory. This should only be used if the dependencies included in the vendor folder are broken or incomplete.

15.10.3.3. `subPackages`

Limits the builder from building child packages that have not been listed. If `subPackages` is not specified, all child packages will be built.

15.11. Haskell

The documentation for the Haskell infrastructure is published at https://haskell4nix.readthedocs.io/. The source code for that site lives in the `doc/` sub-directory of the `cabal2nix` Git repository and changes can be submitted there.

15.12. Idris

15.12.1. Installing Idris

The easiest way to get a working idris version is to install the `idris` attribute:

```
$ # On NixOS
$ nix-env -i nixos.idris
$ # On non-NixOS
$ nix-env -i nixpkgs.idris
```

This however only provides the `prelude` and `base` libraries. To install idris with additional libraries, you can use the `idrisPackages.with-packages` function, e.g. in an overlay in `~/.config/nixpkgs/overlays/my-idris.nix`:

```
self: super: {
  myIdris = with self.idrisPackages; with-packages [ contrib pruviloj ];
}
```

And then:

```
$ # On NixOS
$ nix-env -iA nixos.myIdris
$ # On non-NixOS
$ nix-env -iA nixpkgs.myIdris
```

To see all available Idris packages:

```
$ # On NixOS
$ nix-env -qaPA nixos.idrisPackages
$ # On non-NixOS
$ nix-env -qaPA nixpkgs.idrisPackages
```

Similarly, entering a `nix-shell`:

```
$ nix-shell -p 'idrisPackages.with-packages (with idrisPackages; [ contrib
```

15.12.2. Starting Idris with library support

To have access to these libraries in idris, call it with an argument `-p <library name>` for each library:

```
$ nix-shell -p 'idrisPackages.with-packages (with idrisPackages; [ contrib
[nix-shell:~]$ idris -p contrib -p pruviloj
```

A listing of all available packages the Idris binary has access to is available via `--listlibs`:

```
$ idris --listlibs
00prelude-idx.ibc
pruviloj
base
contrib
prelude
00pruviloj-idx.ibc
00base-idx.ibc
00contrib-idx.ibc
```

15.12.3. Building an Idris project with Nix

As an example of how a Nix expression for an Idris package can be created, here is the one for `idrisPackages.yaml`:

```
{ lib
, build-idris-package
, fetchFromGitHub
, contrib
, lightyear
}:
build-idris-package  {
  name = "yaml";
  version = "2018-01-25";

  # This is the .ipkg file that should be built, defaults to the package na
  # In this case it should build `Yaml.ipkg` instead of `yaml.ipkg`
  # This is only necessary because the yaml packages ipkg file is
```

```
  # different from its package name here.
  ipkgName = "Yaml";
  # Idris dependencies to provide for the build
  idrisDeps = [ contrib lightyear ];

  src = fetchFromGitHub {
    owner = "Heather";
    repo = "Idris.Yaml";
    rev = "5afa51ffc839844862b8316faba3bafa15656db4";
    sha256 = "1g4pi0swmg214kndj85hj50ccmckni7piprsxfdzdfhg87s0avw7";
  };

  meta = with lib; {
    description = "Idris YAML lib";
    homepage = "https://github.com/Heather/Idris.Yaml";
    license = licenses.mit;
    maintainers = [ maintainers.brainrape ];
  };
}
```

Assuming this file is saved as `yaml.nix`, it's buildable using

```
$ nix-build -E '(import <nixpkgs> {}).idrisPackages.callPackage ./yaml.nix
```

Or it's possible to use

```
with import <nixpkgs> {};

{
  yaml = idrisPackages.callPackage ./yaml.nix {};
}
```

in another file (say `default.nix`) to be able to build it with

```
$ nix-build -A yaml
```

15.12.4. Passing options to `idris` commands

The `build-idris-package` function provides also optional input values to set additional options for the used `idris` commands.

Specifically, you can set `idrisBuildOptions`, `idrisTestOptions`, `idrisInstallOptions` and `idrisDocOptions` to provide additional options to the `idris` command respectively when building, testing, installing and generating docs for your package.

For example you could set

```
build-idris-package {
  idrisBuildOptions = [ "--log" "1" "--verbose" ]

  ...
}
```

to require verbose output during `idris` build phase.

15.13. iOS

This component is basically a wrapper/workaround that makes it possible to expose an Xcode installation as a Nix package by means of symlinking to the relevant executables on the host system.

Since Xcode can't be packaged with Nix, nor we can publish it as a Nix package (because of its license) this is basically the only integration strategy making it possible to do iOS application builds that integrate with other components of the Nix ecosystem

The primary objective of this project is to use the Nix expression language to specify how iOS apps can be built from source code, and to automatically spawn iOS simulator instances for testing.

This component also makes it possible to use Hydra, the Nix-based continuous integration server to regularly build iOS apps and to do wireless ad-hoc installations of enterprise IPAs on iOS devices through Hydra.

The Xcode build environment implements a number of features.

15.13.1. Deploying a proxy component wrapper exposing Xcode

The first use case is deploying a Nix package that provides symlinks to the Xcode installation on the host system. This package can be used as a build input to any build function implemented in the Nix expression language that requires Xcode.

```
let
  pkgs = import <nixpkgs> {};

  xcodeenv = import ./xcodeenv {
    inherit (pkgs) stdenv;
  };
in
xcodeenv.composeXcodeWrapper {
```

```
  version = "9.2";
  xcodeBaseDir = "/Applications/Xcode.app";
}
```

By deploying the above expression with `nix-build` and inspecting its content you will notice that several Xcode-related executables are exposed as a Nix package:

```
$ ls result/bin
lrwxr-xr-x  1 sander  staff  94  1 jan  1970 Simulator -> /Applications/Xco
lrwxr-xr-x  1 sander  staff  17  1 jan  1970 codesign -> /usr/bin/codesign
lrwxr-xr-x  1 sander  staff  17  1 jan  1970 security -> /usr/bin/security
lrwxr-xr-x  1 sander  staff  21  1 jan  1970 xcode-select -> /usr/bin/xcode
lrwxr-xr-x  1 sander  staff  61  1 jan  1970 xcodebuild -> /Applications/Xc
lrwxr-xr-x  1 sander  staff  14  1 jan  1970 xcrun -> /usr/bin/xcrun
```

15.13.2. Building an iOS application

We can build an iOS app executable for the simulator, or an IPA/xcarchive file for release purposes, e.g. ad-hoc, enterprise or store installations, by executing the `xcodeenv.buildApp {}` function:

```
let
  pkgs = import <nixpkgs> {};

  xcodeenv = import ./xcodeenv {
    inherit (pkgs) stdenv;
  };
in
xcodeenv.buildApp {
  name = "MyApp";
  src = ./myappsources;
  sdkVersion = "11.2";

  target = null; # Corresponds to the name of the app by default
  configuration = null; # Release for release builds, Debug for debug build
  scheme = null; # -scheme will correspond to the app name by default
  sdk = null; # null will set it to 'iphonesimulator` for simulator builds
  xcodeFlags = "";

  release = true;
  certificateFile = ./mycertificate.p12;
  certificatePassword = "secret";
  provisioningProfile = ./myprovisioning.profile;
```

```
  signMethod = "ad-hoc"; # 'enterprise' or 'store'
  generateIPA = true;
  generateXCArchive = false;

  enableWirelessDistribution = true;
  installURL = "/installipa.php";
  bundleId = "mycompany.myapp";
  appVersion = "1.0";

  # Supports all xcodewrapper parameters as well
  xcodeBaseDir = "/Applications/Xcode.app";
}
```

The above function takes a variety of parameters: * The `name` and `src` parameters are mandatory and specify the name of the app and the location where the source code resides * `sdkVersion` specifies which version of the iOS SDK to use.

It also possile to adjust the `xcodebuild` parameters. This is only needed in rare circumstances. In most cases the default values should suffice:

- Specifies which `xcodebuild` target to build. By default it takes the target that has the same name as the app.
- The `configuration` parameter can be overridden if desired. By default, it will do a debug build for the simulator and a release build for real devices.
- The `scheme` parameter specifies which `-scheme` parameter to propagate to `xcodebuild`. By default, it corresponds to the app name.
- The `sdk` parameter specifies which SDK to use. By default, it picks `iphonesimulator` for simulator builds and `iphoneos` for release builds.
- The `xcodeFlags` parameter specifies arbitrary command line parameters that should be propagated to `xcodebuild`.

By default, builds are carried out for the iOS simulator. To do release builds (builds for real iOS devices), you must set the `release` parameter to `true`. In addition, you need to set the following parameters:

- `certificateFile` refers to a P12 certificate file.
- `certificatePassword` specifies the password of the P12 certificate.
- `provisioningProfile` refers to the provision profile needed to sign the app
- `signMethod` should refer to `ad-hoc` for signing the app with an ad-hoc certificate, `enterprise` for enterprise certificates and `app-store` for App store certificates.
- `generateIPA` specifies that we want to produce an IPA file (this is probably what you want)

- `generateXCArchive` specifies thet we want to produce an xcarchive file.

When building IPA files on Hydra and when it is desired to allow iOS devices to install IPAs by browsing to the Hydra build products page, you can enable the `enableWirelessDistribution` parameter.

When enabled, you need to configure the following options:

- The `installURL` parameter refers to the URL of a PHP script that composes the `itms-services://` URL allowing iOS devices to install the IPA file.
- `bundleId` refers to the bundle ID value of the app
- `appVersion` refers to the app's version number

To use wireless adhoc distributions, you must also install the corresponding PHP script on a web server (see section: "Installing the PHP script for wireless ad hoc installations from Hydra" for more information).

In addition to the build parameters, you can also specify any parameters that the `xcodeenv.composeXcodeWrapper {}` function takes. For example, the `xcodeBaseDir` parameter can be overridden to refer to a different Xcode version.

15.13.3. Spawning simulator instances

In addition to building iOS apps, we can also automatically spawn simulator instances:

```
let
  pkgs = import <nixpkgs> {};

  xcodeenv = import ./xcodeenv {
    inherit (pkgs) stdenv;
  };
in
xcode.simulateApp {
  name = "simulate";

  # Supports all xcodewrapper parameters as well
  xcodeBaseDir = "/Applications/Xcode.app";
}
```

The above expression produces a script that starts the simulator from the provided Xcode installation. The script can be started as follows:

```
./result/bin/run-test-simulator
```

By default, the script will show an overview of UDID for all available simulator instances and asks you to pick one. You can also provide a UDID as a command-line parameter to launch an instance automatically:

```
./result/bin/run-test-simulator 5C93129D-CF39-4B1A-955F-15180C3BD4B8
```

You can also extend the simulator script to automatically deploy and launch an app in the requested simulator instance:

```
let
  pkgs = import <nixpkgs> {};

  xcodeenv = import ./xcodeenv {
    inherit (pkgs) stdenv;
  };
in
xcode.simulateApp {
  name = "simulate";
  bundleId = "mycompany.myapp";
  app = xcode.buildApp {
    # ...
  };

  # Supports all xcodewrapper parameters as well
  xcodeBaseDir = "/Applications/Xcode.app";
}
```

By providing the result of an `xcode.buildApp {}` function and configuring the app bundle id, the app gets deployed automatically and started.

15.13.4. Troubleshooting

In some rare cases, it may happen that after a failure, changes are not picked up. Most likely, this is caused by a derived data cache that Xcode maintains. To wipe it you can run:

```
$ rm -rf ~/Library/Developer/Xcode/DerivedData
```

15.14. Java

Ant-based Java packages are typically built from source as follows:

```
stdenv.mkDerivation {
  name = "...";
  src = fetchurl { ... };
```

```
  nativeBuildInputs = [ jdk ant ];

  buildPhase = "ant";
}
```

Note that `jdk` is an alias for the OpenJDK (self-built where available, or pre-built via Zulu). Platforms with OpenJDK not (yet) in Nixpkgs (`Aarch32`, `Aarch64`) point to the (unfree) `oraclejdk`.

JAR files that are intended to be used by other packages should be installed in `$out/share/java`. JDKs have a stdenv setup hook that add any JARs in the `share/java` directories of the build inputs to the `CLASSPATH` environment variable. For instance, if the package `libfoo` installs a JAR named `foo.jar` in its `share/java` directory, and another package declares the attribute

```
buildInputs = [ libfoo ];
nativeBuildInputs = [ jdk ];
```

then `CLASSPATH` will be set to `/nix/store/...-libfoo/share/java/foo.jar`.

Private JARs should be installed in a location like `$out/share/package-name`.

If your Java package provides a program, you need to generate a wrapper script to run it using a JRE. You can use `makeWrapper` for this:

```
nativeBuildInputs = [ makeWrapper ];

installPhase = ''
  mkdir -p $out/bin
  makeWrapper ${jre}/bin/java $out/bin/foo \
    --add-flags "-cp $out/share/java/foo.jar org.foo.Main"
'';
```

Since the introduction of the Java Platform Module System in Java 9, Java distributions typically no longer ship with a general-purpose JRE: instead, they allow generating a JRE with only the modules required for your application(s). Because we can't predict what modules will be needed on a general-purpose system, the default jre package is the full JDK. When building a minimal system/image, you can override the `modules` parameter on `jre_minimal` to build a JRE with only the modules relevant for you:

```
let
  my_jre = pkgs.jre_minimal.override {
    modules = [
      # The modules used by 'something' and 'other' combined:
      "java.base"
      "java.logging"
    ];
```

```
  };
  something = (pkgs.something.override { jre = my_jre; });
  other = (pkgs.other.override { jre = my_jre; });
in
  ...
```

Note all JDKs passthru `home`, so if your application requires environment variables like `JAVA_HOME` being set, that can be done in a generic fashion with the `--set` argument of `makeWrapper`:

```
--set JAVA_HOME ${jdk.home}
```

It is possible to use a different Java compiler than `javac` from the OpenJDK. For instance, to use the GNU Java Compiler:

```
nativeBuildInputs = [ gcj ant ];
```

Here, Ant will automatically use `gij` (the GNU Java Runtime) instead of the OpenJRE.

15.15. User's Guide to Lua Infrastructure

15.15.1. Using Lua

15.15.1.1. Overview of Lua

Several versions of the Lua interpreter are available: luajit, lua 5.1, 5.2, 5.3. The attribute `lua` refers to the default interpreter, it is also possible to refer to specific versions, e.g. `lua5_2` refers to Lua 5.2.

Lua libraries are in separate sets, with one set per interpreter version.

The interpreters have several common attributes. One of these attributes is `pkgs`, which is a package set of Lua libraries for this specific interpreter. E.g., the `busted` package corresponding to the default interpreter is `lua.pkgs.busted`, and the lua 5.2 version is `lua5_2.pkgs.busted`. The main package set contains aliases to these package sets, e.g. `luaPackages` refers to `lua5_1.pkgs` and `lua52Packages` to `lua5_2.pkgs`.

15.15.1.2. Installing Lua and packages

15.15.1.2.1. Lua environment defined in separate `.nix` file

Create a file, e.g. `build.nix`, with the following expression

```
with import <nixpkgs> {};

lua5_2.withPackages (ps: with ps; [ busted luafilesystem ])
```

and install it in your profile with

```
nix-env -if build.nix
```

Now you can use the Lua interpreter, as well as the extra packages (`busted`, `luafilesystem`) that you added to the environment.

15.15.1.2.2. Lua environment defined in `~/.config/nixpkgs/config.nix`

If you prefer to, you could also add the environment as a package override to the Nixpkgs set, e.g. using `config.nix`,

```
{ # ...

  packageOverrides = pkgs: with pkgs; {
    myLuaEnv = lua5_2.withPackages (ps: with ps; [ busted luafilesystem ]);
  };
}
```

and install it in your profile with

```
nix-env -iA nixpkgs.myLuaEnv
```

The environment is installed by referring to the attribute, and considering the `nixpkgs` channel was used.

15.15.1.2.3. Lua environment defined in `/etc/nixos/configuration.nix`

For the sake of completeness, here's another example how to install the environment system-wide.

```
{ # ...

  environment.systemPackages = with pkgs; [
    (lua.withPackages(ps: with ps; [ busted luafilesystem ]))
  ];
}
```

15.15.1.3. How to override a Lua package using overlays?

Use the following overlay template:

```
final: prev:
```

```
{

  lua = prev.lua.override {
    packageOverrides = luaself: luaprev: {

      luarocks-nix = luaprev.luarocks-nix.overrideAttrs(oa: {
        pname = "luarocks-nix";
        src = /home/my_luarocks/repository;
      });
  };

  luaPackages = lua.pkgs;
}
```

15.15.1.4. Temporary Lua environment with `nix-shell`

There are two methods for loading a shell with Lua packages. The first and recommended method is to create an environment with `lua.buildEnv` or `lua.withPackages` and load that. E.g.

```
$ nix-shell -p 'lua.withPackages(ps: with ps; [ busted luafilesystem ])'
```

opens a shell from which you can launch the interpreter

```
[nix-shell:~] lua
```

The other method, which is not recommended, does not create an environment and requires you to list the packages directly,

```
$ nix-shell -p lua.pkgs.busted lua.pkgs.luafilesystem
```

Again, it is possible to launch the interpreter from the shell. The Lua interpreter has the attribute `pkgs` which contains all Lua libraries for that specific interpreter.

15.15.2. Developing with Lua

Now that you know how to get a working Lua environment with Nix, it is time to go forward and start actually developing with Lua. There are two ways to package lua software, either it is on luarocks and most of it can be taken care of by the luarocks2nix converter or the packaging has to be done manually. Let's present the luarocks way first and the manual one in a second time.

15.15.2.1. Packaging a library on luarocks

Luarocks.org is the main repository of lua packages. The site proposes two types of packages, the rockspec and the src.rock (equivalent of a rockspec but with the source). These packages can have different build types such as `cmake`, `builtin` etc .

Luarocks-based packages are generated in pkgs/development/lua-modules/generated-packages.nix from the whitelist maintainers/scripts/luarocks-packages.csv and updated by running maintainers/scripts/update-luarocks-packages.

luarocks2nix is a tool capable of generating nix derivations from both rockspec and src.rock (and favors the src.rock). The automation only goes so far though and some packages need to be customized. These customizations go in `pkgs/development/lua-modules/overrides.nix`. For instance if the rockspec defines `external_dependencies`, these need to be manually added in its rockspec file then it won't work.

You can try converting luarocks packages to nix packages with the command `nix-shell -p luarocks-nix` and then `luarocks nix PKG_NAME`. Nix rely on luarocks to install lua packages, basically it runs: `luarocks make --deps-mode=none --tree $out`

15.15.2.1.1. Packaging a library manually

You can develop your package as you usually would, just don't forget to wrap it within a `toLuaModule` call, for instance

```
mynewlib = toLuaModule ( stdenv.mkDerivation { ... });
```

There is also the `buildLuaPackage` function that can be used when lua modules are not packaged for luarocks. You can see a few examples at `pkgs/top-level/lua-packages.nix`.

15.15.3. Lua Reference

15.15.3.1. Lua interpreters

Versions 5.1, 5.2 and 5.3 of the lua interpreter are available as respectively `lua5_1`, `lua5_2` and `lua5_3`. Luajit is available too. The Nix expressions for the interpreters can be found in `pkgs/development/interpreters/lua-5`.

15.15.3.1.1. Attributes on lua interpreters packages

Each interpreter has the following attributes:

- `interpreter`. Alias for `${pkgs.lua}/bin/lua`.
- `buildEnv`. Function to build lua interpreter environments with extra packages bundled together. See section *lua.buildEnv function* for usage and documentation.
- `withPackages`. Simpler interface to `buildEnv`.

- `pkgs`. Set of Lua packages for that specific interpreter. The package set can be modified by overriding the interpreter and passing `packageOverrides`.

15.15.3.1.2. `buildLuarocksPackage` function

The `buildLuarocksPackage` function is implemented in `pkgs/development/interpreters/lua-5/build-lua-package.nix` The following is an example:

```
luaposix = buildLuarocksPackage {
  pname = "luaposix";
  version = "34.0.4-1";

  src = fetchurl {
    url    = "https://raw.githubusercontent.com/rocks-moonscript-org/moonro
    sha256 = "0yrm5cn2iyd0zjd4liyj27srphvy0gjrjx572swar6zqr4dwjqp2";
  };
  disabled = (luaOlder "5.1") || (luaAtLeast "5.4");
  propagatedBuildInputs = [ bit32 lua std_normalize ];

  meta = with lib; {
    homepage = "https://github.com/luaposix/luaposix/";
    description = "Lua bindings for POSIX";
    maintainers = with maintainers; [ vyp lblasc ];
    license.fullName = "MIT/X11";
  };
};
```

The `buildLuarocksPackage` delegates most tasks to luarocks:

- it adds `luarocks` as an unpacker for `src.rock` files (zip files really).
- configurePhase `writes a temporary luarocks configuration file which location is exported via the environment variable` LUAROCKS_CONFIG`.
- the `buildPhase` does nothing.
- `installPhase` calls `luarocks make --deps-mode=none --tree $out` to build and install the package
- In the `postFixup` phase, the `wrapLuaPrograms` bash function is called to wrap all programs in the `$out/bin/*` directory to include `$PATH` environment variable and add dependent libraries to script's `LUA_PATH` and `LUA_CPATH`.

By default `meta.platforms` is set to the same value as the interpreter unless overridden otherwise.

15.15.3.1.3. `buildLuaApplication` function

The `buildLuaApplication` function is practically the same as `buildLuaPackage`. The difference is that `buildLuaPackage` by default prefixes the names of the packages with the version of the interpreter. Because with an application we're not interested in multiple version the prefix is dropped.

15.15.3.1.4. lua.withPackages function

The `lua.withPackages` takes a function as an argument that is passed the set of lua packages and returns the list of packages to be included in the environment. Using the `withPackages` function, the previous example for the luafilesystem environment can be written like this:

```
with import <nixpkgs> {};

lua.withPackages (ps: [ps.luafilesystem])
```

`withPackages` passes the correct package set for the specific interpreter version as an argument to the function. In the above example, `ps` equals `luaPackages`. But you can also easily switch to using `lua5_2`:

```
with import <nixpkgs> {};

lua5_2.withPackages (ps: [ps.lua])
```

Now, `ps` is set to `lua52Packages`, matching the version of the interpreter.

15.15.3.2. Possible Todos

- export/use version specific variables such as `LUA_PATH_5_2/LUAROCKS_CONFIG_5_2`
- let luarocks check for dependencies via exporting the different rocktrees in temporary config

15.15.3.3. Lua Contributing guidelines

Following rules should be respected:

- Make sure libraries build for all Lua interpreters.
- Commit names of Lua libraries should reflect that they are Lua libraries, so write for example `luaPackages.luafilesystem: 1.11 -> 1.12`.

15.16. Maven

Maven is a well-known build tool for the Java ecosystem however it has some challenges when integrating into the Nix build system.

The following provides a list of common patterns with how to package a Maven project (or any JVM language that can export to Maven) as a Nix package.

For the purposes of this example let's consider a very basic Maven project with the following `pom.xml` with a single dependency on emoji-java.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
  <modelVersion>4.0.0</modelVersion>
  <groupId>io.github.fzakaria</groupId>
  <artifactId>maven-demo</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <name>NixOS Maven Demo</name>

  <dependencies>
    <dependency>
        <groupId>com.vdurmont</groupId>
        <artifactId>emoji-java</artifactId>
        <version>5.1.1</version>
      </dependency>
  </dependencies>
</project>
```

Our main class file will be very simple:

```
import com.vdurmont.emoji.EmojiParser;

public class Main {
  public static void main(String[] args) {
    String str = "NixOS :grinning: is super cool :smiley:!";
    String result = EmojiParser.parseToUnicode(str);
    System.out.println(result);
  }
}
```

You find this demo project at https://github.com/fzakaria/nixos-maven-example

15.16.1. Solving for dependencies

15.16.1.1. buildMaven with NixOS/mvn2nix-maven-plugin

> ⚠ Although `buildMaven` is the "blessed" way within nixpkgs, as of 2020, it hasn't seen much activity in quite a while.

`buildMaven` is an alternative method that tries to follow similar patterns of other programming languages by generating a lock file. It relies on the maven plugin mvn2nix-maven-plugin.

First you generate a `project-info.json` file using the maven plugin.

> This should be executed in the project's source repository or be told which `pom.xml` to execute with.

```
# run this step within the project's source repository
❯ mvn org.nixos.mvn2nix:mvn2nix-maven-plugin:mvn2nix

❯ cat project-info.json | jq | head
{
  "project": {
    "artifactId": "maven-demo",
    "groupId": "org.nixos",
    "version": "1.0",
    "classifier": "",
    "extension": "jar",
    "dependencies": [
      {
        "artifactId": "maven-resources-plugin",
```

This file is then given to the `buildMaven` function, and it returns 2 attributes.

`repo`: A Maven repository that is a symlink farm of all the dependencies found in the `project-info.json`

`build`: A simple derivation that runs through `mvn compile` & `mvn package` to build the JAR. You may use this as inspiration for more complicated derivations.

Here is an example of building the Maven repository

```
{ pkgs ? import <nixpkgs> { } }:
with pkgs;
(buildMaven ./project-info.json).repo
```

The benefit over the *double invocation* as we will see below, is that the */nix/store* entry is a *linkFarm* of every package, so that changes to your dependency set doesn't involve downloading everything from scratch.

```
❯ tree $(nix-build --no-out-link build-maven-repository.nix) | head
/nix/store/g87va52nkc8jzbmi1aqdcf2f109r4dvn-maven-repository
├── antlr
│   └── antlr
│       └── 2.7.2
│           ├── antlr-2.7.2.jar -> /nix/store/d027c8f2cnmj5yrynpbq2s6wmc9cb
│           └── antlr-2.7.2.pom -> /nix/store/mv42fc5gizl8h5g5vpywz1nfiynmz
├── avalon-framework
│   └── avalon-framework
│       └── 4.1.3
│           ├── avalon-framework-4.1.3.jar -> /nix/store/iv5fp3955w3nq28ff9
```

15.16.1.2. Double Invocation

> ⚠ This pattern is the simplest but may cause unnecessary rebuilds due to the output hash changing.

The double invocation is a *simple* way to get around the problem that `nix-build` may be sandboxed and have no Internet connectivity.

It treats the entire Maven repository as a single source to be downloaded, relying on Maven's dependency resolution to satisfy the output hash. This is similar to fetchers like `fetchgit`, except it has to run a Maven build to determine what to download.

The first step will be to build the Maven project as a fixed-output derivation in order to collect the Maven repository – below is an example.

> Traditionally the Maven repository is at `~/.m2/repository`. We will override this to be the `$out` directory.

```
{ lib, stdenv, maven }:
stdenv.mkDerivation {
  name = "maven-repository";
  buildInputs = [ maven ];
  src = ./.; # or fetchFromGitHub, cleanSourceWith, etc
  buildPhase = ''
    mvn package -Dmaven.repo.local=$out
  '';

  # keep only *.{pom,jar,sha1,nbm} and delete all ephemeral files with last
  installPhase = ''
    find $out -type f \
      -name \*.lastUpdated -or \
      -name resolver-status.properties -or \
```

```
      -name _remote.repositories \
      -delete
  '';

  # don't do any fixup
  dontFixup = true;
  outputHashAlgo = "sha256";
  outputHashMode = "recursive";
  # replace this with the correct SHA256
  outputHash = lib.fakeSha256;
}
```

The build will fail, and tell you the expected `outputHash` to place. When you've set the hash, the build will return with a `/nix/store` entry whose contents are the full Maven repository.

> Some additional files are deleted that would cause the output hash to change potentially on subsequent runs.

```
❯ tree $(nix-build --no-out-link double-invocation-repository.nix) | head
/nix/store/8kicxzp98j68xyi9gl6jda67hp3c54fq-maven-repository
├── backport-util-concurrent
│   └── backport-util-concurrent
│       └── 3.1
│           ├── backport-util-concurrent-3.1.pom
│           └── backport-util-concurrent-3.1.pom.sha1
├── classworlds
│   └── classworlds
│       ├── 1.1
│       │   ├── classworlds-1.1.jar
```

If your package uses *SNAPSHOT* dependencies or *version ranges*; there is a strong likelihood that over-time your output hash will change since the resolved dependencies may change. Hence this method is less recommended then using `buildMaven`.

15.16.2. Building a JAR

Regardless of which strategy is chosen above, the step to build the derivation is the same.

```
{ stdenv, maven, callPackage }:
# pick a repository derivation, here we will use buildMaven
let repository = callPackage ./build-maven-repository.nix { };
in stdenv.mkDerivation rec {
```

```
  pname = "maven-demo";
  version = "1.0";

  src = builtins.fetchTarball "https://github.com/fzakaria/nixos-maven-exam
  buildInputs = [ maven ];

  buildPhase = ''
    echo "Using repository ${repository}"
    mvn --offline -Dmaven.repo.local=${repository} package;
  '';

  installPhase = ''
    install -Dm644 target/${pname}-${version}.jar $out/share/java
  '';
}
```

We place the library in `$out/share/java` since JDK package has a *stdenv setup hook* that adds any JARs in the `share/java` directories of the build inputs to the CLASSPATH environment.

```
❯ tree $(nix-build --no-out-link build-jar.nix)
/nix/store/7jw3xdfagkc2vw8wrsdv68qpsnrxgvky-maven-demo-1.0
└── share
    └── java
        └── maven-demo-1.0.jar

2 directories, 1 file
```

15.16.3. Runnable JAR

The previous example builds a `jar` file but that's not a file one can run.

You need to use it with `java -jar $out/share/java/output.jar` and make sure to provide the required dependencies on the classpath.

The following explains how to use `makeWrapper` in order to make the derivation produce an executable that will run the JAR file you created.

We will use the same repository we built above (either *double invocation* or *buildMaven*) to setup a CLASSPATH for our JAR.

The following two methods are more suited to Nix then building an UberJar which may be the more traditional approach.

15.16.3.1. CLASSPATH

This is ideal if you are providing a derivation for *nixpkgs* and don't want to patch the project's `pom.xml`.

We will read the Maven repository and flatten it to a single list. This list will then be concatenated with the *CLASSPATH* separator to create the full classpath.

We make sure to provide this classpath to the `makeWrapper`.

