



Nixpkgs Manual

Version 23.11pre-git

Table of Contents

[Preface](#)

[Using Nixpkgs](#)

[Platform Support](#)

[Global configuration](#)

[Overlays](#)

[Overriding](#)

[Nixpkgs lib](#)

[Functions reference](#)

[Module System](#)

[Standard environment](#)

[The Standard Environment](#)

[Meta-attributes](#)

[Multiple-output packages](#)

[Cross-compilation](#)

[Platform Notes](#)

v: stable -

[Build helpers](#)[Fetchers](#)[Trivial build helpers](#)[Testers](#)[Special build helpers](#)[Images](#)[Hooks reference](#)[Languages and frameworks](#)[Packages](#)[Development of Nixpkgs](#)[Opening issues](#)[Contributing to Nixpkgs](#)[Quick Start to Adding a Package](#)[Coding conventions](#)[Submitting changes](#)[Vulnerability Roundup](#)[Reviewing contributions](#)[Contributing to Nixpkgs documentation](#)

Preface

Table of Contents

v: stable -

[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 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](#).

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-22.11](#), which includes all packages and modules for the stable NixOS 22.11. 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 bra

v: stable -

correspond to the channel names (e.g. `nixos-22.11-small`).

Using Nixpkgs

Table of Contents

[Platform Support](#)

[Global configuration](#)

[Overlays](#)

[Overriding](#)

Platform Support

Packages receive varying degrees of support, both in terms of maintainer attention and available computation resources for continuous integration (CI).

Below is the list of the best supported platforms:

- `x86_64-linux`: Highest level of support.
- `aarch64-linux`: Well supported, with most packages building successfully in CI.
- `aarch64-darwin`: Receives better support than `x86_64-darwin`.
- `x86_64-darwin`: Receives some support.

There are many other platforms with varying levels of support. The provisional platform list in [Appendix A](#) of [RFC046](#), while not up to date, can be used as guidance.

A more formal definition of the platform support tiers is provided in [RFC046](#), but has not been fully implemented yet.

Global configuration

v: stable -

Table of Contents

[Installing broken packages](#)

[Installing packages on unsupported systems](#)

[Installing unfree packages](#)

[Installing insecure packages](#)

[Modify packages via packageOverrides](#)

[config Options Reference](#)

[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:

v: stable -

```
{  
  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.

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;  
}
```

v: stable -

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.

Installing unfree packages

All users of Nixpkgs are free software users, and many users (and developers) of Nixpkgs want to limit and tightly control their exposure to unfree software. At the same time, many users need (or want) to run some specific pieces of proprietary software. Nixpkgs includes some expressions for unfree software packages. By default unfree software cannot be installed and doesn't show up in searches.

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:

```
{
```

v: stable -

```
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.

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`.

v: stable -

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.

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 { ... };  
  };  
}
```

config Options Reference

The following attributes can be passed in `config`.

[enableParallelBuildingByDefault](#)

Whether to set `enableParallelBuilding` to true by default while building nixpkgs packages. Changing the default may cause a mass rebuild.

Type: boolean

Default: `false`

Declared by:

v: stable -

[pkgs/top-level/config.nix](#)

allowAliases

Whether to expose old attribute names for compatibility.

The recommended setting is to enable this, as it improves backward compatibility, easing updates.

The only reason to disable aliases is for continuous integration purposes. For instance, Nixpkgs should not depend on aliases in its internal code. Projects that aren't Nixpkgs should be cautious of instantly removing all usages of aliases, as migrating too soon can break compatibility with the stable Nixpkgs releases.

Type: boolean

Default: `true`

Declared by:

[pkgs/top-level/config.nix](#)

allowBroken

Whether to allow broken packages.

See [Installing broken packages](#) in the NixOS manual.

Type: boolean

Default: `false || builtins.getenv "NIXPKGS_ALLOW_BROKEN" == "1"`

Declared by:

[pkgs/top-level/config.nix](#)

allowUnfree

Whether to allow unfree packages.