```
{ stdenv, maven, callPackage, makeWrapper, jre }:
let
  repository = callPackage ./build-maven-repository.nix { };
in stdenv.mkDerivation rec {
  pname = "maven-demo";
  version = "1.0";

  src = builtins.fetchTarball
    "https://github.com/fzakaria/nixos-maven-example/archive/main.tar.gz";
  buildInputs = [ maven makeWrapper ];

  buildPhase = ''
    echo "Using repository ${repository}"
    mvn --offline -Dmaven.repo.local=${repository} package;
  '';

  installPhase = ''
    mkdir -p $out/bin

    classpath=$(find ${repository} -name "*.jar" -printf ':%h/%f');
    install -Dm644 target/${pname}-${version}.jar $out/share/java
    # create a wrapper that will automatically set the classpath
    # this should be the paths from the dependency derivation
    makeWrapper ${jre}/bin/java $out/bin/${pname} \
          --add-flags "-classpath $out/share/java/${pname}-${version}.jar:'
          --add-flags "Main"
  '';
}
```

15.16.3.2. MANIFEST file via Maven Plugin

This is ideal if you are the project owner and want to change your `pom.xml` to set the CLASSPATH within it.

Augment the `pom.xml` to create a JAR with the following manifest:

```
<build>
  <plugins>
    <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
            <archive>
                <manifest>
                    <addClasspath>true</addClasspath>
                    <classpathPrefix>../../repository/</classpathPrefix>
                    <classpathLayoutType>repository</classpathLayoutType>
                    <mainClass>Main</mainClass>
                </manifest>
                <manifestEntries>
                    <Class-Path>.</Class-Path>
                </manifestEntries>
            </archive>
        </configuration>
    </plugin>
  </plugins>
</build>
```

The above plugin instructs the JAR to look for the necessary dependencies in the `lib/` relative folder. The layout of the folder is also in the *maven repository* style.

```
❯ unzip -q -c $(nix-build --no-out-link runnable-jar.nix)/share/java/maven-

Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Built-By: nixbld
Class-Path: . ../../repository/com/vdurmont/emoji-java/5.1.1/emoji-jav
 a-5.1.1.jar ../../repository/org/json/json/20170516/json-20170516.jar
Created-By: Apache Maven 3.6.3
Build-Jdk: 1.8.0_265
Main-Class: Main
```

We will modify the derivation above to add a symlink to our repository so that it's accessible to our JAR during the `installPhase`.

```
{ stdenv, maven, callPackage, makeWrapper, jre }:
# pick a repository derivation, here we will use buildMaven
```

```
let repository = callPackage ./build-maven-repository.nix { };
in stdenv.mkDerivation rec {
  pname = "maven-demo";
  version = "1.0";

  src = builtins.fetchTarball
    "https://github.com/fzakaria/nixos-maven-example/archive/main.tar.gz";
  buildInputs = [ maven makeWrapper ];

  buildPhase = ''
    echo "Using repository ${repository}"
    mvn --offline -Dmaven.repo.local=${repository} package;
  '';

  installPhase = ''
    mkdir -p $out/bin

    # create a symbolic link for the repository directory
    ln -s ${repository} $out/repository

    install -Dm644 target/${pname}-${version}.jar $out/share/java
    # create a wrapper that will automatically set the classpath
    # this should be the paths from the dependency derivation
    makeWrapper ${jre}/bin/java $out/bin/${pname} \
          --add-flags "-jar $out/share/java/${pname}-${version}.jar"
  '';
}
```

Our script produces a dependency on `jre` rather than `jdk` to restrict the runtime closure necessary to run the application.

This will give you an executable shell-script that launches your JAR with all the dependencies available.

```
❯ tree $(nix-build --no-out-link runnable-jar.nix)
/nix/store/8d4c3ibw8ynsn01ibhyqmc1zhzz75s26-maven-demo-1.0
├── bin
│   └── maven-demo
├── repository -> /nix/store/g87va52nkc8jzbmi1aqdcf2f109r4dvn-maven-reposit
└── share
    └── java
        └── maven-demo-1.0.jar

❯ $(nix-build --no-out-link --option tarball-ttl 1 runnable-jar.nix)/bin/ma
```

```
NixOS 😀 is super cool 😃!
```

15.17. Node.js

The `pkgs/development/node-packages` folder contains a generated collection of NPM packages that can be installed with the Nix package manager.

As a rule of thumb, the package set should only provide *end user* software packages, such as command-line utilities. Libraries should only be added to the package set if there is a non-NPM package that requires it.

When it is desired to use NPM libraries in a development project, use the `node2nix` generator directly on the `package.json` configuration file of the project.

The package set provides support for the official stable Node.js versions. The latest stable LTS release in `nodePackages`, as well as the latest stable Current release in `nodePackages_latest`.

If your package uses native addons, you need to examine what kind of native build system it uses. Here are some examples:

- `node-gyp`
- `node-gyp-builder`
- `node-pre-gyp`

After you have identified the correct system, you need to override your package expression while adding in build system as a build input. For example, `dat` requires `node-gyp-build`, so we override its expression in `default.nix`:

```
dat = super.dat.override {
  buildInputs = [ self.node-gyp-build pkgs.libtool pkgs.autoconf pkgs.a
  meta.broken = since "12";
};
```

To add a package from NPM to nixpkgs:

1. Modify `pkgs/development/node-packages/node-packages.json` to add, update or remove package entries to have it included in `nodePackages` and `nodePackages_latest`.
2. Run the script: `(cd pkgs/development/node-packages && ./generate.sh)`.
3. Build your new package to test your changes:
   `cd /path/to/nixpkgs && nix-build -A nodePackages.<new-or-updated-package>`
   . To build against the latest stable Current Node.js version (e.g. 14.x):

```
nix-build -A nodePackages_latest.<new-or-updated-package>
```

4. Add and commit all modified and generated files.

For more information about the generation process, consult the [README.md](#) file of the `node2nix` tool.

15.18. OCaml

OCaml libraries should be installed in `$(out)/lib/ocaml/${ocaml.version}/site-lib/`. Such directories are automatically added to the `$OCAMLPATH` environment variable when building another package that depends on them or when opening a `nix-shell`.

Given that most of the OCaml ecosystem is now built with dune, nixpkgs includes a convenience build support function called `buildDunePackage` that will build an OCaml package using dune, OCaml and findlib and any additional dependencies provided as `buildInputs` or `propagatedBuildInputs`.

Here is a simple package example. It defines an (optional) attribute `minimumOCamlVersion` that will be used to throw a descriptive evaluation error if building with an older OCaml is attempted. It uses the `fetchFromGitHub` fetcher to get its source. It sets the `doCheck` (optional) attribute to `true` which means that tests will be run with `dune runtest -p angstrom` after the build (`dune build -p angstrom`) is complete. It uses `alcotest` as a build input (because it is needed to run the tests) and `bigstringaf` and `result` as propagated build inputs (thus they will also be available to libraries depending on this library). The library will be installed using the `angstrom.install` file that dune generates.

```nix
{ lib
, fetchFromGitHub
, buildDunePackage
, alcotest
, result
, bigstringaf
}:

buildDunePackage rec {
  pname = "angstrom";
  version = "0.10.0";

  minimumOCamlVersion = "4.03";

  src = fetchFromGitHub {
    owner  = "inhabitedtype";
    repo   = pname;
    rev    = version;
    sha256 = "0lh6024yf9ds0nh9i93r9m6p5psi8nvrqxl5x7jwl13zb0r9xfpw";
  };
```

```
  buildInputs = [ alcotest ];
  propagatedBuildInputs = [ bigstringaf result ];
  doCheck = true;

  meta = with lib; {
    homepage = "https://github.com/inhabitedtype/angstrom";
    description = "OCaml parser combinators built for speed and memory effi
    license = licenses.bsd3;
    maintainers = with maintainers; [ sternenseemann ];
  };
}
```

Here is a second example, this time using a source archive generated with `dune-release`. It is a good idea to use this archive when it is available as it will usually contain substituted variables such as a `%%VERSION%%` field. This library does not depend on any other OCaml library and no tests are run after building it.

```
{ lib
, fetchurl
, buildDunePackage
}:

buildDunePackage rec {
  pname = "wtf8";
  version = "1.0.1";

  minimumOCamlVersion = "4.01";

  src = fetchurl {
    url = "https://github.com/flowtype/ocaml-${pname}/releases/download/v${
    sha256 = "1msg3vycd3k8qqj61sc23qks541cxpb97vrnrvrhjnqxsqnh6ygq";
  };

  meta = with lib; {
    homepage = "https://github.com/flowtype/ocaml-wtf8";
    description = "WTF-8 is a superset of UTF-8 that allows unpaired surrog
    license = licenses.mit;
    maintainers = [ maintainers.eqyiel ];
  };
}
```

15.19. Perl

15.19.1. Running perl programs on the shell

When executing a Perl script, it is possible you get an error such as `./myscript.pl: bad interpreter: /usr/bin/perl: no such file or directory`. This happens when the script expects Perl to be installed at `/usr/bin/perl`, which is not the case when using Perl from nixpkgs. You can fix the script by changing the first line to:

```
#!/usr/bin/env perl
```

to take the Perl installation from the `PATH` environment variable, or invoke Perl directly with:

```
$ perl ./myscript.pl
```

When the script is using a Perl library that is not installed globally, you might get an error such as `Can't locate DB_File.pm in @INC (you may need to install the DB_File module)`. In that case, you can use `nix-shell` to start an ad-hoc shell with that library installed, for instance:

```
$ nix-shell -p perl perlPackages.DBFile --run ./myscript.pl
```

If you are always using the script in places where `nix-shell` is available, you can embed the `nix-shell` invocation in the shebang like this:

```
#!/usr/bin/env nix-shell
#! nix-shell -i perl -p perl perlPackages.DBFile
```

15.19.2. Packaging Perl programs

Nixpkgs provides a function `buildPerlPackage`, a generic package builder function for any Perl package that has a standard `Makefile.PL`. It's implemented in pkgs/development/perl-modules/generic.

Perl packages from CPAN are defined in pkgs/top-level/perl-packages.nix rather than `pkgs/all-packages.nix`. Most Perl packages are so straight-forward to build that they are defined here directly, rather than having a separate function for each package called from `perl-packages.nix`. However, more complicated packages should be put in a separate file, typically in `pkgs/development/perl-modules`. Here is an example of the former:

```
ClassC3 = buildPerlPackage rec {
  name = "Class-C3-0.21";
  src = fetchurl {
```

```
    url = "mirror://cpan/authors/id/F/FL/FLORA/${name}.tar.gz";
    sha256 = "1bl8z095y4js66pwxnm7s853pi9czala4sqc743fdlnk27kq94gz";
  };
};
```

Note the use of `mirror://cpan/`, and the `${name}` in the URL definition to ensure that the name attribute is consistent with the source that we're actually downloading. Perl packages are made available in `all-packages.nix` through the variable `perlPackages`. For instance, if you have a package that needs `ClassC3`, you would typically write

```
foo = import ../path/to/foo.nix {
  inherit stdenv fetchurl ...;
  inherit (perlPackages) ClassC3;
};
```

in `all-packages.nix`. You can test building a Perl package as follows:

```
$ nix-build -A perlPackages.ClassC3
```

`buildPerlPackage` adds `perl-` to the start of the name attribute, so the package above is actually called `perl-Class-C3-0.21`. So to install it, you can say:

```
$ nix-env -i perl-Class-C3
```

(Of course you can also install using the attribute name: `nix-env -i -A perlPackages.ClassC3`.)

So what does `buildPerlPackage` do? It does the following:

1. In the configure phase, it calls `perl Makefile.PL` to generate a Makefile. You can set the variable `makeMakerFlags` to pass flags to `Makefile.PL`
2. It adds the contents of the `PERL5LIB` environment variable to `#! .../bin/perl` line of Perl scripts as `-Idir` flags. This ensures that a script can find its dependencies. (This can cause this shebang line to become too long for Darwin to handle; see the note below.)
3. In the fixup phase, it writes the propagated build inputs (`propagatedBuildInputs`) to the file `$out/nix-support/propagated-user-env-packages`. `nix-env` recursively installs all packages listed in this file when you install a package that has it. This ensures that a Perl package can find its dependencies.

`buildPerlPackage` is built on top of `stdenv`, so everything can be customised in the usual way. For instance, the `BerkeleyDB` module has a `preConfigure` hook to generate a configuration file used by `Makefile.PL`:

```
{ buildPerlPackage, fetchurl, db }:

buildPerlPackage rec {
  name = "BerkeleyDB-0.36";

  src = fetchurl {
    url = "mirror://cpan/authors/id/P/PM/PMQS/${name}.tar.gz";
    sha256 = "07xf50riarb60l1h6m2dqmql8q5dij619712fsgw7ach04d8g3z1";
  };

  preConfigure = ''
    echo "LIB = ${db.out}/lib" > config.in
    echo "INCLUDE = ${db.dev}/include" >> config.in
  '';
}
```

Dependencies on other Perl packages can be specified in the `buildInputs` and `propagatedBuildInputs` attributes. If something is exclusively a build-time dependency, use `buildInputs`; if it's (also) a runtime dependency, use `propagatedBuildInputs`. For instance, this builds a Perl module that has runtime dependencies on a bunch of other modules:

```
ClassC3Componentised = buildPerlPackage rec {
  name = "Class-C3-Componentised-1.0004";
  src = fetchurl {
    url = "mirror://cpan/authors/id/A/AS/ASH/${name}.tar.gz";
    sha256 = "0xql73jkcdbq4q9m0b0rnca6nrlvf5hyzy8is0crdk65bynvs8q1";
  };
  propagatedBuildInputs = [
    ClassC3 ClassInspector TestException MROCompat
  ];
};
```

On Darwin, if a script has too many `-Idir` flags in its first line (its "shebang line"), it will not run. This can be worked around by calling the `shortenPerlShebang` function from the `postInstall` phase:

```
{ lib, stdenv, buildPerlPackage, fetchurl, shortenPerlShebang }:

ImageExifTool = buildPerlPackage {
  pname = "Image-ExifTool";
  version = "11.50";

  src = fetchurl {
    url = "https://www.sno.phy.queensu.ca/~phil/exiftool/Image-ExifTool-11.
```

```
    sha256 = "0d8v48y94z8maxkmw1rv7v9m0jg2dc8xbp581njb6yhr7abwqdv3";
  };

  buildInputs = lib.optional stdenv.isDarwin shortenPerlShebang;
  postInstall = lib.optional stdenv.isDarwin ''
    shortenPerlShebang $out/bin/exiftool
  '';
};
```

This will remove the `-I` flags from the shebang line, rewrite them in the `use lib` form, and put them on the next line instead. This function can be given any number of Perl scripts as arguments; it will modify them in-place.

15.19.2.1. Generation from CPAN

Nix expressions for Perl packages can be generated (almost) automatically from CPAN. This is done by the program `nix-generate-from-cpan`, which can be installed as follows:

```
$ nix-env -i nix-generate-from-cpan
```

This program takes a Perl module name, looks it up on CPAN, fetches and unpacks the corresponding package, and prints a Nix expression on standard output. For example:

```
$ nix-generate-from-cpan XML::Simple
  XMLSimple = buildPerlPackage rec {
    name = "XML-Simple-2.22";
    src = fetchurl {
      url = "mirror://cpan/authors/id/G/GR/GRANTM/${name}.tar.gz";
      sha256 = "b9450ef22ea9644ae5d6ada086dc4300fa105be050a2030ebd4efd28c19
    };
    propagatedBuildInputs = [ XMLNamespaceSupport XMLSAX XMLSAXExpat ];
    meta = {
      description = "An API for simple XML files";
      license = with lib.licenses; [ artistic1 gpl1Plus ];
    };
  };
```

The output can be pasted into `pkgs/top-level/perl-packages.nix` or wherever else you need it.

15.19.2.2. Cross-compiling modules

Nixpkgs has experimental support for cross-compiling Perl modules. In many cases, it will just work out of the box, even for modules with native extensions. Sometimes, however, the Makefile.PL for a module

may (indirectly) import a native module. In that case, you will need to make a stub for that module that will satisfy the Makefile.PL and install it into `lib/perl5/site_perl/cross_perl/${perl.version}`. See the `postInstall` for `DBI` for an example.

15.20. PHP

15.20.1. User Guide

15.20.1.1. Overview

Several versions of PHP are available on Nix, each of which having a wide variety of extensions and libraries available.

The different versions of PHP that nixpkgs provides are located under attributes named based on major and minor version number; e.g., `php74` is PHP 7.4.

Only versions of PHP that are supported by upstream for the entirety of a given NixOS release will be included in that release of NixOS. See PHP Supported Versions.

The attribute `php` refers to the version of PHP considered most stable and thoroughly tested in nixpkgs for any given release of NixOS - not necessarily the latest major release from upstream.

All available PHP attributes are wrappers around their respective binary PHP package and provide commonly used extensions this way. The real PHP 7.4 package, i.e. the unwrapped one, is available as `php74.unwrapped`; see the next section for more details.

Interactive tools built on PHP are put in `php.packages`; composer is for example available at `php.packages.composer`.

Most extensions that come with PHP, as well as some popular third-party ones, are available in `php.extensions`; for example, the opcache extension shipped with PHP is available at `php.extensions.opcache` and the third-party ImageMagick extension at `php.extensions.imagick`.

15.20.1.2. Installing PHP with extensions

A PHP package with specific extensions enabled can be built using `php.withExtensions`. This is a function which accepts an anonymous function as its only argument; the function should accept two named parameters: `enabled` - a list of currently enabled extensions and `all` - the set of all extensions, and return a list of wanted extensions. For example, a PHP package with all default extensions and ImageMagick enabled:

```
php.withExtensions ({ enabled, all }:
  enabled ++ [ all.imagick ])
```

To exclude some, but not all, of the default extensions, you can filter the `enabled` list like this:

```
php.withExtensions ({ enabled, all }:
  (lib.filter (e: e != php.extensions.opcache) enabled)
  ++ [ all.imagick ])
```

To build your list of extensions from the ground up, you can simply ignore `enabled`:

```
php.withExtensions ({ all, ... }: with all; [ imagick opcache ])
```

`php.withExtensions` provides extensions by wrapping a minimal php base package, providing a `php.ini` file listing all extensions to be loaded. You can access this package through the `php.unwrapped` attribute; useful if you, for example, need access to the `dev` output. The generated `php.ini` file can be accessed through the `php.phpIni` attribute.

If you want a PHP build with extra configuration in the `php.ini` file, you can use `php.buildEnv`. This function takes two named and optional parameters: `extensions` and `extraConfig`. `extensions` takes an extension specification equivalent to that of `php.withExtensions`, `extraConfig` a string of additional `php.ini` configuration parameters. For example, a PHP package with the opcache and ImageMagick extensions enabled, and `memory_limit` set to `256M`:

```
php.buildEnv {
  extensions = { all, ... }: with all; [ imagick opcache ];
  extraConfig = "memory_limit=256M";
}
```

15.20.1.2.1. Example setup for `phpfpm`

You can use the previous examples in a `phpfpm` pool called `foo` as follows:

```
let
  myPhp = php.withExtensions ({ all, ... }: with all; [ imagick opcache ]);
in {
  services.phpfpm.pools."foo".phpPackage = myPhp;
};
```

```
let
  myPhp = php.buildEnv {
    extensions = { all, ... }: with all; [ imagick opcache ];
    extraConfig = "memory_limit=256M";
  };
```

```
in {
  services.phpfpm.pools."foo".phpPackage = myPhp;
};
```

15.20.1.2.2. Example usage with `nix-shell`

This brings up a temporary environment that contains a PHP interpreter with the extensions `imagick` and `opcache` enabled:

```
nix-shell -p 'php.withExtensions ({ all, ... }: with all; [ imagick opcache
```

15.20.1.3. Installing PHP packages with extensions

All interactive tools use the PHP package you get them from, so all packages at `php.packages.*` use the `php` package with its default extensions. Sometimes this default set of extensions isn't enough and you may want to extend it. A common case of this is the `composer` package: a project may depend on certain extensions and `composer` won't work with that project unless those extensions are loaded.

Example of building `composer` with additional extensions:

```
(php.withExtensions ({ all, enabled }:
  enabled ++ (with all; [ imagick redis ]))
).packages.composer
```

15.20.1.4. Overriding PHP packages

`php-packages.nix` form a scope, allowing us to override the packages defined within. For example, to apply a patch to a `mysqlnd` extension, you can simply pass an overlay-style function to `php`'s `packageOverrides` argument:

```
php.override {
  packageOverrides = final: prev: {
    extensions = prev.extensions // {
      mysqlnd = prev.extensions.mysqlnd.overrideAttrs (attrs: {
        patches = attrs.patches or [] ++ [
          …
        ];
      });
    };
  };
}
```

15.21. Python

15.21.1. User Guide

15.21.1.1. Using Python

15.21.1.1.1. Overview

Several versions of the Python interpreter are available on Nix, as well as a high amount of packages. The attribute `python3` refers to the default interpreter, which is currently CPython 3.8. The attribute `python` refers to CPython 2.7 for backwards-compatibility. It is also possible to refer to specific versions, e.g. `python38` refers to CPython 3.8, and `pypy` refers to the default PyPy interpreter.

Python is used a lot, and in different ways. This affects also how it is packaged. In the case of Python on Nix, an important distinction is made between whether the package is considered primarily an application, or whether it should be used as a library, i.e., of primary interest are the modules in `site-packages` that should be importable.

In the Nixpkgs tree Python applications can be found throughout, depending on what they do, and are called from the main package set. Python libraries, however, are in separate sets, with one set per interpreter version.

The interpreters have several common attributes. One of these attributes is `pkgs`, which is a package set of Python libraries for this specific interpreter. E.g., the `toolz` package corresponding to the default interpreter is `python.pkgs.toolz`, and the CPython 3.8 version is `python38.pkgs.toolz`. The main package set contains aliases to these package sets, e.g. `pythonPackages` refers to `python.pkgs` and `python38Packages` to `python38.pkgs`.

15.21.1.1.2. Installing Python and packages

The Nix and NixOS manuals explain how packages are generally installed. In the case of Python and Nix, it is important to make a distinction between whether the package is considered an application or a library.

Applications on Nix are typically installed into your user profile imperatively using `nix-env -i`, and on NixOS declaratively by adding the package name to `environment.systemPackages` in `/etc/nixos/configuration.nix`. Dependencies such as libraries are automatically installed and should not be installed explicitly.

The same goes for Python applications. Python applications can be installed in your profile, and will be wrapped to find their exact library dependencies, without impacting other applications or polluting your user environment.

But Python libraries you would like to use for development cannot be installed, at least not individually, because they won't be able to find each other resulting in import errors. Instead, it is possible to create an environment with `python.buildEnv` or `python.withPackages` where the interpreter and other executables are wrapped to be able to find each other and all of the modules.

In the following examples we will start by creating a simple, ad-hoc environment with a nix-shell that has `numpy` and `toolz` in Python 3.8; then we will create a re-usable environment in a single-file Python script; then we will create a full Python environment for development with this same environment.

Philosphically, this should be familiar to users who are used to a `venv` style of development: individual projects create their own Python environments without impacting the global environment or each other.

15.21.1.1.3. Ad-hoc temporary Python environment with `nix-shell`

The simplest way to start playing with the way nix wraps and sets up Python environments is with `nix-shell` at the cmdline. These environments create a temporary shell session with a Python and a *precise* list of packages (plus their runtime dependencies), with no other Python packages in the Python interpreter's scope.

To create a Python 3.8 session with `numpy` and `toolz` available, run:

```
$ nix-shell -p 'python38.withPackages(ps: with ps; [ numpy toolz ])'
```

By default `nix-shell` will start a `bash` session with this interpreter in our `PATH`, so if we then run:

```
Python console [nix-shell:~/src/nixpkgs]$ python3 Python 3.8.1 (default, Dec 18 2019, 19:06:26) [GCC 9.2.0] on linux Type "help", "copyright", "credits" or "license" for more information. >>> import numpy; import toolz
```

Note that no other modules are in scope, even if they were imperatively installed into our user environment as a dependency of a Python application:

```
Python console >>> import requests Traceback (most recent call last): File "<stdin>", line 1, in <module> ModuleNotFoundError: No module named 'requests'
```

We can add as many additional modules onto the `nix-shell` as we need, and we will still get 1 wrapped Python interpreter. We can start the interpreter directly like so:

```
$ nix-shell -p 'python38.withPackages(ps: with ps; [ numpy toolz requests ]
these derivations will be built:
  /nix/store/xbdsrqrsfa1yva5s7pzsra8k08gxlbz1-python3-3.8.1-env.drv
building '/nix/store/xbdsrqrsfa1yva5s7pzsra8k08gxlbz1-python3-3.8.1-env.drv
created 277 symlinks in user environment
Python 3.8.1 (default, Dec 18 2019, 19:06:26)
[GCC 9.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
```

Notice that this time it built a new Python environment, which now includes `requests`. Building an environment just creates wrapper scripts that expose the selected dependencies to the interpreter while re-using the actual modules. This means if any other env has installed `requests` or `numpy` in a different context, we don't need to recompile them – we just recompile the wrapper script that sets up an interpreter pointing to them. This matters much more for "big" modules like `pytorch` or `tensorflow`.

Module names usually match their names on pypi.org, but you can use the Nixpkgs search website to find them as well (along with non-python packages).

At this point we can create throwaway experimental Python environments with arbitrary dependencies. This is a good way to get a feel for how the Python interpreter and dependencies work in Nix and NixOS, but to do some actual development, we'll want to make it a bit more persistent.

15.21.1.1.3.1. Running Python scripts and using `nix-shell` as shebang

Sometimes, we have a script whose header looks like this:

```
#!/usr/bin/env python3
import numpy as np
a = np.array([1,2])
b = np.array([3,4])
print(f"The dot product of {a} and {b} is: {np.dot(a, b)}")
```

Executing this script requires a `python3` that has `numpy`. Using what we learned in the previous section, we could startup a shell and just run it like so:

```
$ nix-shell -p 'python38.withPackages(ps: with ps; [ numpy ])' --run 'pytho
The dot product of [1 2] and [3 4] is: 11
```

But if we maintain the script ourselves, and if there are more dependencies, it may be nice to encode those dependencies in source to make the script re-usable without that bit of knowledge. That can be done by using `nix-shell` as a shebang, like so:

```
#!/usr/bin/env nix-shell
#!nix-shell -i python3 -p "python3.withPackages(ps: [ ps.numpy ])"
import numpy as np
a = np.array([1,2])
b = np.array([3,4])
print(f"The dot product of {a} and {b} is: {np.dot(a, b)}")
```

Then we simply execute it, without requiring any environment setup at all!

```
$ ./foo.py
```

```
The dot product of [1 2] and [3 4] is: 11
```

If the dependencies are not available on the host where `foo.py` is executed, it will build or download them from a Nix binary cache prior to starting up, prior that it is executed on a machine with a multi-user nix installation.

This provides a way to ship a self bootstrapping Python script, akin to a statically linked binary, where it can be run on any machine (provided nix is installed) without having to assume that `numpy` is installed globally on the system.

By default it is pulling the import checkout of Nixpkgs itself from our nix channel, which is nice as it cache aligns with our other package builds, but we can make it fully reproducible by pinning the `nixpkgs` import:

```
#!/usr/bin/env nix-shell
#!nix-shell -i python3 -p "python3.withPackages(ps: [ ps.numpy ])"
#!nix-shell -I nixpkgs=https://github.com/NixOS/nixpkgs/archive/d373d80b120
import numpy as np
a = np.array([1,2])
b = np.array([3,4])
print(f"The dot product of {a} and {b} is: {np.dot(a, b)}")
```

This will execute with the exact same versions of Python 3.8, numpy, and system dependencies a year from now as it does today, because it will always use exactly git commit `d373d80b1207d52621961b16aa4a3438e4f98167` of Nixpkgs for all of the package versions.

This is also a great way to ensure the script executes identically on different servers.

15.21.1.1.3.2. Load environment from `.nix` expression

We've now seen how to create an ad-hoc temporary shell session, and how to create a single script with Python dependencies, but in the course of normal development we're usually working in an entire package repository.

As explained in the Nix manual, `nix-shell` can also load an expression from a `.nix` file. Say we want to have Python 3.8, `numpy` and `toolz`, like before, in an environment. We can add a `shell.nix` file describing our dependencies:

```
with import <nixpkgs> {};
(python38.withPackages (ps: [ps.numpy ps.toolz])).env
```

And then at the command line, just typing `nix-shell` produces the same environment as before. In a normal project, we'll likely have many more dependencies; this can provide a way for developers to share the environments with each other and with CI builders.

What's happening here?

1. We begin with importing the Nix Packages collections. `import <nixpkgs>` imports the `<nixpkgs>` function, `{}` calls it and the `with` statement brings all attributes of `nixpkgs` in the local scope. These attributes form the main package set.

2. Then we create a Python 3.8 environment with the `withPackages` function, as before.

3. The `withPackages` function expects us to provide a function as an argument that takes the set of all Python packages and returns a list of packages to include in the environment. Here, we select the packages `numpy` and `toolz` from the package set.

To combine this with `mkShell` you can:

```
with import <nixpkgs> {};
let
  pythonEnv = python38.withPackages (ps: [
    ps.numpy
    ps.toolz
  ]);
in mkShell {
  packages = [
    pythonEnv

    black
    mypy

    libffi
    openssl
  ];
}
```

This will create a unified environment that has not just our Python interpreter and its Python dependencies, but also tools like `black` or `mypy` and libraries like `libffi` the `openssl` in scope. This is generic and can span any number of tools or languages across the Nixpkgs ecosystem.

15.21.1.1.3.3. Installing environments globally on the system

Up to now, we've been creating environments scoped to an ad-hoc shell session, or a single script, or a single project. This is generally advisable, as it avoids pollution across contexts.

However, sometimes we know we will often want a Python with some basic packages, and want this available without having to enter into a shell or build context. This can be useful to have things like vim/emacs editors and plugins or shell tools "just work" without having to set them up, or when running other software that expects packages to be installed globally.

To create your own custom environment, create a file in `~/.config/nixpkgs/overlays/` that looks

like this:

```
# ~/.config/nixpkgs/overlays/myEnv.nix
self: super: {
  myEnv = super.buildEnv {
    name = "myEnv";
    paths = [
      # A Python 3 interpreter with some packages
      (self.python3.withPackages (
        ps: with ps; [
          pyflakes
          pytest
          python-language-server
        ]
      ))

      # Some other packages we'd like as part of this env
      self.mypy
      self.black
      self.ripgrep
      self.tmux
    ];
  };
}
```

You can then build and install this to your profile with:

```
nix-env -iA myEnv
```

One limitation of this is that you can only have 1 Python env installed globally, since they conflict on the `python` to load out of your `PATH`.

If you get a conflict or prefer to keep the setup clean, you can have `nix-env` atomically *uninstall* all other imperatively installed packages and replace your profile with just `myEnv` by using the `--replace` flag.

15.21.1.1.3.4. Environment defined in `/etc/nixos/configuration.nix`

For the sake of completeness, here's how to install the environment system-wide on NixOS.

```
{ # ...

  environment.systemPackages = with pkgs; [
    (python38.withPackages(ps: with ps; [ numpy toolz ]))
  ];
```

```
}
```

15.21.1.2. Developing with Python

Above, we were mostly just focused on use cases and what to do to get started creating working Python environments in nix.

Now that you know the basics to be up and running, it is time to take a step back and take a deeper look at how Python packages are packaged on Nix. Then, we will look at how you can use development mode with your code.

15.21.1.2.1. Python library packages in Nixpkgs

With Nix all packages are built by functions. The main function in Nix for building Python libraries is `buildPythonPackage`. Let's see how we can build the `toolz` package.

```
{ lib, buildPythonPackage, fetchPypi }:

buildPythonPackage rec {
  pname = "toolz";
  version = "0.10.0";

  src = fetchPypi {
    inherit pname version;
    sha256 = "08fdd5ef7c96480ad11c12d472de21acd32359996f69a5259299b540feba4
  };

  doCheck = false;

  meta = with lib; {
    homepage = "https://github.com/pytoolz/toolz";
    description = "List processing tools and functional utilities";
    license = licenses.bsd3;
    maintainers = with maintainers; [ fridh ];
  };
}
```

What happens here? The function `buildPythonPackage` is called and as argument it accepts a set. In this case the set is a recursive set, `rec`. One of the arguments is the name of the package, which consists of a basename (generally following the name on PyPi) and a version. Another argument, `src` specifies the source, which in this case is fetched from PyPI using the helper function `fetchPypi`. The argument `doCheck` is used to set whether tests should be run when building the package. Furthermore, we specify some (optional) meta information. The output of the function is a derivation.

An expression for `toolz` can be found in the Nixpkgs repository. As explained in the introduction of this Python section, a derivation of `toolz` is available for each interpreter version, e.g. `python38.pkgs.toolz` refers to the `toolz` derivation corresponding to the CPython 3.8 interpreter.

The above example works when you're directly working on `pkgs/top-level/python-packages.nix` in the Nixpkgs repository. Often though, you will want to test a Nix expression outside of the Nixpkgs tree.

The following expression creates a derivation for the `toolz` package, and adds it along with a `numpy` package to a Python environment.

```
with import <nixpkgs> {};

( let
    my_toolz = python38.pkgs.buildPythonPackage rec {
      pname = "toolz";
      version = "0.10.0";

      src = python38.pkgs.fetchPypi {
        inherit pname version;
        sha256 = "08fdd5ef7c96480ad11c12d472de21acd32359996f69a5259299b540f
      };

      doCheck = false;

      meta = {
        homepage = "https://github.com/pytoolz/toolz/";
        description = "List processing tools and functional utilities";
      };
    };

  in python38.withPackages (ps: [ps.numpy my_toolz])
).env
```

Executing `nix-shell` will result in an environment in which you can use Python 3.8 and the `toolz` package. As you can see we had to explicitly mention for which Python version we want to build a package.

So, what did we do here? Well, we took the Nix expression that we used earlier to build a Python environment, and said that we wanted to include our own version of `toolz`, named `my_toolz`. To introduce our own package in the scope of `withPackages` we used a `let` expression. You can see that we used `ps.numpy` to select numpy from the nixpkgs package set (`ps`). We did not take `toolz` from the Nixpkgs package set this time, but instead took our own version that we introduced with the `let` expression.

15.21.1.2.2. Handling dependencies

Our example, `toolz`, does not have any dependencies on other Python packages or system libraries. According to the manual, `buildPythonPackage` uses the arguments `buildInputs` and `propagatedBuildInputs` to specify dependencies. If something is exclusively a build-time dependency, then the dependency should be included in `buildInputs`, but if it is (also) a runtime dependency, then it should be added to `propagatedBuildInputs`. Test dependencies are considered build-time dependencies and passed to `checkInputs`.

The following example shows which arguments are given to `buildPythonPackage` in order to build `datashape`.

```
{ lib, buildPythonPackage, fetchPypi, numpy, multipledispatch, dateutil, py

buildPythonPackage rec {
  pname = "datashape";
  version = "0.4.7";

  src = fetchPypi {
    inherit pname version;
    sha256 = "14b2ef766d4c9652ab813182e866f493475e65e558bed0822e38bf07bba1a
  };

  checkInputs = [ pytest ];
  propagatedBuildInputs = [ numpy multipledispatch dateutil ];

  meta = with lib; {
    homepage = "https://github.com/ContinuumIO/datashape";
    description = "A data description language";
    license = licenses.bsd2;
    maintainers = with maintainers; [ fridh ];
  };
}
```

We can see several runtime dependencies, `numpy`, `multipledispatch`, and `dateutil`. Furthermore, we have one `checkInputs`, i.e. `pytest`. `pytest` is a test runner and is only used during the `checkPhase` and is therefore not added to `propagatedBuildInputs`.

In the previous case we had only dependencies on other Python packages to consider. Occasionally you have also system libraries to consider. E.g., `lxml` provides Python bindings to `libxml2` and `libxslt`. These libraries are only required when building the bindings and are therefore added as `buildInputs`.

```
{ lib, pkgs, buildPythonPackage, fetchPypi }:
```

```
buildPythonPackage rec {
  pname = "lxml";
  version = "3.4.4";

  src = fetchPypi {
    inherit pname version;
    sha256 = "16a0fa97hym9ysdk3rmqz32xdjqmy4w34ld3rm3jf5viqjx65lxk";
  };

  buildInputs = [ pkgs.libxml2 pkgs.libxslt ];

  meta = with lib; {
    description = "Pythonic binding for the libxml2 and libxslt libraries";
    homepage = "https://lxml.de";
    license = licenses.bsd3;
    maintainers = with maintainers; [ sjourdois ];
  };
}
```

In this example `lxml` and Nix are able to work out exactly where the relevant files of the dependencies are. This is not always the case.

The example below shows bindings to The Fastest Fourier Transform in the West, commonly known as FFTW. On Nix we have separate packages of FFTW for the different types of floats (`"single"`, `"double"`, `"long-double"`). The bindings need all three types, and therefore we add all three as `buildInputs`. The bindings don't expect to find each of them in a different folder, and therefore we have to set `LDFLAGS` and `CFLAGS`.

```
{ lib, pkgs, buildPythonPackage, fetchPypi, numpy, scipy }:

buildPythonPackage rec {
  pname = "pyFFTW";
  version = "0.9.2";

  src = fetchPypi {
    inherit pname version;
    sha256 = "f6bbb6afa93085409ab24885a1a3cdb8909f095a142f4d49e346f2bd1b789
  };

  buildInputs = [ pkgs.fftw pkgs.fftwFloat pkgs.fftwLongDouble];

  propagatedBuildInputs = [ numpy scipy ];
```

```
  # Tests cannot import pyfftw. pyfftw works fine though.
  doCheck = false;

  preConfigure = ''
    export LDFLAGS="-L${pkgs.fftw.dev}/lib -L${pkgs.fftwFloat.out}/lib -L${
    export CFLAGS="-I${pkgs.fftw.dev}/include -I${pkgs.fftwFloat.dev}/inclu
  '';

  meta = with lib; {
    description = "A pythonic wrapper around FFTW, the FFT library, present
    homepage = "http://hgomersall.github.com/pyFFTW";
    license = with licenses; [ bsd2 bsd3 ];
    maintainers = with maintainers; [ fridh ];
  };
}
```

Note also the line `doCheck = false;`, we explicitly disabled running the test-suite.

15.21.1.2.3. Testing Python Packages

It is highly encouraged to have testing as part of the package build. This helps to avoid situations where the package was able to build and install, but is not usable at runtime. Currently, all packages will use the `test` command provided by the setup.py (i.e. `python setup.py test`). However, this is currently deprecated https://github.com/pypa/setuptools/pull/1878 and your package should provide its own checkPhase.

NOTE: The `checkPhase` for python maps to the `installCheckPhase` on a normal derivation. This is due to many python packages not behaving well to the pre-installed version of the package. Version info, and natively compiled extensions generally only exist in the install directory, and thus can cause issues when a test suite asserts on that behavior.

NOTE: Tests should only be disabled if they don't agree with nix (e.g. external dependencies, network access, flakey tests), however, as many tests should be enabled as possible. Failing tests can still be a good indication that the package is not in a valid state.

15.21.1.2.4. Using pytest

Pytest is the most common test runner for python repositories. A trivial test run would be:

```
checkInputs = [ pytest ];
checkPhase = "pytest";
```

However, many repositories' test suites do not translate well to nix's build sandbox, and will generally need many tests to be disabled.

To filter tests using pytest, one can do the following:

```
checkInputs = [ pytest ];
# avoid tests which need additional data or touch network
checkPhase = ''
  pytest tests/ --ignore=tests/integration -k 'not download and not updat
'';
```

`--ignore` will tell pytest to ignore that file or directory from being collected as part of a test run. This is useful is a file uses a package which is not available in nixpkgs, thus skipping that test file is much easier than having to create a new package.

`-k` is used to define a predicate for test names. In this example, we are filtering out tests which contain `download` or `update` in their test case name. Only one `-k` argument is allows, and thus a long predicate should be concatenated with "" and wrapped to the next line.

NOTE: In pytest==6.0.1, the use of "" to continue a line (e.g. `-k 'not download \'`) has been removed, in this case, it's recommended to use `pytestCheckHook`.

15.21.1.2.5. Using pytestCheckHook

`pytestCheckHook` is a convenient hook which will substitute the setuptools `test` command for a checkPhase which runs `pytest`. This is also beneficial when a package may need many items disabled to run the test suite.

Using the example above, the analagous pytestCheckHook usage would be:

```
checkInputs = [ pytestCheckHook ];

# requires additional data
pytestFlagsArray = [ "tests/" "--ignore=tests/integration" ];

disabledTests = [
  # touches network
  "download"
  "update"
];

disabledTestPaths = [
  "tests/test_failing.py"
];
```

This is expecially useful when tests need to be conditionallydisabled, for example:

```
disabledTests = [
  # touches network
  "download"
  "update"
] ++ lib.optionals (pythonAtLeast "3.8") [
  # broken due to python3.8 async changes
  "async"
] ++ lib.optionals stdenv.isDarwin [
  # can fail when building with other packages
  "socket"
];
```

Trying to concatenate the related strings to disable tests in a regular checkPhase would be much harder to read. This also enables us to comment on why specific tests are disabled.

15.21.1.2.6. Using pythonImportsCheck

Although unit tests are highly prefered to validate correctness of a package, not all packages have test suites that can be ran easily, and some have none at all. To help ensure the package still works, `pythonImportsCheck` can attempt to import the listed modules.

```
pythonImportsCheck = [ "requests" "urllib" ];
```

roughly translates to:

```
postCheck = ''
  PYTHONPATH=$out/${python.sitePackages}:$PYTHONPATH
  python -c "import requests; import urllib"
'';
```

However, this is done in it's own phase, and not dependent on whether `doCheck = true;`

This can also be useful in verifying that the package doesn't assume commonly present packages (e.g. `setuptools`)

15.21.1.3. Develop local package

As a Python developer you're likely aware of development mode (`python setup.py develop`); instead of installing the package this command creates a special link to the project code. That way, you can run updated code without having to reinstall after each and every change you make. Development mode is also available. Let's see how you can use it.

In the previous Nix expression the source was fetched from an url. We can also refer to a local source instead using `src = ./path/to/source/tree;`

If we create a `shell.nix` file which calls `buildPythonPackage`, and if `src` is a local source, and if the local source has a `setup.py`, then development mode is activated.

In the following example we create a simple environment that has a Python 3.8 version of our package in it, as well as its dependencies and other packages we like to have in the environment, all specified with `propagatedBuildInputs`. Indeed, we can just add any package we like to have in our environment to `propagatedBuildInputs`.

```
with import <nixpkgs> {};
with python38Packages;

buildPythonPackage rec {
  name = "mypackage";
  src = ./path/to/package/source;
  propagatedBuildInputs = [ pytest numpy pkgs.libsndfile ];
}
```

It is important to note that due to how development mode is implemented on Nix it is not possible to have multiple packages simultaneously in development mode.

15.21.1.4. Organising your packages

So far we discussed how you can use Python on Nix, and how you can develop with it. We've looked at how you write expressions to package Python packages, and we looked at how you can create environments in which specified packages are available.

At some point you'll likely have multiple packages which you would like to be able to use in different projects. In order to minimise unnecessary duplication we now look at how you can maintain a repository with your own packages. The important functions here are `import` and `callPackage`.

15.21.1.5. Including a derivation using `callPackage`

Earlier we created a Python environment using `withPackages`, and included the `toolz` package via a `let` expression. Let's split the package definition from the environment definition.

We first create a function that builds `toolz` in `~/path/to/toolz/release.nix`

```
{ lib, buildPythonPackage }:

buildPythonPackage rec {
  pname = "toolz";
  version = "0.10.0";

  src = fetchPypi {
    inherit pname version;
```

```
    sha256 = "08fdd5ef7c96480ad11c12d472de21acd32359996f69a5259299b540feba4
  };

  meta = with lib; {
    homepage = "https://github.com/pytoolz/toolz/";
    description = "List processing tools and functional utilities";
    license = licenses.bsd3;
    maintainers = with maintainers; [ fridh ];
  };
}
```

It takes an argument `buildPythonPackage`. We now call this function using `callPackage` in the definition of our environment

```
with import <nixpkgs> {};

( let
    toolz = callPackage /path/to/toolz/release.nix {
      buildPythonPackage = python38Packages.buildPythonPackage;
    };
  in python38.withPackages (ps: [ ps.numpy toolz ])
).env
```

Important to remember is that the Python version for which the package is made depends on the `python` derivation that is passed to `buildPythonPackage`. Nix tries to automatically pass arguments when possible, which is why generally you don't explicitly define which `python` derivation should be used. In the above example we use `buildPythonPackage` that is part of the set `python38Packages`, and in this case the `python38` interpreter is automatically used.

15.21.2. Reference

15.21.2.1. Interpreters

Versions 2.7, 3.6, 3.7, 3.8 and 3.9 of the CPython interpreter are available as respectively `python27`, `python36`, `python37`, `python38` and `python39`. The aliases `python2` and `python3` correspond to respectively `python27` and `python39`. The attribute `python` maps to `python2`. The PyPy interpreters compatible with Python 2.7 and 3 are available as `pypy27` and `pypy3`, with aliases `pypy2` mapping to `pypy27` and `pypy` mapping to `pypy2`. The Nix expressions for the interpreters can be found in `pkgs/development/interpreters/python`.

All packages depending on any Python interpreter get appended `out/{python.sitePackages}` to `$PYTHONPATH` if such directory exists.

15.21.2.1.1. Missing `tkinter` module standard library

To reduce closure size the `Tkinter`/`tkinter` is available as a separate package, `pythonPackages.tkinter`.

15.21.2.1.2. Attributes on interpreters packages

Each interpreter has the following attributes:

- `libPrefix`. Name of the folder in `${python}/lib/` for corresponding interpreter.
- `interpreter`. Alias for `${python}/bin/${executable}`.
- `buildEnv`. Function to build python interpreter environments with extra packages bundled together. See section *python.buildEnv function* for usage and documentation.
- `withPackages`. Simpler interface to `buildEnv`. See section *python.withPackages function* for usage and documentation.
- `sitePackages`. Alias for `lib/${libPrefix}/site-packages`.
- `executable`. Name of the interpreter executable, e.g. `python3.8`.
- `pkgs`. Set of Python packages for that specific interpreter. The package set can be modified by overriding the interpreter and passing `packageOverrides`.

15.21.2.2. Optimizations

The Python interpreters are by default not build with optimizations enabled, because the builds are in that case not reproducible. To enable optimizations, override the interpreter of interest, e.g using

```
let
  pkgs = import ./. {};
  mypython = pkgs.python3.override {
    enableOptimizations = true;
    reproducibleBuild = false;
    self = mypython;
  };
in mypython
```

15.21.2.3. Building packages and applications

Python libraries and applications that use `setuptools` or `distutils` are typically built with respectively the `buildPythonPackage` and `buildPythonApplication` functions. These two functions also support installing a `wheel`.

All Python packages reside in `pkgs/top-level/python-packages.nix` and all applications elsewhere. In case a package is used as both a library and an application, then the package should be in

`pkgs/top-level/python-packages.nix` since only those packages are made available for all interpreter versions. The preferred location for library expressions is in `pkgs/development/python-modules`. It is important that these packages are called from `pkgs/top-level/python-packages.nix` and not elsewhere, to guarantee the right version of the package is built.

Based on the packages defined in `pkgs/top-level/python-packages.nix` an attribute set is created for each available Python interpreter. The available sets are

- `pkgs.python27Packages`
- `pkgs.python36Packages`
- `pkgs.python37Packages`
- `pkgs.python38Packages`
- `pkgs.python39Packages`
- `pkgs.pypyPackages`

and the aliases

- `pkgs.python2Packages` pointing to `pkgs.python27Packages`
- `pkgs.python3Packages` pointing to `pkgs.python38Packages`
- `pkgs.pythonPackages` pointing to `pkgs.python2Packages`

15.21.2.3.1. `buildPythonPackage` function

The `buildPythonPackage` function is implemented in `pkgs/development/interpreters/python/mk-python-derivation` using setup hooks.

The following is an example:

```
{ lib, buildPythonPackage, fetchPypi, hypothesis, setuptools_scm, attrs, py

buildPythonPackage rec {
  pname = "pytest";
  version = "3.3.1";

  src = fetchPypi {
    inherit pname version;
    sha256 = "cf8436dc59d8695346fcd3ab296de46425ecab00d64096cebe79fb51ecb2e
  };

  postPatch = ''
    # don't test bash builtins
```

```
    rm testing/test_argcomplete.py
  '';

  checkInputs = [ hypothesis ];
  nativeBuildInputs = [ setuptools_scm ];
  propagatedBuildInputs = [ attrs py setuptools six pluggy ];

  meta = with lib; {
    maintainers = with maintainers; [ domenkozar lovek323 madjar lsix ];
    description = "Framework for writing tests";
  };
}
```

The `buildPythonPackage` mainly does four things:

- In the `buildPhase`, it calls `${python.interpreter} setup.py bdist_wheel` to build a wheel binary zipfile.
- In the `installPhase`, it installs the wheel file using `pip install *.whl`.
- In the `postFixup` phase, the `wrapPythonPrograms` bash function is called to wrap all programs in the `$out/bin/*` directory to include `$PATH` environment variable and add dependent libraries to script's `sys.path`.
- In the `installCheck` phase, `${python.interpreter} setup.py test` is ran.

By default tests are run because `doCheck = true`. Test dependencies, like e.g. the test runner, should be added to `checkInputs`.

By default `meta.platforms` is set to the same value as the interpreter unless overridden otherwise.

15.21.2.3.1.1. `buildPythonPackage` parameters

All parameters from `stdenv.mkDerivation` function are still supported. The following are specific to `buildPythonPackage`:

- `catchConflicts ? true`: If `true`, abort package build if a package name appears more than once in dependency tree. Default is `true`.
- `disabled` ? false: If `true`, package is not built for the particular Python interpreter version.
- `dontWrapPythonPrograms ? false`: Skip wrapping of Python programs.
- `permitUserSite ? false`: Skip setting the `PYTHONNOUSERSITE` environment variable in wrapped programs.
- `format ? "setuptools"`: Format of the source. Valid options are `"setuptools"`, `"pyproject"`, `"flit"`, `"wheel"`, and `"other"`. `"setuptools"` is for when the source has a `setup.py` and

`setuptools` is used to build a wheel, `flit`, in case `flit` should be used to build a wheel, and `wheel` in case a wheel is provided. Use `other` when a custom `buildPhase` and/or `installPhase` is needed.

- `makeWrapperArgs ? []`: A list of strings. Arguments to be passed to `makeWrapper`, which wraps generated binaries. By default, the arguments to `makeWrapper` set `PATH` and `PYTHONPATH` environment variables before calling the binary. Additional arguments here can allow a developer to set environment variables which will be available when the binary is run. For example, `makeWrapperArgs = ["--set FOO BAR" "--set BAZ QUX"]`.
- `namePrefix`: Prepends text to `${name}` parameter. In case of libraries, this defaults to `"python3.8-"` for Python 3.8, etc., and in case of applications to `""`.
- `pipInstallFlags ? []`: A list of strings. Arguments to be passed to `pip install`. To pass options to `python setup.py install`, use `--install-option`. E.g., `pipInstallFlags=["--install-option='--cpp_implementation'"]`.
- `pythonPath ? []`: List of packages to be added into `$PYTHONPATH`. Packages in `pythonPath` are not propagated (contrary to `propagatedBuildInputs`).
- `preShellHook`: Hook to execute commands before `shellHook`.
- `postShellHook`: Hook to execute commands after `shellHook`.
- `removeBinByteCode ? true`: Remove bytecode from `/bin`. Bytecode is only created when the filenames end with `.py`.
- `setupPyGlobalFlags ? []`: List of flags passed to `setup.py` command.
- `setupPyBuildFlags ? []`: List of flags passed to `setup.py build_ext` command.

The `stdenv.mkDerivation` function accepts various parameters for describing build inputs (see "Specifying dependencies"). The following are of special interest for Python packages, either because these are primarily used, or because their behaviour is different:

- `nativeBuildInputs ? []`: Build-time only dependencies. Typically executables as well as the items listed in `setup_requires`.
- `buildInputs ? []`: Build and/or run-time dependencies that need to be compiled for the host machine. Typically non-Python libraries which are being linked.
- `checkInputs ? []`: Dependencies needed for running the `checkPhase`. These are added to `nativeBuildInputs` when `doCheck = true`. Items listed in `tests_require` go here.
- `propagatedBuildInputs ? []`: Aside from propagating dependencies, `buildPythonPackage` also injects code into and wraps executables with the paths included in this list. Items listed in `install_requires` go here.

15.21.2.3.1.2. Overriding Python packages

The `buildPythonPackage` function has a `overridePythonAttrs` method that can be used to override the package. In the following example we create an environment where we have the `blaze` package using an older version of `pandas`. We override first the Python interpreter and pass `packageOverrides` which contains the overrides for packages in the package set.

```
with import <nixpkgs> {};

(let
  python = let
    packageOverrides = self: super: {
      pandas = super.pandas.overridePythonAttrs(old: rec {
        version = "0.19.1";
        src =  super.fetchPypi {
          pname = "pandas";
          inherit version;
          sha256 = "08blshqj9zj1wyjhhw3kl2vas75vhhicvv72flvf1z3jvapgw295";
        };
      });
    };
  in pkgs.python3.override {inherit packageOverrides; self = python;};

in python.withPackages(ps: [ps.blaze])).env
```

15.21.2.3.2. `buildPythonApplication` function

The `buildPythonApplication` function is practically the same as `buildPythonPackage`. The main purpose of this function is to build a Python package where one is interested only in the executables, and not importable modules. For that reason, when adding this package to a `python.buildEnv`, the modules won't be made available.

Another difference is that `buildPythonPackage` by default prefixes the names of the packages with the version of the interpreter. Because this is irrelevant for applications, the prefix is omitted.

When packaging a Python application with `buildPythonApplication`, it should be called with `callPackage` and passed `python` or `pythonPackages` (possibly specifying an interpreter version), like this:

```
{ lib, python3Packages }:

python3Packages.buildPythonApplication rec {
  pname = "luigi";
  version = "2.7.9";
```

```
  src = python3Packages.fetchPypi {
    inherit pname version;
    sha256 = "035w8gqql36zlan0xjrzz9j4lh9hs0qrsgnbyw07qs7lnkvbdv9x";
  };

  propagatedBuildInputs = with python3Packages; [ tornado_4 python-daemon ]

  meta = with lib; {
    ...
  };
}
```

This is then added to `all-packages.nix` just as any other application would be.

```
luigi = callPackage ../applications/networking/cluster/luigi { };
```

Since the package is an application, a consumer doesn't need to care about Python versions or modules, which is why they don't go in `pythonPackages`.

15.21.2.3.3. `toPythonApplication` function

A distinction is made between applications and libraries, however, sometimes a package is used as both. In this case the package is added as a library to `python-packages.nix` and as an application to `all-packages.nix`. To reduce duplication the `toPythonApplication` can be used to convert a library to an application.

The Nix expression shall use `buildPythonPackage` and be called from `python-packages.nix`. A reference shall be created from `all-packages.nix` to the attribute in `python-packages.nix`, and the `toPythonApplication` shall be applied to the reference:

```
youtube-dl = with pythonPackages; toPythonApplication youtube-dl;
```

15.21.2.3.4. `toPythonModule` function

In some cases, such as bindings, a package is created using `stdenv.mkDerivation` and added as attribute in `all-packages.nix`. The Python bindings should be made available from `python-packages.nix`. The `toPythonModule` function takes a derivation and makes certain Python-specific modifications.

```
opencv = toPythonModule (pkgs.opencv.override {
  enablePython = true;
  pythonPackages = self;
```

```
});
```

Do pay attention to passing in the right Python version!

15.21.2.3.5. `python.buildEnv` function

Python environments can be created using the low-level `pkgs.buildEnv` function. This example shows how to create an environment that has the Pyramid Web Framework. Saving the following as `default.nix`

```
with import <nixpkgs> {};

python.buildEnv.override {
  extraLibs = [ pythonPackages.pyramid ];
  ignoreCollisions = true;
}
```

and running `nix-build` will create

```
/nix/store/cf1xhjwzmdki7fasgr4kz6di72ykicl5-python-2.7.8-env
```

with wrapped binaries in `bin/`.

You can also use the `env` attribute to create local environments with needed packages installed. This is somewhat comparable to `virtualenv`. For example, running `nix-shell` with the following `shell.nix`

```
with import <nixpkgs> {};

(python3.buildEnv.override {
  extraLibs = with python3Packages; [ numpy requests ];
}).env
```

will drop you into a shell where Python will have the specified packages in its path.

15.21.2.3.5.1. `python.buildEnv` arguments

- `extraLibs`: List of packages installed inside the environment.
- `postBuild`: Shell command executed after the build of environment.
- `ignoreCollisions`: Ignore file collisions inside the environment (default is `false`).
- `permitUserSite`: Skip setting the `PYTHONNOUSERSITE` environment variable in wrapped binaries in the environment.

15.21.2.3.6. `python.withPackages` function

The `python.withPackages` function provides a simpler interface to the `python.buildEnv` functionality. It takes a function as an argument that is passed the set of python packages and returns the list of the packages to be included in the environment. Using the `withPackages` function, the previous example for the Pyramid Web Framework environment can be written like this:

```
with import <nixpkgs> {};

python.withPackages (ps: [ps.pyramid])
```

`withPackages` passes the correct package set for the specific interpreter version as an argument to the function. In the above example, `ps` equals `pythonPackages`. But you can also easily switch to using python3:

```
with import <nixpkgs> {};

python3.withPackages (ps: [ps.pyramid])
```

Now, `ps` is set to `python3Packages`, matching the version of the interpreter.

As `python.withPackages` simply uses `python.buildEnv` under the hood, it also supports the `env` attribute. The `shell.nix` file from the previous section can thus be also written like this:

```
with import <nixpkgs> {};

(python38.withPackages (ps: [ps.numpy ps.requests])).env
```

In contrast to `python.buildEnv`, `python.withPackages` does not support the more advanced options such as `ignoreCollisions = true` or `postBuild`. If you need them, you have to use `python.buildEnv`.

Python 2 namespace packages may provide `__init__.py` that collide. In that case `python.buildEnv` should be used with `ignoreCollisions = true`.

15.21.2.3.7. Setup hooks

The following are setup hooks specifically for Python packages. Most of these are used in `buildPythonPackage`.

- `eggUnpackhook` to move an egg to the correct folder so it can be installed with the `eggInstallHook`
- `eggBuildHook` to skip building for eggs.

- `eggInstallHook` to install eggs.
- `flitBuildHook` to build a wheel using `flit`.
- `pipBuildHook` to build a wheel using `pip` and PEP 517. Note a build system (e.g. `setuptools` or `flit`) should still be added as `nativeBuildInput`.
- `pipInstallHook` to install wheels.
- `pytestCheckHook` to run tests with `pytest`. See example usage.
- `pythonCatchConflictsHook` to check whether a Python package is not already existing.
- `pythonImportsCheckHook` to check whether importing the listed modules works.
- `pythonRemoveBinBytecode` to remove bytecode from the `/bin` folder.
- `setuptoolsBuildHook` to build a wheel using `setuptools`.
- `setuptoolsCheckHook` to run tests with `python setup.py test`.
- `venvShellHook` to source a Python 3 `venv` at the `venvDir` location. A `venv` is created if it does not yet exist. `postVenvCreation` can be used to to run commands only after venv is first created.
- `wheelUnpackHook` to move a wheel to the correct folder so it can be installed with the `pipInstallHook`.

15.21.2.4. Development mode

Development or editable mode is supported. To develop Python packages `buildPythonPackage` has additional logic inside `shellPhase` to run `pip install -e . --prefix $TMPDIR/`for the package.

Warning: `shellPhase` is executed only if `setup.py` exists.

Given a `default.nix`:

```
with import <nixpkgs> {};

pythonPackages.buildPythonPackage {
  name = "myproject";
  buildInputs = with pythonPackages; [ pyramid ];

  src = ./.;
}
```

Running `nix-shell` with no arguments should give you the environment in which the package would be built with `nix-build`.

Shortcut to setup environments with C headers/libraries and Python packages:

```
nix-shell -p pythonPackages.pyramid zlib libjpeg git
```

Note: There is a boolean value `lib.inNixShell` set to `true` if nix-shell is invoked.

15.21.2.5. Tools

Packages inside nixpkgs are written by hand. However many tools exist in community to help save time. No tool is preferred at the moment.

- pypi2nix: Generate Nix expressions for your Python project. Note that sharing derivations from pypi2nix with nixpkgs is possible but not encouraged.
- nixpkgs-pytools
- poetry2nix

15.21.2.6. Deterministic builds

The Python interpreters are now built deterministically. Minor modifications had to be made to the interpreters in order to generate deterministic bytecode. This has security implications and is relevant for those using Python in a `nix-shell`.

When the environment variable `DETERMINISTIC_BUILD` is set, all bytecode will have timestamp 1. The `buildPythonPackage` function sets `DETERMINISTIC_BUILD=1` and PYTHONHASHSEED=0. Both are also exported in `nix-shell`.

15.21.2.7. Automatic tests

It is recommended to test packages as part of the build process. Source distributions (`sdist`) often include test files, but not always.

By default the command `python setup.py test` is run as part of the `checkPhase`, but often it is necessary to pass a custom `checkPhase`. An example of such a situation is when `py.test` is used.

15.21.2.7.1. Common issues

- Non-working tests can often be deselected. By default `buildPythonPackage` runs `python setup.py test`. Most Python modules follows the standard test protocol where the pytest runner can be used instead. `py.test` supports a `-k` parameter to ignore test methods or classes:

```
buildPythonPackage {
  # ...
  # assumes the tests are located in tests
  checkInputs = [ pytest ];
  checkPhase = ''
```

```
    py.test -k 'not function_name and not other_function' tests
  '';
}
```

- Tests that attempt to access `$HOME` can be fixed by using the following work-around before running tests (e.g. `preCheck`): `export HOME=$(mktemp -d)`

15.21.3. FAQ

15.21.3.1. How to solve circular dependencies?

Consider the packages A and B that depend on each other. When packaging B, a solution is to override package A not to depend on B as an input. The same should also be done when packaging A.

15.21.3.2. How to override a Python package?

We can override the interpreter and pass `packageOverrides`. In the following example we rename the `pandas` package and build it.