See [Installing unfree packages](#) in the NixOS manual.

Type: boolean

v: stable -

Default: false || builtins.getenv "NIXPKGS_ALLOW_UNFREE" == "1"

Declared by:

[pkgs/top-level/config.nix](#)

allowUnsupportedSystem

Whether to allow unsupported packages.

See [Installing packages on unsupported systems](#) in the NixOS manual.

Type: boolean

Default: false || builtins.getenv "NIXPKGS_ALLOW_UNSUPPORTED_SYSTEM" == "1"

Declared by:

[pkgs/top-level/config.nix](#)

checkMeta

Whether to check that the `meta` attribute of derivations are correct during evaluation time.

Type: boolean

Default: false

Declared by:

[pkgs/top-level/config.nix](#)

configurePlatformsByDefault

Whether to set `configurePlatforms` to `["build" "host"]` by default while building nixpkgs packages. Changing the default may cause a mass rebuild.

Type: boolean

Default: false

Declared by:

v: stable -

[pkgs/top-level/config.nix](#)

contentAddressedByDefault

Whether to set `--contentAddressed` to true by default while building nixpkgs packages.

Changing the default may cause a mass rebuild.

Type: boolean

Default: false

Declared by:

[pkgs/top-level/config.nix](#)

cudaSupport

Whether to build packages with CUDA support by default while building nixpkgs packages.

Changing the default may cause a mass rebuild.

Type: boolean

Default: false

Declared by:

[pkgs/top-level/config.nix](#)

doCheckByDefault

Whether to run `checkPhase` by default while building nixpkgs packages. Changing the default may cause a mass rebuild.

Type: boolean

Default: false

Declared by:

[pkgs/top-level/config.nix](#)

rocmSupport

Whether to build packages with ROCm support by default while building nixpkgs packages.

Changing the default may cause a mass rebuild.

Type: boolean

Default: `false`

Declared by:

[pkgs/top-level/config.nix](#)

showDerivationWarnings

Which warnings to display for potentially dangerous or deprecated values passed into `stdenv.mkDerivation`.

A list of warnings can be found in [/pkgs/stdenv/generic/check-meta.nix](#).

This is not a stable interface; warnings may be added, changed or removed without prior notice.

Type: list of value “maintainerless” (singular enum)

Default: `[]`

Declared by:

[pkgs/top-level/config.nix](#)

strictDepsByDefault

Whether to set `strictDeps` to true by default while building nixpkgs packages. Changing the default may cause a mass rebuild.

Type: boolean

Default: `false`

Declared by:

[pkgs/top-level/config.nix](#)

structuredAttrsByDefault

Whether to set `__structuredAttrs` to true by default while building nixpkgs packages.

v: stable -

Changing the default may cause a mass rebuild.

Type: boolean

Default: `false`

Declared by:

[pkgs/top-level/config.nix](#)

warnUndeclaredOptions

Whether to warn when `config` contains an unrecognized attribute.

Type: boolean

Default: `false`

Declared by:

[pkgs/top-level/config.nix](#)

Declarative Package Management

[Build an environment](#)

[Getting documentation](#)

[GNU info setup](#)

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`:

```
{
```

v: stable -

```
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
      ];
    };
  };
}
```

v: stable -

```
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.

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
```

v: stable -

```
    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/de
    '';
    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" ]; v: stable -
    };
  };
};
```

```
    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/*.*"; do
    if [ -r "$i" ]; then
      . "$i"
    fi
  done
fi
```

Now just run `. "${HOME}/.profile"` and you can start loading man pages from your environment.

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/default/share/man
      export INFOPATH=$HOME/.nix-profile/share/info:/nix/var/nix/profiles/default/share/info
    '';
    myPackages = pkgs.buildEnv {
      name = "my-packages";
      paths = [
        (runCommand "profile" {} ''
          mkdir -p $out/etc/profile.d
        )
      ];
    };
  };
}
```

v: stable -

```
        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.