```
with import <nixpkgs> {};

(let
  python = let
    packageOverrides = self: super: {
      pandas = super.pandas.overridePythonAttrs(old: {name="foo";});
    };
  in pkgs.python38.override {inherit packageOverrides;};

in python.withPackages(ps: [ps.pandas])).env
```

Using `nix-build` on this expression will build an environment that contains the package `pandas` but with the new name `foo`.

All packages in the package set will use the renamed package. A typical use case is to switch to another version of a certain package. For example, in the Nixpkgs repository we have multiple versions of `django` and `scipy`. In the following example we use a different version of `scipy` and create an environment that uses it. All packages in the Python package set will now use the updated `scipy` version.

```
with import <nixpkgs> {};

( let
    packageOverrides = self: super: {
      scipy = super.scipy_0_17;
    };
```

```
  in (pkgs.python38.override {inherit packageOverrides;}).withPackages (ps:
).env
```

The requested package `blaze` depends on `pandas` which itself depends on `scipy`.

If you want the whole of Nixpkgs to use your modifications, then you can use `overlays` as explained in this manual. In the following example we build a `inkscape` using a different version of `numpy`.

```
let
  pkgs = import <nixpkgs> {};
  newpkgs = import pkgs.path { overlays = [ (self: super: {
    python38 = let
      packageOverrides = python-self: python-super: {
        numpy = python-super.numpy_1_18;
      };
    in super.python38.override {inherit packageOverrides;};
  } ) ]; };
in newpkgs.inkscape
```

15.21.3.3. `python setup.py bdist_wheel` cannot create .whl

Executing `python setup.py bdist_wheel` in a `nix-shell`fails with

```
ValueError: ZIP does not support timestamps before 1980
```

This is because files from the Nix store (which have a timestamp of the UNIX epoch of January 1, 1970) are included in the .ZIP, but .ZIP archives follow the DOS convention of counting timestamps from 1980.

The command `bdist_wheel` reads the `SOURCE_DATE_EPOCH` environment variable, which `nix-shell` sets to 1. Unsetting this variable or giving it a value corresponding to 1980 or later enables building wheels.

Use 1980 as timestamp:

```
nix-shell --run "SOURCE_DATE_EPOCH=315532800 python3 setup.py bdist_wheel"
```

or the current time:

```
nix-shell --run "SOURCE_DATE_EPOCH=$(date +%s) python3 setup.py bdist_wheel
```

or unset `SOURCE_DATE_EPOCH`:

```
nix-shell --run "unset SOURCE_DATE_EPOCH; python3 setup.py bdist_wheel"
```

15.21.3.4. `install_data` / `data_files` problems

If you get the following error:

```
could not create '/nix/store/6l1bvljpy8gazlsw2aw9skwwp4pmvyxw-python-2.7.8/
Permission denied
```

This is a known bug in `setuptools`. Setuptools `install_data` does not respect `--prefix`. An example of such package using the feature is `pkgs/tools/X11/xpra/default.nix`.

As workaround install it as an extra `preInstall` step:

```
${python.interpreter} setup.py install_data --install-dir=$out --root=$out
sed -i '/ = data\_files/d' setup.py
```

15.21.3.5. Rationale of non-existent global site-packages

On most operating systems a global `site-packages` is maintained. This however becomes problematic if you want to run multiple Python versions or have multiple versions of certain libraries for your projects. Generally, you would solve such issues by creating virtual environments using `virtualenv`.

On Nix each package has an isolated dependency tree which, in the case of Python, guarantees the right versions of the interpreter and libraries or packages are available. There is therefore no need to maintain a global `site-packages`.

If you want to create a Python environment for development, then the recommended method is to use `nix-shell`, either with or without the `python.buildEnv` function.

15.21.3.6. How to consume Python modules using pip in a virtual environment like I am used to on other Operating Systems?

While this approach is not very idiomatic from Nix perspective, it can still be useful when dealing with pre-existing projects or in situations where it's not feasible or desired to write derivations for all required dependencies.

This is an example of a `default.nix` for a `nix-shell`, which allows to consume a virtual environment created by `venv`, and install Python modules through `pip` the traditional way.

Create this `default.nix` file, together with a `requirements.txt` and simply execute `nix-shell`.

```
with import <nixpkgs> { };
```

```
let
  pythonPackages = python3Packages;
in pkgs.mkShell rec {
  name = "impurePythonEnv";
  venvDir = "./.venv";
  buildInputs = [
    # A Python interpreter including the 'venv' module is required to boots
    # the environment.
    pythonPackages.python

    # This execute some shell code to initialize a venv in $venvDir before
    # dropping into the shell
    pythonPackages.venvShellHook

    # Those are dependencies that we would like to use from nixpkgs, which
    # add them to PYTHONPATH and thus make them accessible from within the
    pythonPackages.numpy
    pythonPackages.requests

    # In this particular example, in order to compile any binary extensions
    # require, the Python modules listed in the hypothetical requirements.t
    # the following packages to be installed locally:
    taglib
    openssl
    git
    libxml2
    libxslt
    libzip
    zlib
  ];

  # Run this command, only after creating the virtual environment
  postVenvCreation = ''
    unset SOURCE_DATE_EPOCH
    pip install -r requirements.txt
  '';

  # Now we can execute any commands within the virtual environment.
  # This is optional and can be left out to run pip manually.
  postShellHook = ''
    # allow pip to install wheels
    unset SOURCE_DATE_EPOCH
```

```
  '';

}
```

In case the supplied venvShellHook is insufficient, or when Python 2 support is needed, you can define your own shell hook and adapt to your needs like in the following example:

```
with import <nixpkgs> { };

let
  venvDir = "./.venv";
  pythonPackages = python3Packages;
in pkgs.mkShell rec {
  name = "impurePythonEnv";
  buildInputs = [
    pythonPackages.python
    # Needed when using python 2.7
    # pythonPackages.virtualenv
    # ...
  ];

  # This is very close to how venvShellHook is implemented, but
  # adapted to use 'virtualenv'
  shellHook = ''
    SOURCE_DATE_EPOCH=$(date +%s)

    if [ -d "${venvDir}" ]; then
      echo "Skipping venv creation, '${venvDir}' already exists"
    else
      echo "Creating new venv environment in path: '${venvDir}'"
      # Note that the module venv was only introduced in python 3, so for 2
      # this needs to be replaced with a call to virtualenv
      ${pythonPackages.python.interpreter} -m venv "${venvDir}"
    fi

    # Under some circumstances it might be necessary to add your virtual
    # environment to PYTHONPATH, which you can do here too;
    # PYTHONPATH=$PWD/${venvDir}/${pythonPackages.python.sitePackages}/:$PY

    source "${venvDir}/bin/activate"

    # As in the previous example, this is optional.
    pip install -r requirements.txt
```

```
  '';
}
```

Note that the `pip install` is an imperative action. So every time `nix-shell` is executed it will attempt to download the Python modules listed in requirements.txt. However these will be cached locally within the `virtualenv` folder and not downloaded again.

15.21.3.7. How to override a Python package from `configuration.nix`?

If you need to change a package's attribute(s) from `configuration.nix` you could do:

```
nixpkgs.config.packageOverrides = super: {
  python = super.python.override {
    packageOverrides = python-self: python-super: {
      twisted = python-super.twisted.overrideAttrs (oldAttrs: {
        src = super.fetchPipy {
          pname = "twisted";
          version = "19.10.0";
          sha256 = "7394ba7f272ae722a74f3d969dcf599bc4ef093bc392038748a49
          extension = "tar.bz2";
        };
      });
    };
  };
};
```

`pythonPackages.twisted` is now globally overridden. All packages and also all NixOS services that reference `twisted` (such as `services.buildbot-worker`) now use the new definition. Note that `python-super` refers to the old package set and `python-self` to the new, overridden version.

To modify only a Python package set instead of a whole Python derivation, use this snippet:

```
myPythonPackages = pythonPackages.override {
  overrides = self: super: {
    twisted = ...;
  };
}
```

15.21.3.8. How to override a Python package using overlays?

Use the following overlay template:

```
self: super: {
  python = super.python.override {
    packageOverrides = python-self: python-super: {
      twisted = python-super.twisted.overrideAttrs (oldAttrs: {
        src = super.fetchPypi {
          pname = "twisted";
          version = "19.10.0";
          sha256 = "7394ba7f272ae722a74f3d969dcf599bc4ef093bc392038748a490f
          extension = "tar.bz2";
        };
      });
    };
  };
}
```

15.21.3.9. How to use Intel's MKL with numpy and scipy?

MKL can be configured using an overlay. See the section "Using overlays to configure alternatives".

15.21.3.10. What inputs do `setup_requires`, `install_requires` and `tests_require` map to?

In a `setup.py` or `setup.cfg` it is common to declare dependencies:

- `setup_requires` corresponds to `nativeBuildInputs`
- `install_requires` corresponds to `propagatedBuildInputs`
- `tests_require` corresponds to `checkInputs`

15.21.4. Contributing

15.21.4.1. Contributing guidelines

The following rules are desired to be respected:

- Python libraries are called from `python-packages.nix` and packaged with `buildPythonPackage`. The expression of a library should be in `pkgs/development/python-modules/<name>/default.nix`.
- Python applications live outside of `python-packages.nix` and are packaged with `buildPythonApplication`.
- Make sure libraries build for all Python interpreters.
- By default we enable tests. Make sure the tests are found and, in the case of libraries, are passing for all

interpreters. If certain tests fail they can be disabled individually. Try to avoid disabling the tests altogether. In any case, when you disable tests, leave a comment explaining why.

- Commit names of Python libraries should reflect that they are Python libraries, so write for example `pythonPackages.numpy: 1.11 -> 1.12`.
- Attribute names in `python-packages.nix` as well as `pname`s should match the library's name on PyPI, but be normalized according to PEP 0503. This means that characters should be converted to lowercase and `.` and _ should be replaced by a single `-` (foo-bar-baz instead of Foo__Bar.baz). If necessary, `pname` has to be given a different value within `fetchPypi`.
- Attribute names in `python-packages.nix` should be sorted alphanumerically to avoid merge conflicts and ease locating attributes.

15.22. Qt

Writing Nix expressions for Qt libraries and applications is largely similar as for other C++ software. This section assumes some knowledge of the latter. There are two problems that the Nixpkgs Qt infrastructure addresses, which are not shared by other C++ software:

1. There are usually multiple supported versions of Qt in Nixpkgs. All of a package's dependencies must be built with the same version of Qt. This is similar to the version constraints imposed on interpreted languages like Python.
2. Qt makes extensive use of runtime dependency detection. Runtime dependencies are made into build dependencies through wrappers.

15.22.1. Nix expression for a Qt package (default.nix)

```
{ stdenv, lib, qtbase, wrapQtAppsHook }: ❶

stdenv.mkDerivation {
  pname = "myapp";
  version = "1.0";

  buildInputs = [ qtbase ];
  nativeBuildInputs = [ wrapQtAppsHook ]; ❷
}
```

❶ Import Qt modules directly, that is: `qtbase`, `qtdeclarative`, etc. *Do not* import Qt package sets such as `qt5` because the Qt versions of dependencies may not be coherent, causing build and runtime failures.

❷ All Qt packages must include `wrapQtAppsHook` in `nativeBuildInputs`, or you must explicitly

set `dontWrapQtApps`.

15.22.2. Locating runtime dependencies

Qt applications must be wrapped to find runtime dependencies. Include `wrapQtAppsHook` in `nativeBuildInputs`:

```
{ stdenv, wrapQtAppsHook }:

stdenv.mkDerivation {
  # ...
  nativeBuildInputs = [ wrapQtAppsHook ];
}
```

Add entries to `qtWrapperArgs` are to modify the wrappers created by `wrapQtAppsHook`:

```
{ stdenv, wrapQtAppsHook }:

stdenv.mkDerivation {
  # ...
  nativeBuildInputs = [ wrapQtAppsHook ];
  qtWrapperArgs = [ ''--prefix PATH : /path/to/bin'' ];
}
```

The entries are passed as arguments to wrapProgram.

Set `dontWrapQtApps` to stop applications from being wrapped automatically. Wrap programs manually with `wrapQtApp`, using the syntax of wrapProgram:

```
{ stdenv, lib, wrapQtAppsHook }:

stdenv.mkDerivation {
  # ...
  nativeBuildInputs = [ wrapQtAppsHook ];
  dontWrapQtApps = true;
  preFixup = ''
      wrapQtApp "$out/bin/myapp" --prefix PATH : /path/to/bin
  '';
}
```

Note: `wrapQtAppsHook` ignores files that are non-ELF executables. This means that scripts won't be automatically wrapped so you'll need to manually wrap them as previously mentioned. An

example of when you'd always need to do this is with Python applications that use PyQt.

15.22.3. Adding a library to Nixpkgs

Add Qt libraries to `qt5-packages.nix` to make them available for every supported Qt version.

15.22.3.1. Example adding a Qt library

The following represents the contents of `qt5-packages.nix`.

```
{
  # ...

  mylib = callPackage ../path/to/mylib {};

  # ...
}
```

Libraries are built with every available version of Qt. Use the `meta.broken` attribute to disable the package for unsupported Qt versions:

```
{ stdenv, lib, qtbase }:

stdenv.mkDerivation {
  # ...
  # Disable this library with Qt < 5.9.0
  meta.broken = lib.versionOlder qtbase.version "5.9.0";
}
```

15.22.4. Adding an application to Nixpkgs

Add Qt applications to `qt5-packages.nix`. Add an alias to `all-packages.nix` to select the Qt 5 version used for the application.

15.22.4.1. Example adding a Qt application

The following represents the contents of `qt5-packages.nix`.

```
{
  # ...

  myapp = callPackage ../path/to/myapp {};
```

```
  # ...
}
```

The following represents the contents of `all-packages.nix`.

```
{
  # ...

  myapp = libsForQt5.myapp;

  # ...
}
```

15.23. R

15.23.1. Installation

Define an environment for R that contains all the libraries that you'd like to use by adding the following snippet to your $HOME/.config/nixpkgs/config.nix file:

```
{
    packageOverrides = super: let self = super.pkgs; in
    {

        rEnv = super.rWrapper.override {
            packages = with self.rPackages; [
                devtools
                ggplot2
                reshape2
                yaml
                optparse
                ];
        };
    };
}
```

Then you can use `nix-env -f "<nixpkgs>" -iA rEnv` to install it into your user profile. The set of available libraries can be discovered by running the command `nix-env -f "<nixpkgs>" -qaP -A rPackages`. The first column from that output is the name that has to be passed to rWrapper in the code snipped above.

However, if you'd like to add a file to your project source to make the environment available for other contributors, you can create a `default.nix` file like so:

```
with import <nixpkgs> {};
{
  myProject = stdenv.mkDerivation {
    name = "myProject";
    version = "1";
    src = if lib.inNixShell then null else nix;

    buildInputs = with rPackages; [
      R
      ggplot2
      knitr
    ];
  };
}
```

and then run `nix-shell .` to be dropped into a shell with those packages available.

15.23.2. RStudio

RStudio uses a standard set of packages and ignores any custom R environments or installed packages you may have. To create a custom environment, see `rstudioWrapper`, which functions similarly to `rWrapper`:

```
{
    packageOverrides = super: let self = super.pkgs; in
    {

        rstudioEnv = super.rstudioWrapper.override {
            packages = with self.rPackages; [
                dplyr
                ggplot2
                reshape2
                ];
        };
    };
}
```

Then like above, `nix-env -f "<nixpkgs>" -iA rstudioEnv` will install this into your user profile.

Alternatively, you can create a self-contained `shell.nix` without the need to modify any configuration files:

```
{ pkgs ? import <nixpkgs> {}
```

```
}:

pkgs.rstudioWrapper.override {
  packages = with pkgs.rPackages; [ dplyr ggplot2 reshape2 ];
}
```

Executing `nix-shell` will then drop you into an environment equivalent to the one above. If you need additional packages just add them to the list and re-enter the shell.

15.23.3. Updating the package set

```
nix-shell generate-shell.nix

Rscript generate-r-packages.R cran  > cran-packages.nix.new
mv cran-packages.nix.new cran-packages.nix

Rscript generate-r-packages.R bioc  > bioc-packages.nix.new
mv bioc-packages.nix.new bioc-packages.nix

Rscript generate-r-packages.R bioc-annotation > bioc-annotation-packages.ni
mv bioc-annotation-packages.nix.new bioc-annotation-packages.nix

Rscript generate-r-packages.R bioc-experiment > bioc-experiment-packages.ni
mv bioc-experiment-packages.nix.new bioc-experiment-packages.nix
```

`generate-r-packages.R <repo>` reads `<repo>-packages.nix`, therefor the renaming.

15.23.4. Testing if the Nix-expression could be evaluated

```
nix-build test-evaluation.nix --dry-run
```

If this exits fine, the expression is ok. If not, you have to edit `default.nix`

15.24. Ruby

15.24.1. Using Ruby

Several versions of Ruby interpreters are available on Nix, as well as over 250 gems and many applications written in Ruby. The attribute `ruby` refers to the default Ruby interpreter, which is currently MRI 2.6. It's also possible to refer to specific versions, e.g. `ruby_2_y`, `jruby`, or `mruby`.

In the Nixpkgs tree, Ruby packages can be found throughout, depending on what they do, and are called

from the main package set. Ruby gems, however are separate sets, and there's one default set for each interpreter (currently MRI only).

There are two main approaches for using Ruby with gems. One is to use a specifically locked `Gemfile` for an application that has very strict dependencies. The other is to depend on the common gems, which we'll explain further down, and rely on them being updated regularly.

The interpreters have common attributes, namely `gems`, and `withPackages`. So you can refer to `ruby.gems.nokogiri`, or `ruby_2_6.gems.nokogiri` to get the Nokogiri gem already compiled and ready to use.

Since not all gems have executables like `nokogiri`, it's usually more convenient to use the `withPackages` function like this: `ruby.withPackages (p: with p; [ nokogiri ])`. This will also make sure that the Ruby in your environment will be able to find the gem and it can be used in your Ruby code (for example via `ruby` or `irb` executables) via `require "nokogiri"` as usual.

15.24.1.1. Temporary Ruby environment with `nix-shell`

Rather than having a single Ruby environment shared by all Ruby development projects on a system, Nix allows you to create separate environments per project. `nix-shell` gives you the possibility to temporarily load another environment akin to a combined `chruby` or `rvm` and `bundle exec`.

There are two methods for loading a shell with Ruby packages. The first and recommended method is to create an environment with `ruby.withPackages` and load that.

```
$ nix-shell -p "ruby.withPackages (ps: with ps; [ nokogiri pry ])"
```

The other method, which is not recommended, is to create an environment and list all the packages directly.

```
$ nix-shell -p ruby.gems.nokogiri ruby.gems.pry
```

Again, it's possible to launch the interpreter from the shell. The Ruby interpreter has the attribute `gems` which contains all Ruby gems for that specific interpreter.

15.24.1.1.1. Load Ruby environment from `.nix` expression

As explained in the Nix manual, `nix-shell` can also load an expression from a `.nix` file. Say we want to have Ruby 2.6, `nokogori`, and `pry`. Consider a `shell.nix` file with:

```
with import <nixpkgs> {};
ruby.withPackages (ps: with ps; [ nokogiri pry ])
```

What's happening here?

1. We begin with importing the Nix Packages collections. `import <nixpkgs>` imports the `<nixpkgs>` function, `{}` calls it and the `with` statement brings all attributes of `nixpkgs` in the local scope. These attributes form the main package set.

2. Then we create a Ruby environment with the `withPackages` function.

3. The `withPackages` function expects us to provide a function as an argument that takes the set of all ruby gems and returns a list of packages to include in the environment. Here, we select the packages `nokogiri` and `pry` from the package set.

15.24.1.1.2. Execute command with `--run`

A convenient flag for `nix-shell` is `--run`. It executes a command in the `nix-shell`. We can e.g. directly open a `pry` REPL:

```
$ nix-shell -p "ruby.withPackages (ps: with ps; [ nokogiri pry ])" --run "p
```

Or immediately require `nokogiri` in pry:

```
$ nix-shell -p "ruby.withPackages (ps: with ps; [ nokogiri pry ])" --run "p
```

Or run a script using this environment:

```
$ nix-shell -p "ruby.withPackages (ps: with ps; [ nokogiri pry ])" --run "r
```

15.24.1.1.3. Using `nix-shell` as shebang

In fact, for the last case, there is a more convenient method. You can add a shebang to your script specifying which dependencies `nix-shell` needs. With the following shebang, you can just execute `./example.rb`, and it will run with all dependencies.

```
#! /usr/bin/env nix-shell
#! nix-shell -i ruby -p "ruby.withPackages (ps: with ps; [ nokogiri rest-cl

require 'nokogiri'
require 'rest-client'

body = RestClient.get('http://example.com').body
puts Nokogiri::HTML(body).at('h1').text
```

15.24.2. Developing with Ruby

15.24.2.1. Using an existing Gemfile

In most cases, you'll already have a `Gemfile.lock` listing all your dependencies. This can be used to generate a `gemset.nix` which is used to fetch the gems and combine them into a single environment. The reason why you need to have a separate file for this, is that Nix requires you to have a checksum for each input to your build. Since the `Gemfile.lock` that `bundler` generates doesn't provide us with checksums, we have to first download each gem, calculate its SHA256, and store it in this separate file.

So the steps from having just a `Gemfile` to a `gemset.nix` are:

```
$ bundle lock
$ bundix
```

If you already have a `Gemfile.lock`, you can simply run `bundix` and it will work the same.

To update the gems in your `Gemfile.lock`, you may use the `bundix -l` flag, which will create a new `Gemfile.lock` in case the `Gemfile` has a more recent time of modification.

Once the `gemset.nix` is generated, it can be used in a `bundlerEnv` derivation. Here is an example you could use for your `shell.nix`:

```
# ...
let
  gems = bundlerEnv {
    name = "gems-for-some-project";
    gemdir = ./.;
  };
in mkShell { packages = [ gems gems.wrappedRuby ]; }
```

With this file in your directory, you can run `nix-shell` to build and use the gems. The important parts here are `bundlerEnv` and `wrappedRuby`.

The `bundlerEnv` is a wrapper over all the gems in your gemset. This means that all the `/lib` and `/bin` directories will be available, and the executables of all gems (even of indirect dependencies) will end up in your `$PATH`. The `wrappedRuby` provides you with all executables that come with Ruby itself, but wrapped so they can easily find the gems in your gemset.

One common issue that you might have is that you have Ruby 2.6, but also `bundler` in your gemset. That leads to a conflict for `/bin/bundle` and `/bin/bundler`. You can resolve this by wrapping either your Ruby or your gems in a `lowPrio` call. So in order to give the `bundler` from your gemset priority, it would be used like this:

```
# ...
```

```
mkShell { buildInputs = [ gems (lowPrio gems.wrappedRuby) ]; }
```

15.24.2.2. Gem-specific configurations and workarounds

In some cases, especially if the gem has native extensions, you might need to modify the way the gem is built.

This is done via a common configuration file that includes all of the workarounds for each gem.

This file lives at `/pkgs/development/ruby-modules/gem-config/default.nix`, since it already contains a lot of entries, it should be pretty easy to add the modifications you need for your needs.

In the meanwhile, or if the modification is for a private gem, you can also add the configuration to only your own environment.

Two places that allow this modification are the `ruby` derivation, or `bundlerEnv`.

Here's the `ruby` one:

```
{ pg_version ? "10", pkgs ? import <nixpkgs> { } }:
let
  myRuby = pkgs.ruby.override {
    defaultGemConfig = pkgs.defaultGemConfig // {
      pg = attrs: {
        buildFlags =
        [ "--with-pg-config=${pkgs."postgresql_${pg_version}"}/bin/pg_confi
      };
    };
  };
in myRuby.withPackages (ps: with ps; [ pg ])
```

And an example with `bundlerEnv`:

```
{ pg_version ? "10", pkgs ? import <nixpkgs> { } }:
let
  gems = pkgs.bundlerEnv {
    name = "gems-for-some-project";
    gemdir = ./.;
    gemConfig = pkgs.defaultGemConfig // {
      pg = attrs: {
        buildFlags =
        [ "--with-pg-config=${pkgs."postgresql_${pg_version}"}/bin/pg_confi
      };
    };
  };
```

```
in mkShell { buildInputs = [ gems gems.wrappedRuby ]; }
```

And finally via overlays:

```
{ pg_version ? "10" }:
let
  pkgs = import <nixpkgs> {
    overlays = [
      (self: super: {
        defaultGemConfig = super.defaultGemConfig // {
          pg = attrs: {
            buildFlags = [
              "--with-pg-config=${
                pkgs."postgresql_${pg_version}"
              }/bin/pg_config"
            ];
          };
        };
      })
    ];
  };
in pkgs.ruby.withPackages (ps: with ps; [ pg ])
```

Then we can get whichever postgresql version we desire and the `pg` gem will always reference it correctly:

```
$ nix-shell --argstr pg_version 9_4 --run 'ruby -rpg -e "puts PG.library_ve
90421

$ nix-shell --run 'ruby -rpg -e "puts PG.library_version"'
100007
```

Of course for this use-case one could also use overlays since the configuration for `pg` depends on the `postgresql` alias, but for demonstration purposes this has to suffice.

15.24.2.3. Adding a gem to the default gemset

Now that you know how to get a working Ruby environment with Nix, it's time to go forward and start actually developing with Ruby. We will first have a look at how Ruby gems are packaged on Nix. Then, we will look at how you can use development mode with your code.

All gems in the standard set are automatically generated from a single `Gemfile`. The dependency

resolution is done with `bundler` and makes it more likely that all gems are compatible to each other.

In order to add a new gem to nixpkgs, you can put it into the `/pkgs/development/ruby-modules/with-packages/Gemfile` and run `./maintainers/scripts/update-ruby-packages`.

To test that it works, you can then try using the gem with:

```
NIX_PATH=nixpkgs=$PWD nix-shell -p "ruby.withPackages (ps: with ps; [ name-
```

15.24.2.4. Packaging applications

A common task is to add a ruby executable to nixpkgs, popular examples would be `chef`, `jekyll`, or `sass`. A good way to do that is to use the `bundlerApp` function, that allows you to make a package that only exposes the listed executables, otherwise the package may cause conflicts through common paths like `bin/rake` or `bin/bundler` that aren't meant to be used.

The absolute easiest way to do that is to write a `Gemfile` along these lines:

```
source 'https://rubygems.org' do
  gem 'mdl'
end
```

If you want to package a specific version, you can use the standard Gemfile syntax for that, e.g. `gem 'mdl', '0.5.0'`, but if you want the latest stable version anyway, it's easier to update by simply running the `bundle lock` and `bundix` steps again.

Now you can also make a `default.nix` that looks like this:

```
{ bundlerApp }:

bundlerApp {
  pname = "mdl";
  gemdir = ./.;
  exes = [ "mdl" ];
}
```

All that's left to do is to generate the corresponding `Gemfile.lock` and `gemset.nix` as described above in the `Using an existing Gemfile` section.

15.24.2.4.1. Packaging executables that require wrapping

Sometimes your app will depend on other executables at runtime, and tries to find it through the `PATH` environment variable.

In this case, you can provide a `postBuild` hook to `bundlerApp` that wraps the gem in another script that prefixes the `PATH`.

Of course you could also make a custom `gemConfig` if you know exactly how to patch it, but it's usually much easier to maintain with a simple wrapper so the patch doesn't have to be adjusted for each version.

Here's another example:

```
{ lib, bundlerApp, makeWrapper, git, gnutar, gzip }:

bundlerApp {
  pname = "r10k";
  gemdir = ./.;
  exes = [ "r10k" ];

  buildInputs = [ makeWrapper ];

  postBuild = ''
    wrapProgram $out/bin/r10k --prefix PATH : ${lib.makeBinPath [ git gnuta
  '';
}
```

15.25. Rust

To install the rust compiler and cargo put

```
environment.systemPackages = [
  rustc
  cargo
];
```

into your `configuration.nix` or bring them into scope with `nix-shell -p rustc cargo`.

For other versions such as daily builds (beta and nightly), use either `rustup` from nixpkgs (which will manage the rust installation in your home directory), or use Mozilla's Rust nightlies overlay.

15.25.1. Compiling Rust applications with Cargo

Rust applications are packaged by using the `buildRustPackage` helper from `rustPlatform`:

```
{ lib, rustPlatform }:

rustPlatform.buildRustPackage rec {
```

```
  pname = "ripgrep";
  version = "12.1.1";

  src = fetchFromGitHub {
    owner = "BurntSushi";
    repo = pname;
    rev = version;
    sha256 = "1hqps7l5qrjh9f914r5i6kmcz6f1yb951nv4lby0cjnp5l253kps";
  };

  cargoSha256 = "03wf9r2csi6jpa7v5sw5lpxkrk4wfzwmzx7k3991q3bdjzcwnnwp";

  meta = with lib; {
    description = "A fast line-oriented regex search tool, similar to ag an
    homepage = "https://github.com/BurntSushi/ripgrep";
    license = licenses.unlicense;
    maintainers = [ maintainers.tailhook ];
  };
}
```

`buildRustPackage` requires either the `cargoSha256` or the `cargoHash` attribute which is computed over all crate sources of this package. `cargoHash256` is used for traditional Nix SHA-256 hashes, such as the one in the example above. `cargoHash` should instead be used for SRI hashes. For example:

```
cargoHash = "sha256-l1vL2ZdtDRxSGvP0X/l3nMw8+6WF67KPutJEzUROjg8=";
```

Both types of hashes are permitted when contributing to nixpkgs. The Cargo hash is obtained by inserting a fake checksum into the expression and building the package once. The correct checksum can then be taken from the failed build. A fake hash can be used for `cargoSha256` as follows:

```
cargoSha256 = lib.fakeSha256;
```

For `cargoHash` you can use:

```
cargoHash = lib.fakeHash;
```

Per the instructions in the Cargo Book best practices guide, Rust applications should always commit the `Cargo.lock` file in git to ensure a reproducible build. However, a few packages do not, and Nix depends on this file, so if it is missing you can use `cargoPatches` to apply it in the `patchPhase`. Consider sending a PR upstream with a note to the maintainer describing why it's important to include in the application.

The fetcher will verify that the `Cargo.lock` file is in sync with the `src` attribute, and fail the build if not. It will also will compress the vendor directory into a tar.gz archive.

The tarball with vendored dependencies contains a directory with the package's `name`, which is normally composed of `pname` and `version`. This means that the vendored dependencies hash (`cargoSha256`/`cargoHash`) is dependent on the package name and version. The `cargoDepsName` attribute can be used to use another name for the directory of vendored dependencies. For example, the hash can be made invariant to the version by setting `cargoDepsName` to `pname`:

```
rustPlatform.buildRustPackage rec {
  pname = "broot";
  version = "1.2.0";

  src = fetchCrate {
    inherit pname version;
    sha256 = "1mqaynrqaas82f5957lx31x80v74zwmwmjxxlbywajb61vh00d38";
  };

  cargoHash = "sha256-JmBZcDVYJaK1cK05cxx5BrnGWp4t8ca6FLUbvIot67s=";
  cargoDepsName = pname;

  # ...
}
```

15.25.1.1. Cross compilation

By default, Rust packages are compiled for the host platform, just like any other package is. The `--target` passed to rust tools is computed from this. By default, it takes the `stdenv.hostPlatform.config` and replaces components where they are known to differ. But there are ways to customize the argument:

- To choose a different target by name, define `stdenv.hostPlatform.rustc.config` as that name (a string), and that name will be used instead.

  For example:

  ```
  import <nixpkgs> {
    crossSystem = (import <nixpkgs/lib>).systems.examples.armhf-embedded //
      rustc.config = "thumbv7em-none-eabi";
    };
  }
  ```

  will result in:

```
--target thumbv7em-none-eabi
```

- To pass a completely custom target, define `stdenv.hostPlatform.rustc.config` with its name, and `stdenv.hostPlatform.rustc.platform` with the value. The value will be serialized to JSON in a file called `${stdenv.hostPlatform.rustc.config}.json`, and the path of that file will be used instead.

  For example:

  ```
  import <nixpkgs> {
    crossSystem = (import <nixpkgs/lib>).systems.examples.armhf-embedded //
      rustc.config = "thumb-crazy";
      rustc.platform = { foo = ""; bar = ""; };
    };
  }
  ```

  will result in:

  ```
  --target /nix/store/asdfasdfsadf-thumb-crazy.json # contains {"foo":"","ba
  ```

Finally, as an ad-hoc escape hatch, a computed target (string or JSON file path) can be passed directly to `buildRustPackage`:

```
pkgs.rustPlatform.buildRustPackage {
  /* ... */
  target = "x86_64-fortanix-unknown-sgx";
}
```

This is useful to avoid rebuilding Rust tools, since they are actually target agnostic and don't need to be rebuilt. But in the future, we should always build the Rust tools and standard library crates separately so there is no reason not to take the `stdenv.hostPlatform.rustc`-modifying approach, and the ad-hoc escape hatch to `buildRustPackage` can be removed.

Note that currently custom targets aren't compiled with `std`, so `cargo test` will fail. This can be ignored by adding `doCheck = false;` to your derivation.

15.25.1.2. Running package tests

When using `buildRustPackage`, the `checkPhase` is enabled by default and runs `cargo test` on the package to build. To make sure that we don't compile the sources twice and to actually test the artifacts that will be used at runtime, the tests will be ran in the `release` mode by default.

However, in some cases the test-suite of a package doesn't work properly in the `release` mode. For

these situations, the mode for `checkPhase` can be changed like so:

```
rustPlatform.buildRustPackage {
  /* ... */
  checkType = "debug";
}
```

Please note that the code will be compiled twice here: once in `release` mode for the `buildPhase`, and again in `debug` mode for the `checkPhase`.

Test flags, e.g., `--features xxx/yyy`, can be passed to `cargo test` via the `cargoTestFlags` attribute.

Another attribute, called `checkFlags`, is used to pass arguments to the test binary itself, as stated (here) [https://doc.rust-lang.org/cargo/commands/cargo-test.html].

15.25.1.2.1. Tests relying on the structure of the `target/` directory

Some tests may rely on the structure of the `target/` directory. Those tests are likely to fail because we use `cargo --target` during the build. This means that the artifacts are stored in `target/<architecture>/release/`, rather than in `target/release/`.

This can only be worked around by patching the affected tests accordingly.

15.25.1.2.2. Disabling package-tests

In some instances, it may be necessary to disable testing altogether (with `doCheck = false;`):

- If no tests exist – the `checkPhase` should be explicitly disabled to skip unnecessary build steps to speed up the build.
- If tests are highly impure (e.g. due to network usage).

There will obviously be some corner-cases not listed above where it's sensible to disable tests. The above are just guidelines, and exceptions may be granted on a case-by-case basis.

However, please check if it's possible to disable a problematic subset of the test suite and leave a comment explaining your reasoning.

15.25.1.2.3. Setting `test-threads`

`buildRustPackage` will use parallel test threads by default, sometimes it may be necessary to disable this so the tests run consecutively.

```
rustPlatform.buildRustPackage {
  /* ... */
```

```
  dontUseCargoParallelTests = true;
}
```

15.25.1.3. Building a package in debug mode

By default, `buildRustPackage` will use `release` mode for builds. If a package should be built in `debug` mode, it can be configured like so:

```
rustPlatform.buildRustPackage {
  /* ... */
  buildType = "debug";
}
```

In this scenario, the `checkPhase` will be ran in `debug` mode as well.

15.25.1.4. Custom `build/install`-procedures

Some packages may use custom scripts for building/installing, e.g. with a `Makefile`. In these cases, it's recommended to override the `buildPhase/installPhase/checkPhase`.

Otherwise, some steps may fail because of the modified directory structure of `target/`.

15.25.1.5. Building a crate with an absent or out-of-date Cargo.lock file

`buildRustPackage` needs a `Cargo.lock` file to get all dependencies in the source code in a reproducible way. If it is missing or out-of-date one can use the `cargoPatches` attribute to update or add it.

```
rustPlatform.buildRustPackage rec {
  (...)
  cargoPatches = [
    # a patch file to add/update Cargo.lock in the source code
    ./add-Cargo.lock.patch
  ];
}
```

15.25.2. Compiling non-Rust packages that include Rust code

Several non-Rust packages incorporate Rust code for performance- or security-sensitive parts. `rustPlatform` exposes several functions and hooks that can be used to integrate Cargo in non-Rust packages.

15.25.2.1. Vendoring of dependencies

Since network access is not allowed in sandboxed builds, Rust crate dependencies need to be retrieved using a fetcher. `rustPlatform` provides the `fetchCargoTarball` fetcher, which vendors all dependencies of a crate. For example, given a source path `src` containing `Cargo.toml` and `Cargo.lock`, `fetchCargoTarball` can be used as follows:

```
cargoDeps = rustPlatform.fetchCargoTarball {
  inherit src;
  hash = "sha256-BoHIN/519Top1NUBjpB/oEMqi86Omt3zTQcXFWqrek0=";
};
```

The `src` attribute is required, as well as a hash specified through one of the `sha256` or `hash` attributes. The following optional attributes can also be used:

- `name`: the name that is used for the dependencies tarball. If `name` is not specified, then the name `cargo-deps` will be used.
- `sourceRoot`: when the `Cargo.lock`/`Cargo.toml` are in a subdirectory, `sourceRoot` specifies the relative path to these files.
- `patches`: patches to apply before vendoring. This is useful when the `Cargo.lock`/`Cargo.toml` files need to be patched before vendoring.

15.25.2.2. Hooks

`rustPlatform` provides the following hooks to automate Cargo builds:

- `cargoSetupHook`: configure Cargo to use depenencies vendored through `fetchCargoTarball`. This hook uses the `cargoDeps` environment variable to find the vendored dependencies. If a project already vendors its dependencies, the variable `cargoVendorDir` can be used instead. When the `Cargo.toml`/`Cargo.lock` files are not in `sourceRoot`, then the optional `cargoRoot` is used to specify the Cargo root directory relative to `sourceRoot`.
- `cargoBuildHook`: use Cargo to build a crate. If the crate to be built is a crate in e.g. a Cargo workspace, the relative path to the crate to build can be set through the optional `buildAndTestSubdir` environment variable. Additional Cargo build flags can be passed through `cargoBuildFlags`.
- `maturinBuildHook`: use Maturin to build a Python wheel. Similar to `cargoBuildHook`, the optional variable `buildAndTestSubdir` can be used to build a crate in a Cargo workspace. Additional maturin flags can be passed through `maturinBuildFlags`.
- `cargoCheckHook`: run tests using Cargo. The build type for checks can be set using `cargoCheckType`. Additional flags can be passed to the tests using `checkFlags` and `checkFlagsArray`. By default, tests are run in parallel. This can be disabled by setting `dontUseCargoParallelTests`.
- `cargoInstallHook`: install binaries and static/shared libraries that were built using

`cargoBuildHook`.

15.25.2.3. Examples

15.25.2.3.1. Python package using `setuptools-rust`

For Python packages using `setuptools-rust`, you can use `fetchCargoTarball` and `cargoSetupHook` to retrieve and set up Cargo dependencies. The build itself is then performed by `buildPythonPackage`.

The following example outlines how the `tokenizers` Python package is built. Since the Python package is in the `source/bindings/python` directory of the *tokenizers* project's source archive, we use `sourceRoot` to point the tooling to this directory:

```
{ fetchFromGitHub
, buildPythonPackage
, rustPlatform
, setuptools-rust
}:

buildPythonPackage rec {
  pname = "tokenizers";
  version = "0.10.0";

  src = fetchFromGitHub {
    owner = "huggingface";
    repo = pname;
    rev = "python-v${version}";
    hash = "sha256-rQ2hRV52naEf6PvRsWVCTN7B1oXAQGmnpJw4iIdhamw=";
  };

  cargoDeps = rustPlatform.fetchCargoTarball {
    inherit src sourceRoot;
    name = "${pname}-${version}";
    hash = "sha256-BoHIN/519Top1NUBjpB/oEMqi86Omt3zTQcXFWqrek0=";
  };

  sourceRoot = "source/bindings/python";

  nativeBuildInputs = [ setuptools-rust ] ++ (with rustPlatform; [
    cargoSetupHook
    rust.cargo
    rust.rustc
  ]);
```

```
  # ...
}
```

In some projects, the Rust crate is not in the main Python source directory. In such cases, the `cargoRoot` attribute can be used to specify the crate's directory relative to `sourceRoot`. In the following example, the crate is in `src/rust`, as specified in the `cargoRoot` attribute. Note that we also need to specify the correct path for `fetchCargoTarball`.

```
{ buildPythonPackage
, fetchPypi
, rustPlatform
, setuptools-rust
, openssl
}:

buildPythonPackage rec {
  pname = "cryptography";
  version = "3.4.2"; # Also update the hash in vectors.nix

  src = fetchPypi {
    inherit pname version;
    sha256 = "1i1mx5y9hkyfi9jrrkcw804hmkcglxi6rmf7vin7jfnbr2bf4q64";
  };

  cargoDeps = rustPlatform.fetchCargoTarball {
    inherit src;
    sourceRoot = "${pname}-${version}/${cargoRoot}";
    name = "${pname}-${version}";
    hash = "sha256-PS562W4L1NimqDV2H0jl5vYhL08H9est/pbIxSdYVfo=";
  };

  cargoRoot = "src/rust";

  # ...
}
```

15.25.2.3.2. Python package using `maturin`

Python packages that use Maturin can be built with `fetchCargoTarball`, `cargoSetupHook`, and `maturinBuildHook`. For example, the following (partial) derivation builds the `retworkx` Python package. `fetchCargoTarball` and `cargoSetupHook` are used to fetch and set up the crate

dependencies. `maturinBuildHook` is used to perform the build.

```
{ lib
, buildPythonPackage
, rustPlatform
, fetchFromGitHub
}:

buildPythonPackage rec {
  pname = "retworkx";
  version = "0.6.0";

  src = fetchFromGitHub {
    owner = "Qiskit";
    repo = "retworkx";
    rev = version;
    sha256 = "11n30ldg3y3y6qxg3hbj837pnbwjkqw3nxq6frds647mmmprrd20";
  };

  cargoDeps = rustPlatform.fetchCargoTarball {
    inherit src;
    name = "${pname}-${version}";
    hash = "sha256-heOBK8qi2nuc/Ib+I/vLzZ1fUUD/G/KTw9d7M4Hz5O0=";
  };

  format = "pyproject";

  nativeBuildInputs = with rustPlatform; [ cargoSetupHook maturinBuildHook

  # ...
}
```

15.25.3. Compiling Rust crates using Nix instead of Cargo

15.25.3.1. Simple operation

When run, `cargo build` produces a file called `Cargo.lock`, containing pinned versions of all dependencies. Nixpkgs contains a tool called `carnix` (`nix-env -iA nixos.carnix`), which can be used to turn a `Cargo.lock` into a Nix expression.

That Nix expression calls `rustc` directly (hence bypassing Cargo), and can be used to compile a crate and all its dependencies. Here is an example for a minimal `hello` crate:

```
$ cargo new hello
$ cd hello
$ cargo build
 Compiling hello v0.1.0 (file:///tmp/hello)
  Finished dev [unoptimized + debuginfo] target(s) in 0.20 secs
$ carnix -o hello.nix --src ./. Cargo.lock --standalone
$ nix-build hello.nix -A hello_0_1_0
```

Now, the file produced by the call to `carnix`, called `hello.nix`, looks like:

```
# Generated by carnix 0.6.5: carnix -o hello.nix --src ./. Cargo.lock --sta
{ stdenv, buildRustCrate, fetchgit }:
let kernel = stdenv.buildPlatform.parsed.kernel.name;
    # ... (content skipped)
in
rec {
  hello = f: hello_0_1_0 { features = hello_0_1_0_features { hello_0_1_0 =
  hello_0_1_0_ = { dependencies?[], buildDependencies?[], features?[] }: bu
    crateName = "hello";
    version = "0.1.0";
    authors = [ "pe@pijul.org <pe@pijul.org>" ];
    src = ./.;
    inherit dependencies buildDependencies features;
  };
  hello_0_1_0 = { features?(hello_0_1_0_features {}) }: hello_0_1_0_ {};
  hello_0_1_0_features = f: updateFeatures f (rec {
        hello_0_1_0.default = (f.hello_0_1_0.default or true);
    }) [ ];
}
```

In particular, note that the argument given as `--src` is copied verbatim to the source. If we look at a more complicated dependencies, for instance by adding a single line `libc="*"` to our `Cargo.toml`, we first need to run `cargo build` to update the `Cargo.lock`. Then, `carnix` needs to be run again, and produces the following nix file:

```
# Generated by carnix 0.6.5: carnix -o hello.nix --src ./. Cargo.lock --sta
{ stdenv, buildRustCrate, fetchgit }:
let kernel = stdenv.buildPlatform.parsed.kernel.name;
    # ... (content skipped)
in
rec {
  hello = f: hello_0_1_0 { features = hello_0_1_0_features { hello_0_1_0 =
```

```
  hello_0_1_0_ = { dependencies?[], buildDependencies?[], features?[] }: bu
    crateName = "hello";
    version = "0.1.0";
    authors = [ "pe@pijul.org <pe@pijul.org>" ];
    src = ./.;
    inherit dependencies buildDependencies features;
  };
  libc_0_2_36_ = { dependencies?[], buildDependencies?[], features?[] }: bu
    crateName = "libc";
    version = "0.2.36";
    authors = [ "The Rust Project Developers" ];
    sha256 = "01633h4yfqm0s302fm0dlba469bx8y6cs4nqc8bqrmjqxfxn515l";
    inherit dependencies buildDependencies features;
  };
  hello_0_1_0 = { features?(hello_0_1_0_features {}) }: hello_0_1_0_ {
    dependencies = mapFeatures features ([ libc_0_2_36 ]);
  };
  hello_0_1_0_features = f: updateFeatures f (rec {
    hello_0_1_0.default = (f.hello_0_1_0.default or true);
    libc_0_2_36.default = true;
  }) [ libc_0_2_36_features ];
  libc_0_2_36 = { features?(libc_0_2_36_features {}) }: libc_0_2_36_ {
    features = mkFeatures (features.libc_0_2_36 or {});
  };
  libc_0_2_36_features = f: updateFeatures f (rec {
    libc_0_2_36.default = (f.libc_0_2_36.default or true);
    libc_0_2_36.use_std =
      (f.libc_0_2_36.use_std or false) ||
      (f.libc_0_2_36.default or false) ||
      (libc_0_2_36.default or false);
  }) [];
}
```

Here, the `libc` crate has no `src` attribute, so `buildRustCrate` will fetch it from crates.io. A `sha256` attribute is still needed for Nix purity.

15.25.3.2. Handling external dependencies

Some crates require external libraries. For crates from crates.io, such libraries can be specified in `defaultCrateOverrides` package in nixpkgs itself.

Starting from that file, one can add more overrides, to add features or build inputs by overriding the hello crate in a seperate file.

```
with import <nixpkgs> {};
((import ./hello.nix).hello {}).override {
  crateOverrides = defaultCrateOverrides // {
    hello = attrs: { buildInputs = [ openssl ]; };
  };
}
```

Here, `crateOverrides` is expected to be a attribute set, where the key is the crate name without version number and the value a function. The function gets all attributes passed to `buildRustCrate` as first argument and returns a set that contains all attribute that should be overwritten.

For more complicated cases, such as when parts of the crate's derivation depend on the crate's version, the `attrs` argument of the override above can be read, as in the following example, which patches the derivation:

```
with import <nixpkgs> {};
((import ./hello.nix).hello {}).override {
  crateOverrides = defaultCrateOverrides // {
    hello = attrs: lib.optionalAttrs (lib.versionAtLeast attrs.version "1.0
      postPatch = ''
        substituteInPlace lib/zoneinfo.rs \
          --replace "/usr/share/zoneinfo" "${tzdata}/share/zoneinfo"
      '';
    };
  };
}
```

Another situation is when we want to override a nested dependency. This actually works in the exact same way, since the `crateOverrides` parameter is forwarded to the crate's dependencies. For instance, to override the build inputs for crate `libc` in the example above, where `libc` is a dependency of the main crate, we could do:

```
with import <nixpkgs> {};
((import hello.nix).hello {}).override {
  crateOverrides = defaultCrateOverrides // {
    libc = attrs: { buildInputs = []; };
  };
}
```

15.25.3.3. Options and phases configuration

Actually, the overrides introduced in the previous section are more general. A number of other parameters

can be overridden:

- The version of rustc used to compile the crate:

```
(hello {}).override { rust = pkgs.rust; };
```

- Whether to build in release mode or debug mode (release mode by default):

```
(hello {}).override { release = false; };
```

- Whether to print the commands sent to rustc when building (equivalent to `--verbose` in cargo:

```
(hello {}).override { verbose = false; };
```

- Extra arguments to be passed to `rustc`:

```
(hello {}).override { extraRustcOpts = "-Z debuginfo=2"; };
```

- Phases, just like in any other derivation, can be specified using the following attributes: `preUnpack`, `postUnpack`, `prePatch`, `patches`, `postPatch`, `preConfigure` (in the case of a Rust crate, this is run before calling the "build" script), `postConfigure` (after the "build" script),`preBuild`, `postBuild`, `preInstall` and `postInstall`. As an example, here is how to create a new module before running the build script:

```
(hello {}).override {
  preConfigure = ''
     echo "pub const PATH=\"${hi.out}\";" >> src/path.rs"
  '';
};
```

15.25.3.4. Features

One can also supply features switches. For example, if we want to compile `diesel_cli` only with the `postgres` feature, and no default features, we would write:

```
(callPackage ./diesel.nix {}).diesel {
  default = false;
  postgres = true;
}
```

Where `diesel.nix` is the file generated by Carnix, as explained above.

15.25.4. Setting Up `nix-shell`

Oftentimes you want to develop code from within `nix-shell`. Unfortunately `buildRustCrate` does not support common `nix-shell` operations directly (see this issue) so we will use `stdenv.mkDerivation` instead.

Using the example `hello` project above, we want to do the following: - Have access to `cargo` and `rustc` - Have the `openssl` library available to a crate through it's *normal* compilation mechanism (`pkg-config`).

A typical `shell.nix` might look like:

```
with import <nixpkgs> {};

stdenv.mkDerivation {
  name = "rust-env";
  nativeBuildInputs = [
    rustc cargo

    # Example Build-time Additional Dependencies
    pkg-config
  ];
  buildInputs = [
    # Example Run-time Additional Dependencies
    openssl
  ];

  # Set Environment Variables
  RUST_BACKTRACE = 1;
}
```

You should now be able to run the following:

```
$ nix-shell --pure
$ cargo build
$ cargo test
```

15.25.4.1. Controlling Rust Version Inside `nix-shell`

To control your rust version (i.e. use nightly) from within `shell.nix` (or other nix expressions) you can use the following `shell.nix`

```
# Latest Nightly
with import <nixpkgs> {};
```

```
let src = fetchFromGitHub {
      owner = "mozilla";
      repo = "nixpkgs-mozilla";
      # commit from: 2019-05-15
      rev = "9f35c4b09fd44a77227e79ff0c1b4b6a69dff533";
      sha256 = "18h0nvh55b5an4gmlgfbvwbyqj91bklf1zymis6lbdh75571qaz0";
   };
in
with import "${src.out}/rust-overlay.nix" pkgs pkgs;
stdenv.mkDerivation {
  name = "rust-env";
  buildInputs = [
    # Note: to use stable, just replace `nightly` with `stable`
    latest.rustChannels.nightly.rust

    # Add some extra dependencies from `pkgs`
    pkg-config openssl
  ];

  # Set Environment Variables
  RUST_BACKTRACE = 1;
}
```

Now run:

```
$ rustc --version
rustc 1.26.0-nightly (188e693b3 2018-03-26)
```

To see that you are using nightly.

15.25.5. Using the Rust nightlies overlay

Mozilla provides an overlay for nixpkgs to bring a nightly version of Rust into scope. This overlay can *also* be used to install recent unstable or stable versions of Rust, if desired.

15.25.5.1. Rust overlay installation

You can use this overlay by either changing your local nixpkgs configuration, or by adding the overlay declaratively in a nix expression, e.g. in `configuration.nix`. For more information see #sec-overlays-install.

15.25.5.1.1. Imperative rust overlay installation

Clone nixpkgs-mozilla, and create a symbolic link to the file rust-overlay.nix in the

`~/.config/nixpkgs/overlays` directory.

```
$ git clone https://github.com/mozilla/nixpkgs-mozilla.git
$ mkdir -p ~/.config/nixpkgs/overlays
$ ln -s $(pwd)/nixpkgs-mozilla/rust-overlay.nix ~/.config/nixpkgs/overlays/
```

15.25.5.2. Declarative rust overlay installation

Add the following to your `configuration.nix`, `home-configuration.nix`, `shell.nix`, or similar:

```
{ pkgs ? import <nixpkgs> {
    overlays = [
      (import (builtins.fetchTarball https://github.com/mozilla/nixpkgs-moz
      # Further overlays go here
    ];
  };
};
```

Note that this will fetch the latest overlay version when rebuilding your system.

15.25.5.3. Rust overlay usage

The overlay contains attribute sets corresponding to different versions of the rust toolchain, such as:

- `latest.rustChannels.stable`
- `latest.rustChannels.nightly`
- a function `rustChannelOf`, called as `(rustChannelOf { date = "2018-04-11"; channel = "nightly"; })`, or...
- `(nixpkgs.rustChannelOf { rustToolchain = ./rust-toolchain; })` if you have a local `rust-toolchain` file (see https://github.com/mozilla/nixpkgs-mozilla#using-in-nix-expressions for an example)

Each of these contain packages such as `rust`, which contains your usual rust development tools with the respective toolchain chosen. For example, you might want to add `latest.rustChannels.stable.rust` to the list of packages in your configuration.

Imperatively, the latest stable version can be installed with the following command:

```
$ nix-env -Ai nixpkgs.latest.rustChannels.stable.rust
```

Or using the attribute with nix-shell:

```
$ nix-shell -p nixpkgs.latest.rustChannels.stable.rust
```

Substitute the `nixpkgs` prefix with `nixos` on NixOS. To install the beta or nightly channel, "stable" should be substituted by "nightly" or "beta", or use the function provided by this overlay to pull a version based on a build date.

The overlay automatically updates itself as it uses the same source as rustup.

15.26. TeX Live

Since release 15.09 there is a new TeX Live packaging that lives entirely under attribute `texlive`.

15.26.1. User's guide

- For basic usage just pull `texlive.combined.scheme-basic` for an environment with basic LaTeX support.
- It typically won't work to use separately installed packages together. Instead, you can build a custom set of packages like this:

```
texlive.combine {
  inherit (texlive) scheme-small collection-langkorean algorithms cm-supe
}
```

- There are all the schemes, collections and a few thousand packages, as defined upstream (perhaps with tiny differences).
- By default you only get executables and files needed during runtime, and a little documentation for the core packages. To change that, you need to add `pkgFilter` function to `combine`.

```
texlive.combine {
  # inherit (texlive) whatever-you-want;
  pkgFilter = pkg:
    pkg.tlType == "run" || pkg.tlType == "bin" || pkg.pname == "cm-super"
  # elem tlType [ "run" "bin" "doc" "source" ]
  # there are also other attributes: version, name
}
```

- You can list packages e.g. by `nix repl`.