Overlays

Table of Contents

[Installing overlays](#)

v: stable -

Defining overlays

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.

Installing overlays

Set overlays in NixOS or Nix expressions

Install overlays via configuration lookup

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

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.

Install overlays via configuration lookup

v: stable -

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`.

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:
```

```
{
```

v: stable -

```
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 [the section called “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.

Using overlays to configure alternatives

[BLAS/LAPACK](#)

[Switching the MPI implementation](#)

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.

v: stable -

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 and CBLAS)

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:

v: stable -

```
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:+:}${LD_LIBRARY_PATH}
```

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.

To override `blas` and `lapack` with its reference implementations (i.e. for development purposes), one can use the following overlay:

```
self: super:  
  
{  
    blas = super.blas.override {  
        blasProvider = self.lapack-reference;  
    };  
  
    lapack = super.lapack.override {
```

v: stable -

```
    lapackProvider = self.lapack-reference;
};

}
```

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). The attributes `blas` and `lapack` are `LP64` by default. Their `ILP64` version are provided through the attributes `blas-ilp64` and `lapack-ilp64`. 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 {
  ...
}
```

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`
- [MVAPICH](#), attribute name `mvapich`

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

```
self: super:
```

v: stable -

```
{  
  mpi = self.mpich;  
}
```

Overriding

Table of Contents

[<pkg>.override](#)

[<pkg>.overrideAttrs](#)

[<pkg>.overrideDerivation](#)

[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.

<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; ... }
```

It's also possible to access the previous arguments.

v: stable -

```
pkgs.foo.override (previous: { arg1 = previous.arg1; ... })
```

```
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.

<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 usages:

```
helloBar = pkgs.hello.overrideAttrs (finalAttrs: previousAttrs: {
  pname = previousAttrs.pname + "-bar";
});
```

In the above example, “-bar” is appended to the `pname` attribute, while all other attributes will be retained from the original `hello` package.

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

The argument `finalAttrs` refers to the final attributes passed to `mkDerivation`, plus the `finalPackage` attribute which is equal to the result of `mkDerivation` or subsequent

v: stable -

`overrideAttrs` calls.

If only a one-argument function is written, the argument has the meaning of `previousAttrs`.

Function arguments can be omitted entirely if there is no need to access `previousAttrs` or `finalAttrs`.

```
helloWithDebug = pkgs.hello.overrideAttrs {  
    separateDebugInfo = true;  
};
```

In the above example, the `separateDebugInfo` attribute is overridden to be true, thus building debug info for `helloWithDebug`.

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).

<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

v: stable -

package abstraction. 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";
    hash = "sha256-MxBJRcM2rYzQYwJ5XKxhXTQByvSg5jZc5cSHEZoB2IY=";
  };
  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.

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.

Nixpkgs lib

Table of Contents

[Functions reference](#)

[Module System](#)

Functions reference

Table of Contents

[Nixpkgs Library Functions](#)

[Generators](#)

[Debugging Nix Expressions](#)

v: stable -

[prefer-remote-fetch overlay](#)[pkgs.nix-gitignore](#)

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

Nixpkgs Library Functions

[lib.asserts: assertion functions](#)[lib.attrsets: attribute set functions](#)[lib.strings: string manipulation functions](#)[lib.versions: version string functions](#)[lib.trivial: miscellaneous functions](#)[lib.fixedPoints: explicit recursion functions](#)[lib.lists: list manipulation functions](#)[lib.debug: debugging functions](#)[lib.options: NixOS / nixpkgs option handling](#)[lib.path: path functions](#)[lib.filesystem: filesystem functions](#)[lib.fileset: file set functions](#)[lib.sources: source filtering functions](#)[lib.cli: command-line serialization functions](#)[lib.gvariant: GVariant formatted string serialization functions](#)[lib.customisation: Functions to customise \(derivation-related\) functions, derivatons, or attri](#)

v: stable -

lib.meta: functions for derivation metadata

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

lib.asserts: assertion functions

lib.asserts.assertMsg

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

Throw if pred is false, else return pred. Intended to be used to augment asserts with helpful error messages.

`pred`

Predicate that needs to succeed, otherwise `msg` is thrown

`msg`

Message to throw in case `pred` fails

Example 1. lib.asserts.assertMsg usage example