```
$ nix repl
nix-repl> :l <nixpkgs>
nix-repl> texlive.collection-[TAB]
```

- Note that the wrapper assumes that the result has a chance to be useful. For example, the core executables should be present, as well as some core data files. The supported way of ensuring this is by including some scheme, for example `scheme-basic`, into the combination.

15.26.2. Custom packages

You may find that you need to use an external TeX package. A derivation for such package has to provide contents of the "texmf" directory in its output and provide the `tlType` attribute. Here is a (very verbose) example:

```
with import <nixpkgs> {};

let
  foiltex_run = stdenvNoCC.mkDerivation {
    pname = "latex-foiltex";
    version = "2.1.4b";
    passthru.tlType = "run";

    srcs = [
      (fetchurl {
        url = "http://mirrors.ctan.org/macros/latex/contrib/foiltex/foiltex
        sha256 = "07frz0krpz7kkcwlayrwrj2a2pixmv0icbngyw92srp9fp23cqpz";
      })
      (fetchurl {
        url = "http://mirrors.ctan.org/macros/latex/contrib/foiltex/foiltex
        sha256 = "09wkyidxk3n3zvqxfs61wlypmbhi1pxmjdi1kns9n2ky8ykbff99";
      })
    ];

    unpackPhase = ''
      runHook preUnpack

      for _src in $srcs; do
        cp "$_src" $(stripHash "$_src")
      done

      runHook postUnpack
    '';

    nativeBuildInputs = [ texlive.combined.scheme-small ];

    dontConfigure = true;
```

```
    buildPhase = ''
      runHook preBuild

      # Generate the style files
      latex foiltex.ins

      runHook postBuild
    '';

    installPhase = ''
      runHook preInstall

      path="$out/tex/latex/foiltex"
      mkdir -p "$path"
      cp *.{cls,def,clo} "$path/"

      runHook postInstall
    '';

    meta = with lib; {
      description = "A LaTeX2e class for overhead transparencies";
      license = licenses.unfreeRedistributable;
      maintainers = with maintainers; [ veprbl ];
      platforms = platforms.all;
    };
  };
  foiltex = { pkgs = [ foiltex_run ]; };

  latex_with_foiltex = texlive.combine {
    inherit (texlive) scheme-small;
    inherit foiltex;
  };
in
  runCommand "test.pdf" {
    nativeBuildInputs = [ latex_with_foiltex ];
  } ''
cat >test.tex <<EOF
\documentclass{foils}

\title{Presentation title}
\date{}
```

```
\begin{document}
\maketitle
\end{document}
EOF
  pdflatex test.tex
  cp test.pdf $out
''
```

15.27. Titanium

The Nixpkgs repository contains facilities to deploy a variety of versions of the Titanium SDK versions, a cross-platform mobile app development framework using JavaScript as an implementation language, and includes a function abstraction making it possible to build Titanium applications for Android and iOS devices from source code.

Not all Titanium features supported – currently, it can only be used to build Android and iOS apps.

15.27.1. Building a Titanium app

We can build a Titanium app from source for Android or iOS and for debugging or release purposes by invoking the `titaniumenv.buildApp {}` function:

```
titaniumenv.buildApp {
  name = "myapp";
  src = ./myappsource;

  preBuild = "";
  target = "android"; # or 'iphone'
  tiVersion = "7.1.0.GA";
  release = true;

  androidsdkArgs = {
    platformVersions = [ "25" "26" ];
  };
  androidKeyStore = ./keystore;
  androidKeyAlias = "myfirstapp";
  androidKeyStorePassword = "secret";

  xcodeBaseDir = "/Applications/Xcode.app";
  xcodewrapperArgs = {
    version = "9.3";
  };
  iosMobileProvisioningProfile = ./myprovisioning.profile;
```

```
  iosCertificateName = "My Company";
  iosCertificate = ./mycertificate.p12;
  iosCertificatePassword = "secret";
  iosVersion = "11.3";
  iosBuildStore = false;

  enableWirelessDistribution = true;
  installURL = "/installipa.php";
}
```

The `titaniumenv.buildApp {}` function takes the following parameters:

- The `name` parameter refers to the name in the Nix store.
- The `src` parameter refers to the source code location of the app that needs to be built.
- `preRebuild` contains optional build instructions that are carried out before the build starts.
- `target` indicates for which device the app must be built. Currently only "android" and "iphone" (for iOS) are supported.
- `tiVersion` can be used to optionally override the requested Titanium version in `tiapp.xml`. If not specified, it will use the version in `tiapp.xml`.
- `release` should be set to true when building an app for submission to the Google Playstore or Apple Appstore. Otherwise, it should be false.

When the `target` has been set to `android`, we can configure the following parameters:

- The `androidSdkArgs` parameter refers to an attribute set that propagates all parameters to the `androidenv.composeAndroidPackages {}` function. This can be used to install all relevant Android plugins that may be needed to perform the Android build. If no parameters are given, it will deploy the platform SDKs for API-levels 25 and 26 by default.

When the `release` parameter has been set to true, you need to provide parameters to sign the app:

- `androidKeyStore` is the path to the keystore file
- `androidKeyAlias` is the key alias
- `androidKeyStorePassword` refers to the password to open the keystore file.

When the `target` has been set to `iphone`, we can configure the following parameters:

- The `xcodeBaseDir` parameter refers to the location where Xcode has been installed. When none value is given, the above value is the default.
- The `xcodewrapperArgs` parameter passes arbitrary parameters to the `xcodeenv.composeXcodeWrapper {}` function. This can, for example, be used to adjust the default version of Xcode.

When `release` has been set to true, you also need to provide the following parameters:

- `iosMobileProvisioningProfile` refers to a mobile provisioning profile needed for signing.
- `iosCertificateName` refers to the company name in the P12 certificate.
- `iosCertificate` refers to the path to the P12 file.
- `iosCertificatePassword` contains the password to open the P12 file.
- `iosVersion` refers to the iOS SDK version to use. It defaults to the latest version.
- `iosBuildStore` should be set to `true` when building for the Apple Appstore submission. For enterprise or ad-hoc builds it should be set to `false`.

When `enableWirelessDistribution` has been enabled, you must also provide the path of the PHP script (`installURL`) (that is included with the iOS build environment) to enable wireless ad-hoc installations.

15.27.2. Emulating or simulating the app

It is also possible to simulate the correspond iOS simulator build by using `xcodeenv.simulateApp {}` and emulate an Android APK by using `androidenv.emulateApp {}`.

15.28. Vim

Both Neovim and Vim can be configured to include your favorite plugins and additional libraries.

Loading can be deferred; see examples.

At the moment we support three different methods for managing plugins:

- Vim packages (*recommend*)
- VAM (=vim-addon-manager)
- Pathogen
- vim-plug

15.28.1. Custom configuration

Adding custom .vimrc lines can be done using the following code:

```
vim_configurable.customize {
  # `name` specifies the name of the executable and package
  name = "vim-with-plugins";

  vimrcConfig.customRC = ''
```

```
    set hidden
  '';
}
```

This configuration is used when Vim is invoked with the command specified as name, in this case `vim-with-plugins`.

For Neovim the `configure` argument can be overridden to achieve the same:

```
neovim.override {
  configure = {
    customRC = ''
      # here your custom configuration goes!
    '';
  };
}
```

If you want to use `neovim-qt` as a graphical editor, you can configure it by overriding Neovim in an overlay or passing it an overridden Neovimn:

```
neovim-qt.override {
  neovim = neovim.override {
    configure = {
      customRC = ''
        # your custom configuration
      '';
    };
  };
}
```

15.28.2. Managing plugins with Vim packages

To store you plugins in Vim packages (the native Vim plugin manager, see `:help packages`) the following example can be used:

```
vim_configurable.customize {
  vimrcConfig.packages.myVimPackage = with pkgs.vimPlugins; {
    # loaded on launch
    start = [ youcompleteme fugitive ];
    # manually loadable by calling `:packadd $plugin-name`
    # however, if a Vim plugin has a dependency that is not explicitly list
    # opt that dependency will always be added to start to avoid confusion.
    opt = [ phpCompletion elm-vim ];
```

```
    # To automatically load a plugin when opening a filetype, add vimrc lin
    # autocmd FileType php :packadd phpCompletion
  };
}
```

`myVimPackage` is an arbitrary name for the generated package. You can choose any name you like. For Neovim the syntax is:

```
neovim.override {
  configure = {
    customRC = ''
      # here your custom configuration goes!
    '';
    packages.myVimPackage = with pkgs.vimPlugins; {
      # see examples below how to use custom packages
      start = [ ];
      # If a Vim plugin has a dependency that is not explicitly listed in
      # opt that dependency will always be added to start to avoid confusio
      opt = [ ];
    };
  };
}
```

The resulting package can be added to `packageOverrides` in `~/.nixpkgs/config.nix` to make it installable:

```
{
  packageOverrides = pkgs: with pkgs; {
    myVim = vim_configurable.customize {
      # `name` specifies the name of the executable and package
      name = "vim-with-plugins";
      # add here code from the example section
    };
    myNeovim = neovim.override {
      configure = {
      # add here code from the example section
      };
    };
  };
}
```

After that you can install your special grafted `myVim` or `myNeovim` packages.

15.28.2.1. What if your favourite Vim plugin isn't already packaged?

If one of your favourite plugins isn't packaged, you can package it yourself:

```
{ config, pkgs, ... }:

let
  easygrep = pkgs.vimUtils.buildVimPlugin {
    name = "vim-easygrep";
    src = pkgs.fetchFromGitHub {
      owner = "dkprice";
      repo = "vim-easygrep";
      rev = "d0c36a77cc63c22648e792796b1815b44164653a";
      sha256 = "0y2p5mz0d5fhg6n68lhfhl8p4mlwkb82q337c22djs4w5zyzggbc";
    };
  };
in
{
  environment.systemPackages = [
    (
      pkgs.neovim.override {
        configure = {
          packages.myPlugins = with pkgs.vimPlugins; {
          start = [
            vim-go # already packaged plugin
            easygrep # custom package
          ];
          opt = [];
        };
        # ...
      };
     }
    )
  ];
}
```

15.28.3. Managing plugins with vim-plug

To use vim-plug to manage your Vim plugins the following example can be used:

```
vim_configurable.customize {
  vimrcConfig.packages.myVimPackage = with pkgs.vimPlugins; {
```

```
    # loaded on launch
    plug.plugins = [ youcompleteme fugitive phpCompletion elm-vim ];
  };
}
```

For Neovim the syntax is:

```
neovim.override {
  configure = {
    customRC = ''
      # here your custom configuration goes!
    '';
    plug.plugins = with pkgs.vimPlugins; [
      vim-go
    ];
  };
}
```

15.28.4. Managing plugins with VAM

15.28.4.1. Handling dependencies of Vim plugins

VAM introduced .json files supporting dependencies without versioning assuming that “using latest version” is ok most of the time.

15.28.4.2. Example

First create a vim-scripts file having one plugin name per line. Example:

```
"tlib"
{'name': 'vim-addon-sql'}
{'filetype_regex': '\%(vim)$', 'names': ['reload', 'vim-dev-plugin']}
```

Such vim-scripts file can be read by VAM as well like this:

```
call vam#Scripts(expand('~/.vim-scripts'), {})
```

Create a default.nix file:

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "ghc7102" }:
nixpkgs.vim_configurable.customize { name = "vim"; vimrcConfig.vam.pluginDi
```

Create a generate.vim file:

```
ActivateAddons vim-addon-vim2nix
let vim_scripts = "vim-scripts"
call nix#ExportPluginsForNix({
\  'path_to_nixpkgs': eval('{"'.substitute(substitute(substitute($NIX_PATH,
\  'cache_file': '/tmp/vim2nix-cache',
\  'try_catch': 0,
\  'plugin_dictionaries': ["vim-addon-manager"]+map(readfile(vim_scripts),
\ })
```

Then run

```
nix-shell -p vimUtils.vim_with_vim2nix --command "vim -c 'source generate.v
```

You should get a Vim buffer with the nix derivations (output1) and vam.pluginDictionaries (output2). You can add your Vim to your system's configuration file like this and start it by "vim-my":

```
my-vim =
  let plugins = let inherit (vimUtils) buildVimPluginFrom2Nix; in {
    copy paste output1 here
  }; in vim_configurable.customize {
    name = "vim-my";

    vimrcConfig.vam.knownPlugins = plugins; # optional
    vimrcConfig.vam.pluginDictionaries = [
       copy paste output2 here
    ];

    # Pathogen would be
    # vimrcConfig.pathogen.knownPlugins = plugins; # plugins
    # vimrcConfig.pathogen.pluginNames = ["tlib"];
  };
```

Sample output1:

```
"reload" = buildVimPluginFrom2Nix { # created by nix#NixDerivation
  name = "reload";
  src = fetchgit {
    url = "git://github.com/xolox/vim-reload";
    rev = "0a601a668727f5b675cb1ddc19f6861f3f7ab9e1";
```

```
    sha256 = "0vb832l9yxj919f5hfg6qj6bn9ni57gnjd3bj7zpq7d4iv2s4wdh";
  };
  dependencies = ["nim-misc"];

};
[...]
```

Sample output2:

```
[
  ''vim-addon-manager''
  ''tlib''
  { "name" = ''vim-addon-sql''; }
  { "filetype_regex" = ''\%(vim)$$''; "names" = [ ''reload'' ''vim-dev-plug
]
```

15.28.5. Adding new plugins to nixpkgs

Nix expressions for Vim plugins are stored in pkgs/misc/vim-plugins. For the vast majority of plugins, Nix expressions are automatically generated by running `./update.py`. This creates a generated.nix file based on the plugins listed in vim-plugin-names. Plugins are listed in alphabetical order in `vim-plugin-names` using the format `[github username]/[repository]`. For example https://github.com/scrooloose/nerdtree becomes `scrooloose/nerdtree`.

Some plugins require overrides in order to function properly. Overrides are placed in overrides.nix. Overrides are most often required when a plugin requires some dependencies, or extra steps are required during the build process. For example `deoplete-fish` requires both `deoplete-nvim` and `vim-fish`, and so the following override was added:

```
deoplete-fish = super.deoplete-fish.overrideAttrs(old: {
  dependencies = with super; [ deoplete-nvim vim-fish ];
});
```

Sometimes plugins require an override that must be changed when the plugin is updated. This can cause issues when Vim plugins are auto-updated but the associated override isn't updated. For these plugins, the override should be written so that it specifies all information required to install the plugin, and running `./update.py` doesn't change the derivation for the plugin. Manually updating the override is required to update these types of plugins. An example of such a plugin is `LanguageClient-neovim`.

To add a new plugin, run `./update.py --add "[owner]/[name]"`. **NOTE**: This script automatically commits to your git repository. Be sure to check out a fresh branch before running.

Finally, there are some plugins that are also packaged in nodePackages because they have Javascript-

related build steps, such as running webpack. Those plugins are not listed in `vim-plugin-names` or managed by `update.py` at all, and are included separately in `overrides.nix`. Currently, all these plugins are related to the `coc.nvim` ecosystem of Language Server Protocol integration with vim/neovim.

15.28.6. Updating plugins in nixpkgs

Run the update script with a GitHub API token that has at least `public_repo` access. Running the script without the token is likely to result in rate-limiting (429 errors). For steps on creating an API token, please refer to GitHub's token documentation.

```
GITHUB_API_TOKEN=my_token ./pkgs/misc/vim-plugins/update.py
```

Alternatively, set the number of processes to a lower count to avoid rate-limiting.

```
./pkgs/misc/vim-plugins/update.py --proc 1
```

15.28.7. Important repositories

- vim-pi is a plugin repository from VAM plugin manager meant to be used by others as well used by
- vim2nix which generates the .nix code

Chapter 16. Packages

Table of Contents

16.1. Citrix Workspace

16.2. DLib

16.3. Eclipse

16.4. Elm

16.5. Emacs

16.6. Firefox

16.7. Fish

16.8. FUSE

16.9. ibus-engines.typing-booster

16.10. Kakoune

16.11. Linux kernel

16.12. Locales

16.13. Nginx

16.14. OpenGL

16.15. Interactive shell helpers

16.16. Steam

16.17. Cataclysm: Dark Days Ahead

16.18. Urxvt

16.19. Weechat

16.20. X.org

This chapter contains information about how to use and maintain the Nix expressions for a number of specific packages, such as the Linux kernel or X.org.

16.1. Citrix Workspace

The Citrix Workspace App is a remote desktop viewer which provides access to XenDesktop installations.

16.1.1. Basic usage

The tarball archive needs to be downloaded manually as the license agreements of the vendor for Citrix Workspace needs to be accepted first. Then run `nix-prefetch-url file://$PWD/linuxx64-$version.tar.gz`. With the archive available in the store the package can be built and installed with Nix.

16.1.2. Citrix Selfservice

The selfservice is an application managing Citrix desktops and applications. Please note that this feature only works with at least citrix_workspace_20_06_0 and later versions.

In order to set this up, you first have to download the `.cr` file from the Netscaler Gateway. After that you can configure the `selfservice` like this:

```
$ storebrowse -C ~/Downloads/receiverconfig.cr
$ selfservice
```

16.1.3. Custom certificates

The `Citrix Workspace App` in `nixpkgs` trusts several certificates from the Mozilla database by

default. However several companies using Citrix might require their own corporate certificate. On distros with imperative packaging these certs can be stored easily in `$ICAROOT`, however this directory is a store path in `nixpkgs`. In order to work around this issue the package provides a simple mechanism to add custom certificates without rebuilding the entire package using `symlinkJoin`:

```
with import <nixpkgs> { config.allowUnfree = true; };
let
  extraCerts = [
    ./custom-cert-1.pem
    ./custom-cert-2.pem # ...
  ];
in citrix_workspace.override { inherit extraCerts; }
```

16.2. DLib

DLib is a modern, C++-based toolkit which provides several machine learning algorithms.

16.2.1. Compiling without AVX support

Especially older CPUs don't support AVX (Advanced Vector Extensions) instructions that are used by DLib to optimize their algorithms.

On the affected hardware errors like `Illegal instruction` will occur. In those cases AVX support needs to be disabled:

```
self: super: { dlib = super.dlib.override { avxSupport = false; }; }
```

16.3. Eclipse

The Nix expressions related to the Eclipse platform and IDE are in `pkgs/applications/editors/eclipse`.

Nixpkgs provides a number of packages that will install Eclipse in its various forms. These range from the bare-bones Eclipse Platform to the more fully featured Eclipse SDK or Scala-IDE packages and multiple version are often available. It is possible to list available Eclipse packages by issuing the command:

```
$ nix-env -f '<nixpkgs>' -qaP -A eclipses --description
```

Once an Eclipse variant is installed it can be run using the `eclipse` command, as expected. From within Eclipse it is then possible to install plugins in the usual manner by either manually specifying an Eclipse update site or by installing the Marketplace Client plugin and using it to discover and install other plugins. This installation method provides an Eclipse installation that closely resemble a manually installed Eclipse.

If you prefer to install plugins in a more declarative manner then Nixpkgs also offer a number of Eclipse plugins that can be installed in an *Eclipse environment*. This type of environment is created using the function `eclipseWithPlugins` found inside the `nixpkgs.eclipses` attribute set. This function takes as argument `{ eclipse, plugins ? [], jvmArgs ? [] }` where `eclipse` is a one of the Eclipse packages described above, `plugins` is a list of plugin derivations, and `jvmArgs` is a list of arguments given to the JVM running the Eclipse. For example, say you wish to install the latest Eclipse Platform with the popular Eclipse Color Theme plugin and also allow Eclipse to use more RAM. You could then add

```
packageOverrides = pkgs: {
  myEclipse = with pkgs.eclipses; eclipseWithPlugins {
    eclipse = eclipse-platform;
    jvmArgs = [ "-Xmx2048m" ];
    plugins = [ plugins.color-theme ];
  };
}
```

to your Nixpkgs configuration (`~/.config/nixpkgs/config.nix`) and install it by running `nix-env -f '<nixpkgs>' -iA myEclipse` and afterward run Eclipse as usual. It is possible to find out which plugins are available for installation using `eclipseWithPlugins` by running

```
$ nix-env -f '<nixpkgs>' -qaP -A eclipses.plugins --description
```

If there is a need to install plugins that are not available in Nixpkgs then it may be possible to define these plugins outside Nixpkgs using the `buildEclipseUpdateSite` and `buildEclipsePlugin` functions found in the `nixpkgs.eclipses.plugins` attribute set. Use the `buildEclipseUpdateSite` function to install a plugin distributed as an Eclipse update site. This function takes `{ name, src }` as argument where `src` indicates the Eclipse update site archive. All Eclipse features and plugins within the downloaded update site will be installed. When an update site archive is not available then the `buildEclipsePlugin` function can be used to install a plugin that consists of a pair of feature and plugin JARs. This function takes an argument `{ name, srcFeature, srcPlugin }` where `srcFeature` and `srcPlugin` are the feature and plugin JARs, respectively.

Expanding the previous example with two plugins using the above functions we have

```
packageOverrides = pkgs: {
  myEclipse = with pkgs.eclipses; eclipseWithPlugins {
    eclipse = eclipse-platform;
    jvmArgs = [ "-Xmx2048m" ];
    plugins = [
      plugins.color-theme
      (plugins.buildEclipsePlugin {
        name = "myplugin1-1.0";
```

```
        srcFeature = fetchurl {
          url = "http://…/features/myplugin1.jar";
          sha256 = "123…";
        };
        srcPlugin = fetchurl {
          url = "http://…/plugins/myplugin1.jar";
          sha256 = "123…";
        };
      });
      (plugins.buildEclipseUpdateSite {
        name = "myplugin2-1.0";
        src = fetchurl {
          stripRoot = false;
          url = "http://…/myplugin2.zip";
          sha256 = "123…";
        };
      });
    ];
  };
}
```

16.4. Elm

To start a development environment do

```
nix-shell -p elmPackages.elm elmPackages.elm-format
```

To update the Elm compiler, see `nixpkgs/pkgs/development/compilers/elm/README.md`.

To package Elm applications, read about elm2nix.

16.5. Emacs

16.5.1. Configuring Emacs

The Emacs package comes with some extra helpers to make it easier to configure. `emacs.pkgs.withPackages` allows you to manage packages from ELPA. This means that you will not have to install that packages from within Emacs. For instance, if you wanted to use `company` `counsel`, `flycheck`, `ivy`, `magit`, `projectile`, and `use-package` you could use this as a `~/.config/nixpkgs/config.nix` override:

```
{
  packageOverrides = pkgs: with pkgs; {
```

```
    myEmacs = emacs.pkgs.withPackages (epkgs: (with epkgs.melpaStablePackag
      company
      counsel
      flycheck
      ivy
      magit
      projectile
      use-package
    ]));
  }
}
```

You can install it like any other packages via `nix-env -iA myEmacs`. However, this will only install those packages. It will not `configure` them for us. To do this, we need to provide a configuration file. Luckily, it is possible to do this from within Nix! By modifying the above example, we can make Emacs load a custom config file. The key is to create a package that provide a `default.el` file in `/share/emacs/site-start/`. Emacs knows to load this file automatically when it starts.

```
{
  packageOverrides = pkgs: with pkgs; rec {
    myEmacsConfig = writeText "default.el" ''
      ;; initialize package

      (require 'package)
      (package-initialize 'noactivate)
      (eval-when-compile
        (require 'use-package))

      ;; load some packages

      (use-package company
        :bind ("<C-tab>" . company-complete)
        :diminish company-mode
        :commands (company-mode global-company-mode)
        :defer 1
        :config
        (global-company-mode))

      (use-package counsel
        :commands (counsel-descbinds)
        :bind (([remap execute-extended-command] . counsel-M-x)
               ("C-x C-f" . counsel-find-file)
               ("C-c g" . counsel-git)
```

```
            ("C-c j" . counsel-git-grep)
            ("C-c k" . counsel-ag)
            ("C-x l" . counsel-locate)
            ("M-y" . counsel-yank-pop)))

  (use-package flycheck
    :defer 2
    :config (global-flycheck-mode))

  (use-package ivy
    :defer 1
    :bind (("C-c C-r" . ivy-resume)
           ("C-x C-b" . ivy-switch-buffer)
           :map ivy-minibuffer-map
           ("C-j" . ivy-call))
    :diminish ivy-mode
    :commands ivy-mode
    :config
    (ivy-mode 1))

  (use-package magit
    :defer
    :if (executable-find "git")
    :bind (("C-x g" . magit-status)
           ("C-x G" . magit-dispatch-popup))
    :init
    (setq magit-completing-read-function 'ivy-completing-read))

  (use-package projectile
    :commands projectile-mode
    :bind-keymap ("C-c p" . projectile-command-map)
    :defer 5
    :config
    (projectile-global-mode))
'';

myEmacs = emacs.pkgs.withPackages (epkgs: (with epkgs.melpaStablePackag
  (runCommand "default.el" {} ''
     mkdir -p $out/share/emacs/site-lisp
     cp ${myEmacsConfig} $out/share/emacs/site-lisp/default.el
   '')
  company
  counsel
```

```
      flycheck
      ivy
      magit
      projectile
      use-package
    ]));
  };
}
```

This provides a fairly full Emacs start file. It will load in addition to the user's presonal config. You can always disable it by passing `-q` to the Emacs command.

Sometimes `emacs.pkgs.withPackages` is not enough, as this package set has some priorities imposed on packages (with the lowest priority assigned to Melpa Unstable, and the highest for packages manually defined in `pkgs/top-level/emacs-packages.nix`). But you can't control this priorities when some package is installed as a dependency. You can override it on per-package-basis, providing all the required dependencies manually - but it's tedious and there is always a possibility that an unwanted dependency will sneak in through some other package. To completely override such a package you can use `overrideScope'`.

```
overrides = self: super: rec {
  haskell-mode = self.melpaPackages.haskell-mode;
  ...
};
((emacsPackagesFor emacs).overrideScope' overrides).emacs.pkgs.withPackages
  (p: with p; [
    # here both these package will use haskell-mode of our own choice
    ghc-mod
    dante
  ])
```

16.6. Firefox

16.6.1. Build wrapped Firefox with extensions and policies

The `wrapFirefox` function allows to pass policies, preferences and extension that are available to firefox. With the help of `fetchFirefoxAddon` this allows build a firefox version that already comes with addons pre-installed:

```
{
  myFirefox = wrapFirefox firefox-unwrapped {
    nixExtensions = [
```

```
      (fetchFirefoxAddon {
        name = "ublock"; # Has to be unique!
        url = "https://addons.mozilla.org/firefox/downloads/file/3679754/ub
        sha256 = "1h768ljlh3pi23l27qp961v1hd0nbj2vasgy11bmcrlqp40zgvnr";
      })
    ];

    extraPolicies = {
      CaptivePortal = false;
      DisableFirefoxStudies = true;
      DisablePocket = true;
      DisableTelemetry = true;
      DisableFirefoxAccounts = true;
      FirefoxHome = {
        Pocket = false;
        Snippets = false;
      };
       UserMessaging = {
         ExtensionRecommendations = false;
         SkipOnboarding = true;
       };
    };

    extraPrefs = ''
      // Show more ssl cert infos
      lockPref("security.identityblock.show_extended_validation", true);
    '';
  };
}
```

If `nixExtensions != null` then all manually installed addons will be uninstalled from your browser profile. To view available enterprise policies visit enterprise policies or type into the Firefox url bar: `about:policies#documentation`. Nix installed addons do not have a valid signature, which is why signature verification is disabled. This does not compromise security because downloaded addons are checksumed and manual addons can't be installed. Also make sure that the `name` field of fetchFirefoxAddon is unique. If you remove an addon from the nixExtensions array, rebuild and start Firefox the removed addon will be completly removed with all of its settings.

16.6.2. Troubleshooting

If addons do not appear installed although they have been defined in your nix configuration file reset the local addon state of your Firefox profile by clicking `help -> restart with addons disabled -> restart -> refresh firefox`. This can

happen if you switch from manual addon mode to nix addon mode and then back to manual mode and then again to nix addon mode.

16.7. Fish

Fish is a "smart and user-friendly command line shell" with support for plugins.

16.7.1. Vendor Fish scripts

Any package may ship its own Fish completions, configuration snippets, and functions. Those should be installed to `$out/share/fish/vendor_{completions,conf,functions}.d` respectively.

When the `programs.fish.enable` and `programs.fish.vendor.{completions,config,functions}.enable` options from the NixOS Fish module are set to true, those paths are symlinked in the current system environment and automatically loaded by Fish.

16.7.2. Packaging Fish plugins

While packages providing standalone executables belong to the top level, packages which have the sole purpose of extending Fish belong to the `fishPlugins` scope and should be registered in `pkgs/shells/fish/plugins/default.nix`.

The `buildFishPlugin` utility function can be used to automatically copy Fish scripts from `$src/{completions,conf,conf.d,functions}` to the standard vendor installation paths. It also sets up the test environment so that the optional `checkPhase` is executed in a Fish shell with other already packaged plugins and package-local Fish functions specified in `checkPlugins` and `checkFunctionDirs` respectively.

See `pkgs/shells/fish/plugins/pure.nix` for an example of Fish plugin package using `buildFishPlugin` and running unit tests with the `fishtape` test runner.

16.7.3. Fish wrapper

The `wrapFish` package is a wrapper around Fish which can be used to create Fish shells initialised with some plugins as well as completions, configuration snippets and functions sourced from the given paths. This provides a convenient way to test Fish plugins and scripts without having to alter the environment.

```
wrapFish {
  pluginPkgs = with fishPlugins; [ pure foreign-env ];
  completionDirs = [];
  functionDirs = [];
  confDirs = [ "/path/to/some/fish/init/dir/" ];
}
```

16.8. FUSE

Some packages rely on FUSE to provide support for additional filesystems not supported by the kernel.

In general, FUSE software are primarily developed for Linux but many of them can also run on macOS. Nixpkgs supports FUSE packages on macOS, but it requires macFUSE to be installed outside of Nix. macFUSE currently isn't packaged in Nixpkgs mainly because it includes a kernel extension, which isn't supported by Nix outside of NixOS.

If a package fails to run on macOS with an error message similar to the following, it's a likely sign that you need to have macFUSE installed.

```
dyld: Library not loaded: /usr/local/lib/libfuse.2.dylib
Referenced from: /nix/store/w8bi72bssv0bnxhwfw3xr1mvn7myf37x-sshfs-fuse-2.1
Reason: image not found
[1]    92299 abort      /nix/store/w8bi72bssv0bnxhwfw3xr1mvn7myf37x-sshfs-f
```

Package maintainers may often encounter the following error when building FUSE packages on macOS:

```
checking for fuse.h... no
configure: error: No fuse.h found.
```

This happens on autoconf based projects that uses `AC_CHECK_HEADERS` or `AC_CHECK_LIBS` to detect libfuse, and will occur even when the `fuse` package is included in `buildInputs`. It happens because libfuse headers throw an error on macOS if the `FUSE_USE_VERSION` macro is undefined. Many proejcts do define `FUSE_USE_VERSION`, but only inside C source files. This results in the above error at configure time because the configure script would attempt to compile sample FUSE programs without defining `FUSE_USE_VERSION`.

There are two possible solutions for this problem in Nixpkgs:

1. Pass `FUSE_USE_VERSION` to the configure script by adding `CFLAGS=-DFUSE_USE_VERSION=25` in `configureFlags`. The actual value would have to match the definition used in the upstream source code.

2. Remove `AC_CHECK_HEADERS` / `AC_CHECK_LIBS` for libfuse.

However, a better solution might be to fix the build script upstream to use `PKG_CHECK_MODULES` instead. This approach wouldn't suffer from the problem that `AC_CHECK_HEADERS`/`AC_CHECK_LIBS` has at the price of introducing a dependency on pkg-config.

16.9. ibus-engines.typing-booster

This package is an ibus-based completion method to speed up typing.

16.9.1. Activating the engine

IBus needs to be configured accordingly to activate `typing-booster`. The configuration depends on the desktop manager in use. For detailed instructions, please refer to the upstream docs.

On NixOS you need to explicitly enable `ibus` with given engines before customizing your desktop to use `typing-booster`. This can be achieved using the `ibus` module:

```
{ pkgs, ... }: {
  i18n.inputMethod = {
    enabled = "ibus";
    ibus.engines = with pkgs.ibus-engines; [ typing-booster ];
  };
}
```

16.9.2. Using custom hunspell dictionaries

The IBus engine is based on `hunspell` to support completion in many languages. By default the dictionaries `de-de`, `en-us`, `fr-moderne` `es-es`, `it-it`, `sv-se` and `sv-fi` are in use. To add another dictionary, the package can be overridden like this:

```
ibus-engines.typing-booster.override { langs = [ "de-at" "en-gb" ]; }
```

Note: each language passed to `langs` must be an attribute name in `pkgs.hunspellDicts`.

16.9.3. Built-in emoji picker

The `ibus-engines.typing-booster` package contains a program named `emoji-picker`. To display all emojis correctly, a special font such as `noto-fonts-emoji` is needed:

On NixOS it can be installed using the following expression:

```
{ pkgs, ... }: { fonts.fonts = with pkgs; [ noto-fonts-emoji ]; }
```

16.10. Kakoune

Kakoune can be built to autoload plugins:

```
(kakoune.override {
  plugins = with pkgs.kakounePlugins; [ parinfer-rust ];
})
```

16.11. Linux kernel

The Nix expressions to build the Linux kernel are in `pkgs/os-specific/linux/kernel`.

The function that builds the kernel has an argument `kernelPatches` which should be a list of `{name, patch, extraConfig}` attribute sets, where `name` is the name of the patch (which is included in the kernel's `meta.description` attribute), `patch` is the patch itself (possibly compressed), and `extraConfig` (optional) is a string specifying extra options to be concatenated to the kernel configuration file (`.config`).

The kernel derivation exports an attribute `features` specifying whether optional functionality is or isn't enabled. This is used in NixOS to implement kernel-specific behaviour. For instance, if the kernel has the `iwlwifi` feature (i.e. has built-in support for Intel wireless chipsets), then NixOS doesn't have to build the external `iwlwifi` package:

```
modulesTree = [kernel]
  ++ pkgs.lib.optional (!kernel.features ? iwlwifi) kernelPackages.iwlwifi
  ++ ...;
```

How to add a new (major) version of the Linux kernel to Nixpkgs:

1. Copy the old Nix expression (e.g. `linux-2.6.21.nix`) to the new one (e.g. `linux-2.6.22.nix`) and update it.
2. Add the new kernel to `all-packages.nix` (e.g., create an attribute `kernel_2_6_22`).
3. Now we're going to update the kernel configuration. First unpack the kernel. Then for each supported platform (`i686`, `x86_64`, `uml`) do the following:
    1. Make an copy from the old config (e.g. `config-2.6.21-i686-smp`) to the new one (e.g. `config-2.6.22-i686-smp`).
    2. Copy the config file for this platform (e.g. `config-2.6.22-i686-smp`) to `.config` in the kernel source tree.
    3. Run `make oldconfig ARCH={i386,x86_64,um}` and answer all questions. (For the uml configuration, also add `SHELL=bash`.) Make sure to keep the configuration consistent between platforms (i.e. don't enable some feature on `i686` and disable it on `x86_64`).
    4. If needed you can also run `make menuconfig`:

        ```
        $ nix-env -i ncurses
        $ export NIX_CFLAGS_LINK=-lncurses
        $ make menuconfig ARCH=arch
        ```

    5. Copy `.config` over the new config file (e.g. `config-2.6.22-i686-smp`).
4. Test building the kernel: `nix-build -A kernel_2_6_22`. If it compiles, ship it! For extra credit,

try booting NixOS with it.

5. It may be that the new kernel requires updating the external kernel modules and kernel-dependent packages listed in the `linuxPackagesFor` function in `all-packages.nix` (such as the NVIDIA drivers, AUFS, etc.). If the updated packages aren't backwards compatible with older kernels, you may need to keep the older versions around.

16.12. Locales

To allow simultaneous use of packages linked against different versions of `glibc` with different locale archive formats Nixpkgs patches `glibc` to rely on `LOCALE_ARCHIVE` environment variable.

On non-NixOS distributions this variable is obviously not set. This can cause regressions in language support or even crashes in some Nixpkgs-provided programs. The simplest way to mitigate this problem is exporting the `LOCALE_ARCHIVE` variable pointing to `${glibcLocales}/lib/locale/locale-archive`. The drawback (and the reason this is not the default) is the relatively large (a hundred MiB) size of the full set of locales. It is possible to build a custom set of locales by overriding parameters `allLocales` and `locales` of the package.

16.13. Nginx

Nginx is a reverse proxy and lightweight webserver.

16.13.1. ETags on static files served from the Nix store

HTTP has a couple different mechanisms for caching to prevent clients from having to download the same content repeatedly if a resource has not changed since the last time it was requested. When nginx is used as a server for static files, it implements the caching mechanism based on the `Last-Modified` response header automatically; unfortunately, it works by using filesystem timestamps to determine the value of the `Last-Modified` header. This doesn't give the desired behavior when the file is in the Nix store, because all file timestamps are set to 0 (for reasons related to build reproducibility).

Fortunately, HTTP supports an alternative (and more effective) caching mechanism: the `ETag` response header. The value of the `ETag` header specifies some identifier for the particular content that the server is sending (e.g. a hash). When a client makes a second request for the same resource, it sends that value back in an `If-None-Match` header. If the ETag value is unchanged, then the server does not need to resend the content.

As of NixOS 19.09, the nginx package in Nixpkgs is patched such that when nginx serves a file out of `/nix/store`, the hash in the store path is used as the `ETag` header in the HTTP response, thus providing proper caching functionality. This happens automatically; you do not need to do modify any configuration to get this behavior.

16.14. OpenGL

OpenGL support varies depending on which hardware is used and which drivers are available and loaded.

Broadly, we support both GL vendors: Mesa and NVIDIA.

16.14.1. NixOS Desktop

The NixOS desktop or other non-headless configurations are the primary target for OpenGL libraries and applications. The current solution for discovering which drivers are available is based on libglvnd. `libglvnd` performs "vendor-neutral dispatch", trying a variety of techniques to find the system's GL implementation. In practice, this will be either via standard GLX for X11 users or EGL for Wayland users, and supporting either NVIDIA or Mesa extensions.

16.14.2. Nix on GNU/Linux

If you are using a non-NixOS GNU/Linux/X11 desktop with free software video drivers, consider launching OpenGL-dependent programs from Nixpkgs with Nixpkgs versions of `libglvnd` and `mesa.drivers` in `LD_LIBRARY_PATH`. For Mesa drivers, the Linux kernel version doesn't have to match nixpkgs.

For proprietary video drivers you might have luck with also adding the corresponding video driver package.

16.15. Interactive shell helpers

Some packages provide the shell integration to be more useful. But unlike other systems, nix doesn't have a standard `share` directory location. This is why a bunch `PACKAGE-share` scripts are shipped that print the location of the corresponding shared folder. Current list of such packages is as following:

- `fzf` : `fzf-share`

E.g. `fzf` can then used in the `.bashrc` like this:

```
source "$(fzf-share)/completion.bash"
source "$(fzf-share)/key-bindings.bash"
```

16.16. Steam

16.16.1. Steam in Nix

Steam is distributed as a `.deb` file, for now only as an i686 package (the amd64 package only has documentation). When unpacked, it has a script called `steam` that in Ubuntu (their target distro) would go to `/usr/bin`. When run for the first time, this script copies some files to the user's home, which include another script that is the ultimate responsible for launching the steam binary, which is also in $HOME.

Nix problems and constraints:

- We don't have `/bin/bash` and many scripts point there. Similarly for `/usr/bin/python`.
- We don't have the dynamic loader in `/lib`.

- The `steam.sh` script in $HOME can not be patched, as it is checked and rewritten by steam.
- The steam binary cannot be patched, it's also checked.

The current approach to deploy Steam in NixOS is composing a FHS-compatible chroot environment, as documented here. This allows us to have binaries in the expected paths without disrupting the system, and to avoid patching them to work in a non FHS environment.

16.16.2. How to play

Use `programs.steam.enable = true;` if you want to add steam to systemPackages and also enable a few workarrounds aswell as Steam controller support or other Steam supported controllers such as the DualShock 4 or Nintendo Switch Pr.

16.16.3. Troubleshooting

- **Steam fails to start. What do I do?** Try to run

  ```
  strace steam
  ```

  to see what is causing steam to fail.

- **Using the FOSS Radeon or nouveau (nvidia) drivers**
  - The `newStdcpp` parameter was removed since NixOS 17.09 and should not be needed anymore.
  - Steam ships statically linked with a version of libcrypto that conflics with the one dynamically loaded by radeonsi_dri.so. If you get the error

    ```
    steam.sh: line 713: 7842 Segmentation fault (core dumped)
    ```

    have a look at this pull request.

- **Java**

  1. There is no java in steam chrootenv by default. If you get a message like

  ```
  /home/foo/.local/share/Steam/SteamApps/common/towns/towns.sh: line 1: java
  ```

  You need to add

  ```
  steam.override { withJava = true; };
  ```

16.16.4. steam-run

The FHS-compatible chroot used for steam can also be used to run other linux games that expect a FHS environment. To do it, add

```
pkgs.steam.override ({
          nativeOnly = true;
          newStdcpp = true;
        }).run
```

to your configuration, rebuild, and run the game with

```
steam-run ./foo
```

16.17. Cataclysm: Dark Days Ahead

16.17.1. How to install Cataclysm DDA

To install the latest stable release of Cataclysm DDA to your profile, execute `nix-env -f "<nixpkgs>" -iA cataclysm-dda`. For the curses build (build without tiles), install `cataclysmDDA.stable.curses`. Note: `cataclysm-dda` is an alias to `cataclysmDDA.stable.tiles`.

If you like access to a development build of your favorite git revision, override `cataclysm-dda-git` (or `cataclysmDDA.git.curses` if you like curses build):

```
cataclysm-dda-git.override {
  version = "YYYY-MM-DD";
  rev = "YOUR_FAVORITE_REVISION";
  sha256 = "CHECKSUM_OF_THE_REVISION";
}
```

The sha256 checksum can be obtained by

```
nix-prefetch-url --unpack "https://github.com/CleverRaven/Cataclysm-DDA/arc
```

The default configuration directory is `~/.cataclysm-dda`. If you prefer `$XDG_CONFIG_HOME/cataclysm-dda`, override the derivation:

```
cataclysm-dda.override {
  useXdgDir = true;
}
```

16.17.2. Important note for overriding packages

After applying `overrideAttrs`, you need to fix `passthru.pkgs` and `passthru.withMods` attributes either manually or by using `attachPkgs`:

```
let
  # You enabled parallel building.
  myCDDA = cataclysm-dda-git.overrideAttrs (_: {
    enableParallelBuilding = true;
  });

  # Unfortunately, this refers to the package before overriding and
  # parallel building is still disabled.
  badExample = myCDDA.withMods (_: []);

  inherit (cataclysmDDA) attachPkgs pkgs wrapCDDA;

  # You can fix it by hand
  goodExample1 = myCDDA.overrideAttrs (old: {
    passthru = old.passthru // {
      pkgs = pkgs.override { build = goodExample1; };
      withMods = wrapCDDA goodExample1;
    };
  });

  # or by using a helper function `attachPkgs`.
  goodExample2 = attachPkgs pkgs myCDDA;
in

# badExample                      # parallel building disabled
# goodExample1.withMods (_: [])   # parallel building enabled
goodExample2.withMods (_: [])     # parallel building enabled
```

16.17.3. Customizing with mods

To install Cataclysm DDA with mods of your choice, you can use `withMods` attribute:

```
cataclysm-dda.withMods (mods: with mods; [
  tileset.UndeadPeople
])
```

All mods, soundpacks, and tilesets available in nixpkgs are found in `cataclysmDDA.pkgs`.

Here is an example to modify existing mods and/or add more mods not available in nixpkgs:

```nix
let
  customMods = self: super: lib.recursiveUpdate super {
    # Modify existing mod
    tileset.UndeadPeople = super.tileset.UndeadPeople.overrideAttrs (old: {
      # If you like to apply a patch to the tileset for example
      patches = [ ./path/to/your.patch ];
    });

    # Add another mod
    mod.Awesome = cataclysmDDA.buildMod {
      modName = "Awesome";
      version = "0.x";
      src = fetchFromGitHub {
        owner = "Someone";
        repo = "AwesomeMod";
        rev = "...";
        sha256 = "...";
      };
      # Path to be installed in the unpacked source (default: ".")
      modRoot = "contents/under/this/path/will/be/installed";
    };

    # Add another soundpack
    soundpack.Fantastic = cataclysmDDA.buildSoundPack {
      # ditto
    };

    # Add another tileset
    tileset.SuperDuper = cataclysmDDA.buildTileSet {
      # ditto
    };
  };
in
cataclysm-dda.withMods (mods: with mods.extend customMods; [
  tileset.UndeadPeople
  mod.Awesome
  soundpack.Fantastic
  tileset.SuperDuper
])
```

16.18. Urxvt

Urxvt, also known as rxvt-unicode, is a highly customizable terminal emulator.

16.18.1. Configuring urxvt

In `nixpkgs`, urxvt is provided by the package `rxvt-unicode`. It can be configured to include your choice of plugins, reducing its closure size from the default configuration which includes all available plugins. To make use of this functionality, use an overlay or directly install an expression that overrides its configuration, such as

```
rxvt-unicode.override {
  configure = { availablePlugins, ... }: {
    plugins = with availablePlugins; [ perls resize-font vtwheel ];
  };
}
```

If the `configure` function returns an attrset without the `plugins` attribute, `availablePlugins` will be used automatically.

In order to add plugins but also keep all default plugins installed, it is possible to use the following method:

```
rxvt-unicode.override {
  configure = { availablePlugins, ... }: {
    plugins = (builtins.attrValues availablePlugins) ++ [ custom-plugin ];
  };
}
```

To get a list of all the plugins available, open the Nix REPL and run

```
$ nix repl
:l <nixpkgs>
map (p: p.name) pkgs.rxvt-unicode.plugins
```

Alternatively, if your shell is bash or zsh and have completion enabled, simply type `nixpkgs.rxvt-unicode.plugins.<tab>`.

In addition to `plugins` the options `extraDeps` and `perlDeps` can be used to install extra packages. `extraDeps` can be used, for example, to provide `xsel` (a clipboard manager) to the clipboard plugin, without installing it globally:

```
rxvt-unicode.override {
  configure = { availablePlugins, ... }: {
```

```
    pluginsDeps = [ xsel ];
  };
}
```

`perlDeps` is a handy way to provide Perl packages to your custom plugins (in `$HOME/.urxvt/ext`). For example, if you need `AnyEvent` you can do:

```
rxvt-unicode.override {
  configure = { availablePlugins, ... }: {
    perlDeps = with perlPackages; [ AnyEvent ];
  };
}
```

16.18.2. Packaging urxvt plugins

Urxvt plugins resides in `pkgs/applications/misc/rxvt-unicode-plugins`. To add a new plugin create an expression in a subdirectory and add the package to the set in `pkgs/applications/misc/rxvt-unicode-plugins/default.nix`.

A plugin can be any kind of derivation, the only requirement is that it should always install perl scripts in `$out/lib/urxvt/perl`. Look for existing plugins for examples.

If the plugin is itself a perl package that needs to be imported from other plugins or scripts, add the following passthrough:

```
passthru.perlPackages = [ "self" ];
```

This will make the urxvt wrapper pick up the dependency and set up the perl path accordingly.

16.19. Weechat

Weechat can be configured to include your choice of plugins, reducing its closure size from the default configuration which includes all available plugins. To make use of this functionality, install an expression that overrides its configuration such as

```
weechat.override {configure = {availablePlugins, ...}: {
    plugins = with availablePlugins; [ python perl ];
  }
}
```

If the `configure` function returns an attrset without the `plugins` attribute, `availablePlugins` will be used automatically.

The plugins currently available are `python`, `perl`, `ruby`, `guile`, `tcl` and `lua`.

The python and perl plugins allows the addition of extra libraries. For instance, the `inotify.py` script in `weechat-scripts` requires D-Bus or libnotify, and the `fish.py` script requires `pycrypto`. To use these scripts, use the plugin's `withPackages` attribute:

```
weechat.override { configure = {availablePlugins, ...}: {
    plugins = with availablePlugins; [
            (python.withPackages (ps: with ps; [ pycrypto python-dbus ]))
        ];
    };
}
```

In order to also keep all default plugins installed, it is possible to use the following method:

```
weechat.override { configure = { availablePlugins, ... }: {
  plugins = builtins.attrValues (availablePlugins // {
    python = availablePlugins.python.withPackages (ps: with ps; [ pycrypto
  });
}; }
```

WeeChat allows to set defaults on startup using the `--run-command`. The `configure` method can be used to pass commands to the program:

```
weechat.override {
  configure = { availablePlugins, ... }: {
    init = ''
      /set foo bar
      /server add freenode chat.freenode.org
    '';
  };
}
```

Further values can be added to the list of commands when running `weechat --run-command "your-commands"`.

Additionally it's possible to specify scripts to be loaded when starting `weechat`. These will be loaded before the commands from `init`:

```
weechat.override {
  configure = { availablePlugins, ... }: {
    scripts = with pkgs.weechatScripts; [
      weechat-xmpp weechat-matrix-bridge wee-slack
```

```
    ];
    init = ''
      /set plugins.var.python.jabber.key "val"
    '':
  };
}
```

In `nixpkgs` there's a subpackage which contains derivations for WeeChat scripts. Such derivations expect a `passthru.scripts` attribute which contains a list of all scripts inside the store path. Furthermore all scripts have to live in `$out/share`. An exemplary derivation looks like this:

```
{ stdenv, fetchurl }:

stdenv.mkDerivation {
  name = "exemplary-weechat-script";
  src = fetchurl {
    url = "https://scripts.tld/your-scripts.tar.gz";
    sha256 = "...";
  };
  passthru.scripts = [ "foo.py" "bar.lua" ];
  installPhase = ''
    mkdir $out/share
    cp foo.py $out/share
    cp bar.lua $out/share
  '';
}
```

16.20. X.org

The Nix expressions for the X.org packages reside in `pkgs/servers/x11/xorg/default.nix`. This file is automatically generated from lists of tarballs in an X.org release. As such it should not be modified directly; rather, you should modify the lists, the generator script or the file `pkgs/servers/x11/xorg/overrides.nix`, in which you can override or add to the derivations produced by the generator.

16.20.1. Katamari Tarballs

X.org upstream releases used to include katamari releases, which included a holistic recommended version for each tarball, up until 7.7. To create a list of tarballs in a katamari release:

```
export release="X11R7.7"
export url="mirror://xorg/$release/src/everything/"
cat $(PRINT_PATH=1 nix-prefetch-url $url | tail -n 1) \
```

```
| perl -e 'while (<>) { if (/(href|HREF)="([^"]*.bz2)"/) { print "$ENV{'u
| sort > "tarballs-$release.list"
```

16.20.2. Individual Tarballs

The upstream release process for X11R7.8 does not include a planned katamari. Instead, each component of X.org is released as its own tarball. We maintain `pkgs/servers/x11/xorg/tarballs.list` as a list of tarballs for each individual package. This list includes X.org core libraries and protocol descriptions, extra newer X11 interface libraries, like `xorg.libxcb`, and classic utilities which are largely unused but still available if needed, like `xorg.imake`.

16.20.3. Generating Nix Expressions

The generator is invoked as follows:

```
cd pkgs/servers/x11/xorg
<tarballs.list perl ./generate-expr-from-tarballs.pl
```

For each of the tarballs in the `.list` files, the script downloads it, unpacks it, and searches its `configure.ac` and `*.pc.in` files for dependencies. This information is used to generate `default.nix`. The generator caches downloaded tarballs between runs. Pay close attention to the `NOT FOUND: $NAME` messages at the end of the run, since they may indicate missing dependencies. (Some might be optional dependencies, however.)

16.20.4. Overriding the Generator

If the expression for a package requires derivation attributes that the generator cannot figure out automatically (say, `patches` or a `postInstall` hook), you should modify `pkgs/servers/x11/xorg/overrides.nix`.

Part IV. Contributing to Nixpkgs

Table of Contents

17. Quick Start to Adding a Package

18. Coding conventions

19. Submitting changes

20. Vulnerability Roundup

21. Reviewing contributions

Chapter 17. Quick Start to Adding a Package

To add a package to Nixpkgs:

1. Checkout the Nixpkgs source tree:

```
$ git clone https://github.com/NixOS/nixpkgs
$ cd nixpkgs
```

2. Find a good place in the Nixpkgs tree to add the Nix expression for your package. For instance, a library package typically goes into `pkgs/development/libraries/pkgname`, while a web browser goes into `pkgs/applications/networking/browsers/pkgname`. See Section 18.3, "File naming and organisation" for some hints on the tree organisation. Create a directory for your package, e.g.

```
$ mkdir pkgs/development/libraries/libfoo
```

3. In the package directory, create a Nix expression — a piece of code that describes how to build the package. In this case, it should be a *function* that is called with the package dependencies as arguments, and returns a build of the package in the Nix store. The expression should usually be called `default.nix`.

```
$ emacs pkgs/development/libraries/libfoo/default.nix
$ git add pkgs/development/libraries/libfoo/default.nix
```

   You can have a look at the existing Nix expressions under `pkgs/` to see how it's done. Here are some good ones:

   - GNU Hello: `pkgs/applications/misc/hello/default.nix`. Trivial package, which specifies some `meta` attributes which is good practice.
   - GNU cpio: `pkgs/tools/archivers/cpio/default.nix`. Also a simple package. The generic builder in `stdenv` does everything for you. It has no dependencies beyond `stdenv`.
   - GNU Multiple Precision arithmetic library (GMP): `pkgs/development/libraries/gmp/5.1.x.nix`. Also done by the generic builder, but has a dependency on `m4`.
   - Pan, a GTK-based newsreader: `pkgs/applications/networking/newsreaders/pan/default.nix`. Has an optional dependency on `gtkspell`, which is only built if `spellCheck` is `true`.

- Apache HTTPD: `pkgs/servers/http/apache-httpd/2.4.nix`. A bunch of optional features, variable substitutions in the configure flags, a post-install hook, and miscellaneous hackery.
- Thunderbird: `pkgs/applications/networking/mailreaders/thunderbird/default.nix`. Lots of dependencies.
- JDiskReport, a Java utility: `pkgs/tools/misc/jdiskreport/default.nix`. Nixpkgs doesn't have a decent `stdenv` for Java yet so this is pretty ad-hoc.
- XML::Simple, a Perl module: `pkgs/top-level/perl-packages.nix` (search for the `XMLSimple` attribute). Most Perl modules are so simple to build that they are defined directly in `perl-packages.nix`; no need to make a separate file for them.
- Adobe Reader: `pkgs/applications/misc/adobe-reader/default.nix`. Shows how binary-only packages can be supported. In particular the builder uses `patchelf` to set the RUNPATH and ELF interpreter of the executables so that the right libraries are found at runtime.

Some notes:

- All `meta` attributes are optional, but it's still a good idea to provide at least the `description`, `homepage` and `license`.
- You can use `nix-prefetch-url url` to get the SHA-256 hash of source distributions. There are similar commands as `nix-prefetch-git` and `nix-prefetch-hg` available in `nix-prefetch-scripts` package.
- A list of schemes for `mirror://` URLs can be found in `pkgs/build-support/fetchurl/mirrors.nix`.

The exact syntax and semantics of the Nix expression language, including the built-in function, are described in the Nix manual in the chapter on writing Nix expressions.

4. Add a call to the function defined in the previous step to `pkgs/top-level/all-packages.nix` with some descriptive name for the variable, e.g. `libfoo`.

```
$ emacs pkgs/top-level/all-packages.nix
```

The attributes in that file are sorted by category (like "Development / Libraries") that more-or-less correspond to the directory structure of Nixpkgs, and then by attribute name.

5. To test whether the package builds, run the following command from the root of the nixpkgs source tree:

```
$ nix-build -A libfoo
```

where `libfoo` should be the variable name defined in the previous step. You may want to add the

flag `-K` to keep the temporary build directory in case something fails. If the build succeeds, a symlink `./result` to the package in the Nix store is created.

6. If you want to install the package into your profile (optional), do

```
$ nix-env -f . -iA libfoo
```

7. Optionally commit the new package and open a pull request to nixpkgs, or use the Patches category on Discourse for sending a patch without a GitHub account.

Chapter 18. Coding conventions

Table of Contents

18.1. Syntax

18.2. Package naming

18.3. File naming and organisation

18.4. Fetching Sources

18.5. Obtaining source hash

18.6. Patches

18.7. Package tests

18.1. Syntax

- Use 2 spaces of indentation per indentation level in Nix expressions, 4 spaces in shell scripts.
- Do not use tab characters, i.e. configure your editor to use soft tabs. For instance, use `(setq-default indent-tabs-mode nil)` in Emacs. Everybody has different tab settings so it's asking for trouble.
- Use `lowerCamelCase` for variable names, not `UpperCamelCase`. Note, this rule does not apply to package attribute names, which instead follow the rules in Section 18.2, "Package naming".
- Function calls with attribute set arguments are written as

```
foo {
  arg = ...;
}
```

not

```
foo
{
  arg = ...;
}
```

Also fine is

```
foo { arg = ...; }
```

if it's a short call.

- In attribute sets or lists that span multiple lines, the attribute names or list elements should be aligned:

```
# A long list.
list = [
  elem1
  elem2
  elem3
];

# A long attribute set.
attrs = {
  attr1 = short_expr;
  attr2 =
    if true then big_expr else big_expr;
};

# Combined
listOfAttrs = [
  {
    attr1 = 3;
    attr2 = "fff";
  }
  {
    attr1 = 5;
    attr2 = "ggg";
  }
];
```

- Short lists or attribute sets can be written on one line:

```
# A short list.
list = [ elem1 elem2 elem3 ];
```

```
# A short set.
attrs = { x = 1280; y = 1024; };
```

- Breaking in the middle of a function argument can give hard-to-read code, like

```
someFunction { x = 1280;
  y = 1024; } otherArg
  yetAnotherArg
```

(especially if the argument is very large, spanning multiple lines).

Better:

```
someFunction
  { x = 1280; y = 1024; }
  otherArg
  yetAnotherArg
```

or

```
let res = { x = 1280; y = 1024; };
in someFunction res otherArg yetAnotherArg
```

- The bodies of functions, asserts, and withs are not indented to prevent a lot of superfluous indentation levels, i.e.

```
{ arg1, arg2 }:
assert system == "i686-linux";
stdenv.mkDerivation { ...
```

not

```
{ arg1, arg2 }:
  assert system == "i686-linux";
    stdenv.mkDerivation { ...
```

- Function formal arguments are written as:

```
{ arg1, arg2, arg3 }:
```

but if they don't fit on one line they're written as:

```
{ arg1, arg2, arg3
, arg4, ...
, # Some comment...
  argN
}:
```

- Functions should list their expected arguments as precisely as possible. That is, write

```
{ stdenv, fetchurl, perl }: ...
```

instead of

```
args: with args; ...
```

or

```
{ stdenv, fetchurl, perl, ... }: ...
```

For functions that are truly generic in the number of arguments (such as wrappers around `mkDerivation`) that have some required arguments, you should write them using an `@`-pattern:

```
{ stdenv, doCoverageAnalysis ? false, ... } @ args:

stdenv.mkDerivation (args // {
  ... if doCoverageAnalysis then "bla" else "" ...
})
```

instead of

```
args:

args.stdenv.mkDerivation (args // {
  ... if args ? doCoverageAnalysis && args.doCoverageAnalysis then "bla" e
})
```

- Unnecessary string conversions should be avoided. Do

```
rev = version;
```

instead of

```
rev = "${version}";
```

- Arguments should be listed in the order they are used, with the exception of `lib`, which always goes first.
- The top-level `lib` must be used in the master and 21.05 branch over its alias `stdenv.lib` as it now causes evaluation errors when aliases are disabled which is the case for ofborg. `lib` is unrelated to `stdenv`, and so `stdenv.lib` should only be used as a convenience alias when developing locally to avoid having to modify the function inputs just to test something out.

18.2. Package naming

The key words *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* in this section are to be interpreted as described in RFC 2119. Only *emphasized* words are to be interpreted in this way.

In Nixpkgs, there are generally three different names associated with a package:

- The `name` attribute of the derivation (excluding the version part). This is what most users see, in particular when using `nix-env`.
- The variable name used for the instantiated package in `all-packages.nix`, and when passing it as a dependency to other functions. Typically this is called the *package attribute name*. This is what Nix expression authors see. It can also be used when installing using `nix-env -iA`.
- The filename for (the directory containing) the Nix expression.

Most of the time, these are the same. For instance, the package `e2fsprogs` has a `name` attribute `"e2fsprogs-version"`, is bound to the variable name `e2fsprogs` in `all-packages.nix`, and the Nix expression is in `pkgs/os-specific/linux/e2fsprogs/default.nix`.

There are a few naming guidelines:

- The `name` attribute *should* be identical to the upstream package name.
- The `name` attribute *must not* contain uppercase letters — e.g., `"mplayer-1.0rc2"` instead of `"MPlayer-1.0rc2"`.
- The version part of the `name` attribute *must* start with a digit (following a dash) — e.g., `"hello-0.3.1rc2"`.
- If a package is not a release but a commit from a repository, then the version part of the name *must* be the date of that (fetched) commit. The date *must* be in `"YYYY-MM-DD"` format. Also append `"unstable"` to the name - e.g., `"pkgname-unstable-2014-09-23"`.
- Dashes in the package name *should* be preserved in new variable names, rather than converted to underscores or camel cased — e.g., `http-parser` instead of `http_parser` or `httpParser`. The hyphenated style is preferred in all three package names.

- If there are multiple versions of a package, this *should* be reflected in the variable names in `all-packages.nix`, e.g. `json-c-0-9` and `json-c-0-11`. If there is an obvious "default" version, make an attribute like `json-c = json-c-0-9;`. See also Section 18.3.2, "Versioning"

18.3. File naming and organisation

Names of files and directories should be in lowercase, with dashes between words — not in camel case. For instance, it should be `all-packages.nix`, not `allPackages.nix` or `AllPackages.nix`.

18.3.1. Hierarchy

Each package should be stored in its own directory somewhere in the `pkgs/` tree, i.e. in `pkgs/category/subcategory/.../pkgname`. Below are some rules for picking the right category for a package. Many packages fall under several categories; what matters is the *primary* purpose of a package. For example, the `libxml2` package builds both a library and some tools; but it's a library foremost, so it goes under `pkgs/development/libraries`.

When in doubt, consider refactoring the `pkgs/` tree, e.g. creating new categories or splitting up an existing category.

If it's used to support *software development*:

- **If it's a *library* used by other packages:**
  - `development/libraries` (e.g. `libxml2`)
- **If it's a *compiler*:**
  - `development/compilers` (e.g. `gcc`)
- **If it's an *interpreter*:**
  - `development/interpreters` (e.g. `guile`)
- **If it's a (set of) development *tool(s)*:**
  - **If it's a *parser generator* (including lexers):**
    - `development/tools/parsing` (e.g. `bison`, `flex`)
  - **If it's a *build manager*:**
    - `development/tools/build-managers` (e.g. `gnumake`)
  - **Else:**
    - `development/tools/misc` (e.g. `binutils`)
- **Else:**
  - `development/misc`

If it's a (set of) *tool(s)*:

(A tool is a relatively small program, especially one intended to be used non-interactively.)

- **If it's for *networking*:**
  - `tools/networking` (e.g. `wget`)
- **If it's for *text processing*:**
  - `tools/text` (e.g. `diffutils`)
- **If it's a *system utility*, i.e., something related or essential to the operation of a system:**
  - `tools/system` (e.g. `cron`)
- **If it's an *archiver* (which may include a compression function):**
  - `tools/archivers` (e.g. `zip`, `tar`)
- **If it's a *compression* program:**
  - `tools/compression` (e.g. `gzip`, `bzip2`)
- **If it's a *security*-related program:**
  - `tools/security` (e.g. `nmap`, `gnupg`)
- **Else:**
  - `tools/misc`

If it's a *shell*:

- `shells` (e.g. `bash`)

If it's a *server*:

- **If it's a web server:**
  - `servers/http` (e.g. `apache-httpd`)
- **If it's an implementation of the X Windowing System:**
  - `servers/x11` (e.g. `xorg` — this includes the client libraries and programs)
- **Else:**
  - `servers/misc`

If it's a *desktop environment*:

- `desktops` (e.g. `kde`, `gnome`, `enlightenment`)

If it's a *window manager*:

- `applications/window-managers` (e.g. `awesome`, `stumpwm`)

If it's an *application*:

A (typically large) program with a distinct user interface, primarily used interactively.

- **If it's a *version management system*:**
  - `applications/version-management` (e.g. `subversion`)
- **If it's a *terminal emulator*:**
  - `applications/terminal-emulators` (e.g. `alacritty` or `rxvt` or `termite`)
- **If it's for *video playback / editing*:**
  - `applications/video` (e.g. `vlc`)
- **If it's for *graphics viewing / editing*:**
  - `applications/graphics` (e.g. `gimp`)
- **If it's for *networking*:**
  - **If it's a *mailreader*:**
    - `applications/networking/mailreaders` (e.g. `thunderbird`)
  - **If it's a *newsreader*:**
    - `applications/networking/newsreaders` (e.g. `pan`)
  - **If it's a *web browser*:**
    - `applications/networking/browsers` (e.g. `firefox`)
  - **Else:**
    - `applications/networking/misc`
- **Else:**
  - `applications/misc`

If it's *data* (i.e., does not have a straight-forward executable semantics):

- **If it's a *font*:**
  - `data/fonts`
- **If it's an *icon theme*:**
  - `data/icons`

- **If it's related to *SGML/XML processing*:**
  - **If it's an *XML DTD*:**
    - `data/sgml+xml/schemas/xml-dtd` (e.g. `docbook`)
  - **If it's an *XSLT stylesheet*:**

    (Okay, these are executable…)

    - `data/sgml+xml/stylesheets/xslt` (e.g. `docbook-xsl`)
- **If it's a *theme* for a *desktop environment*, a *window manager* or a *display manager*:**
  - `data/themes`

If it's a *game*:

- `games`

Else:

- `misc`

18.3.2. Versioning

Because every version of a package in Nixpkgs creates a potential maintenance burden, old versions of a package should not be kept unless there is a good reason to do so. For instance, Nixpkgs contains several versions of GCC because other packages don't build with the latest version of GCC. Other examples are having both the latest stable and latest pre-release version of a package, or to keep several major releases of an application that differ significantly in functionality.

If there is only one version of a package, its Nix expression should be named `e2fsprogs/default.nix`. If there are multiple versions, this should be reflected in the filename, e.g. `e2fsprogs/1.41.8.nix` and `e2fsprogs/1.41.9.nix`. The version in the filename should leave out unnecessary detail. For instance, if we keep the latest Firefox 2.0.x and 3.5.x versions in Nixpkgs, they should be named `firefox/2.0.nix` and `firefox/3.5.nix`, respectively (which, at a given point, might contain versions `2.0.0.20` and `3.5.4`). If a version requires many auxiliary files, you can use a subdirectory for each version, e.g. `firefox/2.0/default.nix` and `firefox/3.5/default.nix`.

All versions of a package *must* be included in `all-packages.nix` to make sure that they evaluate correctly.

18.4. Fetching Sources

There are multiple ways to fetch a package source in nixpkgs. The general guideline is that you should package reproducible sources with a high degree of availability. Right now there is only one fetcher which has mirroring support and that is `fetchurl`. Note that you should also prefer protocols which have a corresponding proxy environment variable.

You can find many source fetch helpers in `pkgs/build-support/fetch*`.

In the file `pkgs/top-level/all-packages.nix` you can find fetch helpers, these have names on the form `fetchFrom*`. The intention of these are to provide snapshot fetches but using the same api as some of the version controlled fetchers from `pkgs/build-support/`. As an example going from bad to good:

- Bad: Uses `git://` which won't be proxied.

```
src = fetchgit {
  url = "git://github.com/NixOS/nix.git";
  rev = "1f795f9f44607cc5bec70d1300150bfefcef2aae";
  sha256 = "1cw5fszffl5pkpa6s6wjnkiv6lm5k618s32sp60kvmvpy7a2v9kg";
}
```

- Better: This is ok, but an archive fetch will still be faster.

```
src = fetchgit {
  url = "https://github.com/NixOS/nix.git";
  rev = "1f795f9f44607cc5bec70d1300150bfefcef2aae";
  sha256 = "1cw5fszffl5pkpa6s6wjnkiv6lm5k618s32sp60kvmvpy7a2v9kg";
}
```

- Best: Fetches a snapshot archive and you get the rev you want.

```
src = fetchFromGitHub {
  owner = "NixOS";
  repo = "nix";
  rev = "1f795f9f44607cc5bec70d1300150bfefcef2aae";
  sha256 = "1i2yxndxb6yc9l6c99pypbd92lfq5aac4klq7y2v93c9qvx2cgpc";
}
```

  Find the value to put as `sha256` by running
  `nix run -f '<nixpkgs>' nix-prefetch-github -c nix-prefetch-github --rev 1f795f9f44607cc5bec70d1300150bfefcef2aae NixOS nix`
  or
  `nix-prefetch-url --unpack https://github.com/NixOS/nix/archive/1f795f9f44607cc5bec70d1300150bfefcef2aae.tar.gz`
  .

18.5. Obtaining source hash

Preferred source hash type is sha256. There are several ways to get it.

1. Prefetch URL (with `nix-prefetch-XXX URL`, where `XXX` is one of `url`, `git`, `hg`, `cvs`, `bzr`, `svn`). Hash is printed to stdout.

2. Prefetch by package source (with `nix-prefetch-url '<nixpkgs>' -A PACKAGE.src`, where `PACKAGE` is package attribute name). Hash is printed to stdout.

   This works well when you've upgraded existing package version and want to find out new hash, but is useless if package can't be accessed by attribute or package has multiple sources (`.srcs`, architecture-dependent sources, etc).

3. Upstream provided hash: use it when upstream provides `sha256` or `sha512` (when upstream provides `md5`, don't use it, compute `sha256` instead).

   A little nuance is that `nix-prefetch-*` tools produce hash encoded with `base32`, but upstream usually provides hexadecimal (`base16`) encoding. Fetchers understand both formats. Nixpkgs does not standardize on any one format.

   You can convert between formats with nix-hash, for example:

   ```
   $ nix-hash --type sha256 --to-base32 HASH
   ```

4. Extracting hash from local source tarball can be done with `sha256sum`. Use `nix-prefetch-url file:///path/to/tarball` if you want base32 hash.

5. Fake hash: set fake hash in package expression, perform build and extract correct hash from error Nix prints.

   For package updates it is enough to change one symbol to make hash fake. For new packages, you can use `lib.fakeSha256`, `lib.fakeSha512` or any other fake hash.

   This is last resort method when reconstructing source URL is non-trivial and `nix-prefetch-url -A` isn't applicable (for example, one of `kodi` dependencies). The easiest way then would be replace hash with a fake one and rebuild. Nix build will fail and error message will contain desired hash.

Warning: This method has security problems. Check below for details.

18.5.1. Obtaining hashes securely

Let's say Man-in-the-Middle (MITM) sits close to your network. Then instead of fetching source you can fetch malware, and instead of source hash you get hash of malware. Here are security considerations for this scenario:

- `http://` URLs are not secure to prefetch hash from;
- hashes from upstream (in method 3) should be obtained via secure protocol;

- `https://` URLs are secure in methods 1, 2, 3;
- `https://` URLs are not secure in method 5. When obtaining hashes with fake hash method, TLS checks are disabled. So refetch source hash from several different networks to exclude MITM scenario. Alternatively, use fake hash method to make Nix error, but instead of extracting hash from error, extract `https://` URL and prefetch it with method 1.

18.6. Patches

Patches available online should be retrieved using `fetchpatch`.

```
patches = [
  (fetchpatch {
    name = "fix-check-for-using-shared-freetype-lib.patch";
    url = "http://git.ghostscript.com/?p=ghostpdl.git;a=patch;h=8f5d285";
    sha256 = "1f0k043rng7f0rfl9hhb89qzvvksqmkrikmm38p61yfx51l325xr";
  })
];
```

Otherwise, you can add a `.patch` file to the `nixpkgs` repository. In the interest of keeping our maintenance burden to a minimum, only patches that are unique to `nixpkgs` should be added in this way.

```
patches = [ ./0001-changes.patch ];
```

If you do need to do create this sort of patch file, one way to do so is with git:

1. Move to the root directory of the source code you're patching.

   ```
   $ cd the/program/source
   ```

2. If a git repository is not already present, create one and stage all of the source files.

   ```
   $ git init
   $ git add .
   ```

3. Edit some files to make whatever changes need to be included in the patch.

4. Use git to create a diff, and pipe the output to a patch file:

   ```
   $ git diff > nixpkgs/pkgs/the/package/0001-changes.patch
   ```

If a patch is available online but does not cleanly apply, it can be modified in some fixed ways by using

additional optional arguments for `fetchpatch`:

- `stripLen`: Remove the first `stripLen` components of pathnames in the patch.
- `extraPrefix`: Prefix pathnames by this string.
- `excludes`: Exclude files matching this pattern.
- `includes`: Include only files matching this pattern.
- `revert`: Revert the patch.

Note that because the checksum is computed after applying these effects, using or modifying these arguments will have no effect unless the `sha256` argument is changed as well.

18.7. Package tests

Tests are important to ensure quality and make reviews and automatic updates easy.

Nix package tests are a lightweight alternative to NixOS module tests. They can be used to create simple integration tests for packages while the module tests are used to test services or programs with a graphical user interface on a NixOS VM. Unittests that are included in the source code of a package should be executed in the `checkPhase`.

18.7.1. Writing package tests

This is an example using the `phoronix-test-suite` package with the current best practices.

Add the tests in `passthru.tests` to the package definition like this:

```
{ stdenv, lib, fetchurl, callPackage }:

stdenv.mkDerivation {
  …

  passthru.tests = {
    simple-execution = callPackage ./tests.nix { };
  };

  meta = { … };
}
```

Create `tests.nix` in the package directory:

```
{ runCommand, phoronix-test-suite }:

let
```

```
  inherit (phoronix-test-suite) pname version;
in

runCommand "${pname}-tests" { meta.timeout = 3; }
  ''
    # automatic initial setup to prevent interactive questions
    ${phoronix-test-suite}/bin/phoronix-test-suite enterprise-setup >/dev/n
    # get version of installed program and compare with package version
    if [[ `${phoronix-test-suite}/bin/phoronix-test-suite version` != *"${v
      echo "Error: program version does not match package version"
      exit 1
    fi
    # run dummy command
    ${phoronix-test-suite}/bin/phoronix-test-suite dummy_module.dummy-comma
    # needed for Nix to register the command as successful
    touch $out
  ''
```

18.7.2. Running package tests

You can run these tests with:

```
$ cd path/to/nixpkgs
$ nix-build -A phoronix-test-suite.tests
```

18.7.3. Examples of package tests

Here are examples of package tests:

- Jasmin compile test
- Lobster compile test
- Spacy annotation test
- Libtorch test
- Multiple tests for nanopb

Chapter 19. Submitting changes

Table of Contents

19.1. Making patches

19.2. Submitting changes

19.3. Submitting security fixes

19.4. Deprecating/removing packages

19.5. Pull Request Template

19.6. Hotfixing pull requests

19.7. Commit policy

19.1. Making patches

- Read Manual (How to write packages for Nix).
- Fork the Nixpkgs repository on GitHub.
- Create a branch for your future fix.
  - You can make branch from a commit of your local `nixos-version`. That will help you to avoid additional local compilations. Because you will receive packages from binary cache. For example

    ```
    $ nixos-version --hash
    0998212
    $ git checkout 0998212
    $ git checkout -b 'fix/pkg-name-update'
    ```

  - Please avoid working directly on the `master` branch.
- Make commits of logical units.
- If you removed pkgs or made some major NixOS changes, write about it in the release notes for the next stable release. For example `nixos/doc/manual/release-notes/rl-2003.xml`.
- Check for unnecessary whitespace with `git diff --check` before committing.
- Format the commit in a following way:

  ```
  (pkg-name | nixos/<module>): (from -> to | init at version | refactor | e
  Additional information.
  ```

  - Examples:
    - `nginx: init at 2.0.1`
    - `firefox: 54.0.1 -> 55.0`

    - `nixos/hydra: add bazBaz option`
    - `nixos/nginx: refactor config generation`
- Test your changes. If you work with
  - nixpkgs:
    - update pkg
      - `nix-env -i pkg-name -f <path to your local nixpkgs folder>`
    - add pkg
      - Make sure it's in `pkgs/top-level/all-packages.nix`
      - `nix-env -i pkg-name -f <path to your local nixpkgs folder>`
    - *If you don't want to install pkg in you profile.*
      - `nix-build -A pkg-attribute-name <path to your local nixpkgs folder>/default.nix`
        and check results in the folder `result`. It will appear in the same directory where you did `nix-build`.
    - If you did `nix-env -i pkg-name` you can do `nix-env -e pkg-name` to uninstall it from your system.
  - NixOS and its modules:
    - You can add new module to your NixOS configuration file (usually it's `/etc/nixos/configuration.nix`). And do
      `sudo nixos-rebuild test -I nixpkgs=<path to your local nixpkgs folder> --fast`
      .
- If you have commits `pkg-name: oh, forgot to insert whitespace`: squash commits in this case. Use `git rebase -i`.
- Rebase your branch against current `master`.

19.2. Submitting changes

- Push your changes to your fork of nixpkgs.
- Create the pull request
- Follow the contribution guidelines.

19.3. Submitting security fixes

Security fixes are submitted in the same way as other changes and thus the same guidelines apply.

- If a new version fixing the vulnerability has been released, update the package;
- If the security fix comes in the form of a patch and a CVE is available, then add the patch to the Nixpkgs tree, and apply it to the package. The name of the patch should be the CVE identifier, so e.g. `CVE-2019-13636.patch`; If a patch is fetched the name needs to be set as well, e.g.:

```
(fetchpatch {
  name = "CVE-2019-11068.patch";
  url = "https://gitlab.gnome.org/GNOME/libxslt/commit/e03553605b45c88f0b
  sha256 = "0pkpb4837km15zgg6h57bncp66d5lwrlvkr73h0lanywq7zrwhj8";
})
```

If a security fix applies to both master and a stable release then, similar to regular changes, they are preferably delivered via master first and cherry-picked to the release branch.

Critical security fixes may by-pass the staging branches and be delivered directly to release branches such as `master` and `release-*`.

19.4. Deprecating/removing packages

There is currently no policy when to remove a package.

Before removing a package, one should try to find a new maintainer or fix smaller issues first.

19.4.1. Steps to remove a package from Nixpkgs

We use jbidwatcher as an example for a discontinued project here.

1. Have Nixpkgs checked out locally and up to date.
2. Create a new branch for your change, e.g. `git checkout -b jbidwatcher`
3. Remove the actual package including its directory, e.g. `rm -rf pkgs/applications/misc/jbidwatcher`
4. Remove the package from the list of all packages (`pkgs/top-level/all-packages.nix`).
5. Add an alias for the package name in `pkgs/top-level/aliases.nix` (There is also `pkgs/misc/vim-plugins/aliases.nix`. Package sets typically do not have aliases, so we can't add them there.)

   For example in this case:
   `jbidwatcher = throw "jbidwatcher was discontinued in march 2021"; # added 2021-03-15`

   The throw message should explain in short why the package was removed for users that still have it installed.

6. Test if the changes introduced any issues by running `nix-env -qaP -f . --show-trace`. It should show the list of packages without errors.

7. Commit the changes. Explain again why the package was removed. If it was declared discontinued upstream, add a link to the source.

```
$ git add pkgs/applications/misc/jbidwatcher/default.nix pkgs/top-level
$ git commit
```

   Example commit message:

```
jbidwatcher: remove

project was discontinued in march 2021. the program does not work anymo

https://web.archive.org/web/20210315205723/http://www.jbidwatcher.com/
```

8. Push changes to your GitHub fork with `git push`

9. Create a pull request against Nixpkgs. Mention the package maintainer.

This is how the pull request looks like in this case: https://github.com/NixOS/nixpkgs/pull/116470

19.5. Pull Request Template

The pull request template helps determine what steps have been made for a contribution so far, and will help guide maintainers on the status of a change. The motivation section of the PR should include any extra details the title does not address and link any existing issues related to the pull request.

When a PR is created, it will be pre-populated with some checkboxes detailed below:

19.5.1. Tested using sandboxing

When sandbox builds are enabled, Nix will setup an isolated environment for each build process. It is used to remove further hidden dependencies set by the build environment to improve reproducibility. This includes access to the network during the build outside of `fetch*` functions and files outside the Nix store. Depending on the operating system access to other resources are blocked as well (ex. inter process communication is isolated on Linux); see sandbox in Nix manual for details.

Sandboxing is not enabled by default in Nix due to a small performance hit on each build. In pull requests for nixpkgs people are asked to test builds with sandboxing enabled (see `Tested using sandboxing` in the pull request template) because inhttps://nixos.org/hydra/ sandboxing is also used.

Depending if you use NixOS or other platforms you can use one of the following methods to enable

sandboxing **before** building the package:

- **Globally enable sandboxing on NixOS**: add the following to `configuration.nix`

```
nix.useSandbox = true;
```

- **Globally enable sandboxing on non-NixOS platforms**: add the following to: `/etc/nix/nix.conf`

```
sandbox = true
```

19.5.2. Built on platform(s)

Many Nix packages are designed to run on multiple platforms. As such, it's important to let the maintainer know which platforms your changes have been tested on. It's not always practical to test a change on all platforms, and is not required for a pull request to be merged. Only check the systems you tested the build on in this section.

19.5.3. Tested via one or more NixOS test(s) if existing and applicable for the change (look inside nixos/tests)

Packages with automated tests are much more likely to be merged in a timely fashion because it doesn't require as much manual testing by the maintainer to verify the functionality of the package. If there are existing tests for the package, they should be run to verify your changes do not break the tests. Tests can only be run on Linux. For more details on writing and running tests, see the section in the NixOS manual.

19.5.4. Tested compilation of all pkgs that depend on this change using `nixpkgs-review`

If you are updating a package's version, you can use nixpkgs-review to make sure all packages that depend on the updated package still compile correctly. The `nixpkgs-review` utility can look for and build all dependencies either based on uncommited changes with the `wip` option or specifying a github pull request number.

review changes from pull request number 12345:

```
nix run nixpkgs.nixpkgs-review -c nixpkgs-review pr 12345
```

review uncommitted changes:

```
nix run nixpkgs.nixpkgs-review -c nixpkgs-review wip
```

review changes from last commit:

```
nix run nixpkgs.nixpkgs-review -c nixpkgs-review rev HEAD
```

19.5.5. Tested execution of all binary files (usually in `./result/bin/`)

It's important to test any executables generated by a build when you change or create a package in nixpkgs. This can be done by looking in `./result/bin` and running any files in there, or at a minimum, the main executable for the package. For example, if you make a change to texlive, you probably would only check the binaries associated with the change you made rather than testing all of them.

19.5.6. Meets Nixpkgs contribution standards

The last checkbox is fits CONTRIBUTING.md. The contributing document has detailed information on standards the Nix community has for commit messages, reviews, licensing of contributions you make to the project, etc... Everyone should read and understand the standards the community has for contributing before submitting a pull request.

19.6. Hotfixing pull requests

- Make the appropriate changes in you branch.
- Don't create additional commits, do
  - `git rebase -i`
  - `git push --force` to your branch.

19.7. Commit policy

- Commits must be sufficiently tested before being merged, both for the master and staging branches.
- Hydra builds for master and staging should not be used as testing platform, it's a build farm for changes that have been already tested.
- When changing the bootloader installation process, extra care must be taken. Grub installations cannot be rolled back, hence changes may break people's installations forever. For any non-trivial change to the bootloader please file a PR asking for review, especially from @edolstra.

Figure 19.1. Staging workflow

This GitHub Action brings changes from `master` to `staging-next` and from `staging-next` to `staging` every 6 hours.

19.7.1. Master branch

The `master` branch is the main development branch. It should only see non-breaking commits that do not cause mass rebuilds.

19.7.2. Staging branch

The `staging` branch is a development branch where mass-rebuilds go. It should only see non-breaking mass-rebuild commits. That means it is not to be used for testing, and changes must have been well tested already. If the branch is already in a broken state, please refrain from adding extra new breakages.

19.7.3. Staging-next branch

The `staging-next` branch is for stabilizing mass-rebuilds submitted to the `staging` branch prior to merging them into `master`. Mass-rebuilds should go via the `staging` branch. It should only see non-breaking commits that are fixing issues blocking it from being merged into the `master` branch.

If the branch is already in a broken state, please refrain from adding extra new breakages. Stabilize it for a few days and then merge into master.

19.7.4. Stable release branches

For cherry-picking a commit to a stable release branch ("backporting"), use `git cherry-pick -x <original commit>` so that the original commit id is included in the commit.

Add a reason for the backport by using `git cherry-pick -xe <original commit>` instead when it is not obvious from the original commit message. It is not needed when it's a minor version update that includes security and bug fixes but don't add new features or when the commit fixes an otherwise broken package.

Here is an example of a cherry-picked commit message with good reason description:

```
zfs: Keep trying root import until it works

Works around #11003.
```

```
(cherry picked from commit 98b213a11041af39b39473906b595290e2a4e2f9)

Reason: several people cannot boot with ZFS on NVMe
```

Other examples of reasons are:

- Previously the build would fail due to, e.g., `getaddrinfo` not being defined
- The previous download links were all broken
- Crash when starting on some X11 systems

Chapter 20. Vulnerability Roundup

Table of Contents

20.1. Issues

20.2. Triaging and Fixing

20.1. Issues

Vulnerable packages in Nixpkgs are managed using issues. Currently opened ones can be found using the following:

github.com/NixOS/nixpkgs/issues?q=is:issue+is:open+"Vulnerability+roundup"

Each issue correspond to a vulnerable version of a package; As a consequence:

- One issue can contain several CVEs;
- One CVE can be shared across several issues;
- A single package can be concerned by several issues.

A "Vulnerability roundup" issue usually respects the following format:

```
<link to relevant package search on search.nix.gsc.io>, <link to relevant f

<list of related CVEs, their CVSS score, and the impacted NixOS version>

<list of the scanned Nixpkgs versions>

<list of relevant contributors>
```

Note that there can be an extra comment containing links to previously reported (and still open) issues for

the same package.

20.2. Triaging and Fixing

Note: An issue can be a "false positive" (i.e. automatically opened, but without the package it refers to being actually vulnerable). If you find such a "false positive", comment on the issue an explanation of why it falls into this category, linking as much information as the necessary to help maintainers double check.

If you are investigating a "true positive":

- Find the earliest patched version or a code patch in the CVE details;
- Is the issue already patched (version up-to-date or patch applied manually) in Nixpkgs's `master` branch?
  - **No**:
    - Submit a security fix;
    - Once the fix is merged into `master`, submit the change to the vulnerable release branch(es);
  - **Yes**: Backport the change to the vulnerable release branch(es).
- When the patch has made it into all the relevant branches (`master`, and the vulnerable releases), close the relevant issue(s).

Chapter 21. Reviewing contributions

Table of Contents

21.1. Package updates

21.2. New packages

21.3. Module updates

21.4. New modules

21.5. Other submissions

21.6. Merging pull requests

Warning: The following section is a draft, and the policy for reviewing is still being discussed in issues such as #11166 and #20836.

The Nixpkgs project receives a fairly high number of contributions via GitHub pull requests. Reviewing and approving these is an important task and a way to contribute to the project.

The high change rate of Nixpkgs makes any pull request that remains open for too long subject to conflicts that will require extra work from the submitter or the merger. Reviewing pull requests in a timely manner and being responsive to the comments is the key to avoid this issue. GitHub provides sort filters that can be used to see the most recently and the least recently updated pull requests. We highly encourage looking at this list of ready to merge, unreviewed pull requests.

When reviewing a pull request, please always be nice and polite. Controversial changes can lead to controversial opinions, but it is important to respect every community member and their work.

GitHub provides reactions as a simple and quick way to provide feedback to pull requests or any comments. The thumb-down reaction should be used with care and if possible accompanied with some explanation so the submitter has directions to improve their contribution.

pull request reviews should include a list of what has been reviewed in a comment, so other reviewers and mergers can know the state of the review.

All the review template samples provided in this section are generic and meant as examples. Their usage is optional and the reviewer is free to adapt them to their liking.

21.1. Package updates

A package update is the most trivial and common type of pull request. These pull requests mainly consist of updating the version part of the package name and the source hash.

It can happen that non-trivial updates include patches or more complex changes.

Reviewing process:

- Ensure that the package versioning fits the guidelines.
- Ensure that the commit text fits the guidelines.
- Ensure that the package maintainers are notified.
  - CODEOWNERS will make GitHub notify users based on the submitted changes, but it can happen that it misses some of the package maintainers.
- Ensure that the meta field information is correct.
  - License can change with version updates, so it should be checked to match the upstream license.
  - If the package has no maintainer, a maintainer must be set. This can be the update submitter or a community member that accepts to take maintainership of the package.
- Ensure that the code contains no typos.
- Building the package locally.
  - pull requests are often targeted to the master or staging branch, and building the pull request locally when it is submitted can trigger many source builds.

- It is possible to rebase the changes on nixos-unstable or nixpkgs-unstable for easier review by running the following commands from a nixpkgs clone.

  ```
  $ git fetch origin nixos-unstable
  $ git fetch origin pull/PRNUMBER/head
  $ git rebase --onto nixos-unstable BASEBRANCH FETCH_HEAD
  ```

  - The first command fetches the nixos-unstable branch.
  - The second command fetches the pull request changes, `PRNUMBER` is the number at the end of the pull request title and `BASEBRANCH` the base branch of the pull request.
  - The third command rebases the pull request changes to the nixos-unstable branch.

- The nixpkgs-review tool can be used to review a pull request content in a single command. `PRNUMBER` should be replaced by the number at the end of the pull request title. You can also provide the full github pull request url.

  ```
  $ nix-shell -p nixpkgs-review --run "nixpkgs-review pr PRNUMBER"
  ```

- Running every binary.

Sample template for a package update review is provided below.

```
##### Reviewed points

- [ ] package name fits guidelines
- [ ] package version fits guidelines
- [ ] package build on ARCHITECTURE
- [ ] executables tested on ARCHITECTURE
- [ ] all depending packages build

##### Possible improvements

##### Comments
```

21.2. New packages

New packages are a common type of pull requests. These pull requests consists in adding a new nix-expression for a package.

Review process:

- Ensure that the package versioning fits the guidelines.

- Ensure that the commit name fits the guidelines.
- Ensure that the meta fields contain correct information.
  - License must match the upstream license.
  - Platforms should be set (or the package will not get binary substitutes).
  - Maintainers must be set. This can be the package submitter or a community member that accepts taking up maintainership of the package.
- Report detected typos.
- Ensure the package source:
  - Uses mirror URLs when available.
  - Uses the most appropriate functions (e.g. packages from GitHub should use `fetchFromGitHub`).
- Building the package locally.
- Running every binary.

Sample template for a new package review is provided below.

```
##### Reviewed points

- [ ] package path fits guidelines
- [ ] package name fits guidelines
- [ ] package version fits guidelines
- [ ] package build on ARCHITECTURE
- [ ] executables tested on ARCHITECTURE
- [ ] `meta.description` is set and fits guidelines
- [ ] `meta.license` fits upstream license
- [ ] `meta.platforms` is set
- [ ] `meta.maintainers` is set
- [ ] build time only dependencies are declared in `nativeBuildInputs`
- [ ] source is fetched using the appropriate function
- [ ] phases are respected
- [ ] patches that are remotely available are fetched with `fetchpatch`

##### Possible improvements

##### Comments
```

21.3. Module updates

Module updates are submissions changing modules in some ways. These often contains changes to the

options or introduce new options.

Reviewing process:

- Ensure that the module maintainers are notified.
  - CODEOWNERS will make GitHub notify users based on the submitted changes, but it can happen that it misses some of the package maintainers.
- Ensure that the module tests, if any, are succeeding.
- Ensure that the introduced options are correct.
  - Type should be appropriate (string related types differs in their merging capabilities, `optionSet` and `string` types are deprecated).
  - Description, default and example should be provided.
- Ensure that option changes are backward compatible.
  - `mkRenamedOptionModule` and `mkAliasOptionModule` functions provide way to make option changes backward compatible.
- Ensure that removed options are declared with `mkRemovedOptionModule`
- Ensure that changes that are not backward compatible are mentioned in release notes.
- Ensure that documentations affected by the change is updated.

Sample template for a module update review is provided below.

```
##### Reviewed points

- [ ] changes are backward compatible
- [ ] removed options are declared with `mkRemovedOptionModule`
- [ ] changes that are not backward compatible are documented in release no
- [ ] module tests succeed on ARCHITECTURE
- [ ] options types are appropriate
- [ ] options description is set
- [ ] options example is provided
- [ ] documentation affected by the changes is updated

##### Possible improvements

##### Comments
```

21.4. New modules

New modules submissions introduce a new module to NixOS.

Reviewing process:

- Ensure that the module tests, if any, are succeeding.
- Ensure that the introduced options are correct.
  - Type should be appropriate (string related types differs in their merging capabilities, `optionSet` and `string` types are deprecated).
  - Description, default and example should be provided.
- Ensure that module `meta` field is present
  - Maintainers should be declared in `meta.maintainers`.
  - Module documentation should be declared with `meta.doc`.
- Ensure that the module respect other modules functionality.
  - For example, enabling a module should not open firewall ports by default.

Sample template for a new module review is provided below.

```
##### Reviewed points

- [ ] module path fits the guidelines
- [ ] module tests succeed on ARCHITECTURE
- [ ] options have appropriate types
- [ ] options have default
- [ ] options have example
- [ ] options have descriptions
- [ ] No unneeded package is added to environment.systemPackages
- [ ] meta.maintainers is set
- [ ] module documentation is declared in meta.doc

##### Possible improvements

##### Comments
```

21.5. Other submissions

Other type of submissions requires different reviewing steps.

If you consider having enough knowledge and experience in a topic and would like to be a long-term reviewer for related submissions, please contact the current reviewers for that topic. They will give you information about the reviewing process. The main reviewers for a topic can be hard to find as there is no

list, but checking past pull requests to see who reviewed or git-blaming the code to see who committed to that topic can give some hints.

Container system, boot system and library changes are some examples of the pull requests fitting this category.

21.6. Merging pull requests

It is possible for community members that have enough knowledge and experience on a special topic to contribute by merging pull requests.

Please see the discussion in GitHub nixpkgs issue #50105 for information on how to proceed to be granted this level of access.

In a case a contributor definitively leaves the Nix community, they should create an issue or post on Discourse with references of packages and modules they maintain so the maintainership can be taken over by other contributors.

Chapter 22. Contributing to this documentation

The DocBook sources of the Nixpkgs manual are in the doc subdirectory of the Nixpkgs repository.

You can quickly check your edits with `make`:

```
$ cd /path/to/nixpkgs/doc
$ nix-shell
[nix-shell]$ make
```

If you experience problems, run `make debug` to help understand the docbook errors.

After making modifications to the manual, it's important to build it before committing. You can do that as follows:

```
$ cd /path/to/nixpkgs/doc
$ nix-shell
[nix-shell]$ make clean
[nix-shell]$ nix-build .
```

If the build succeeds, the manual will be in `./result/share/doc/nixpkgs/manual.html`.

Copyright © 2021 NixOS contributors
CC-BY-SA-4.0