```
assertMsg false "nope"
stderr> error: nope

assert assertMsg ("foo" == "bar") "foo is not bar, silly"; ""
stderr> error: foo is not bar, silly
```

Located at [lib/asserts.nix:19](#) in `<nixpkgs>`.

lib.asserts.assertOneOf

Type: `assertOneOf :: String -> ComparableVal -> List ComparableVal -> Bool`

v: stable -

Specialized `assertMsg` for checking if `val` is one of the elements of the list `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 2. `lib.asserts.assertOneOf` usage example

```
let sslLibrary = "libressl";
in assertOneOf "sslLibrary" sslLibrary [ "openssl" "bearssl" ]
stderr> error: sslLibrary must be one of [
stderr>   "openssl"
stderr>   "bearssl"
stderr> ], but is: "libressl"
```

Located at [lib/asserts.nix:40](#) in <nixpkgs>.

lib.asserts.assertEachOneOf

Type: `assertEachOneOf :: String -> List ComparableVal -> List ComparableVal -> Bool`

Specialized `assertMsg` for checking if every one of `vals` is one of the elements of the list `xs`. Useful for checking lists of supported attributes.

`name`

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

`vals`

The list of values of what the user provided, to be compared against the values in `xs` v: stable -

xs

The list of valid values

Example 3. **lib.asserts.assertEachOneOf** usage example

```
let sslLibraries = [ "libressl" "bearssl" ];
in assertEachOneOf "sslLibraries" sslLibraries [ "openssl" "bearssl" ]
stderr> error: each element in sslLibraries must be one of [
stderr>   "openssl"
stderr>   "bearssl"
stderr> ], but is: [
stderr>   "libressl"
stderr>   "bearssl"
stderr> ]
```

Located at [lib/asserts.nix:70](#) in <nixpkgs>.

lib.attrsets: attribute set functions

Operations on attribute sets.

lib.attrsets.attrByPath

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

Return an attribute from nested attribute sets.

attrPath

A list of strings representing the attribute path to return from **set**

default

Default value if **attrPath** does not resolve to an existing value

set

The nested attribute set to select values from

v: stable -

Example 4. `lib.attrsets.attrByPath` usage example

```
x = { a = { b = 3; }; }
# ["a" "b"] is equivalent to x.a.b
# 6 is a default value to return if the path does not exist in attrset
attrByPath ["a" "b"] 6 x
=> 3
attrByPath ["z" "z"] 6 x
=> 6
```

Located at [lib/attrsets.nix:30](#) in <nixpkgs>.

lib.attrsets.hasAttrByPath

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

Return if an attribute from nested attribute set exists.

Laws:

1. hasAttrByPath [] x == true

attrPath

A list of strings representing the attribute path to check from `set`

e

The nested attribute set to check

Example 5. `lib.attrsets.hasAttrByPath` usage example

```
x = { a = { b = 3; }; }
hasAttrByPath ["a" "b"] x
=> true
hasAttrByPath ["z" "z"] x
=> false
```

v: stable -

```
hasAttrByPath [] (throw "no need")
=> true
```

Located at [lib/attrsets.nix:63](#) in <nixpkgs>.

lib.attrsets.longestValidPathPrefix

Type: attrsets.longestValidPathPrefix :: [String] -> Value -> [String]

Return the longest prefix of an attribute path that refers to an existing attribute in a nesting of attribute sets.

Can be used after [mapAttrsRecursiveCond](#) to apply a condition, although this will evaluate the predicate function on sibling attributes as well.

Note that the empty attribute path is valid for all values, so this function only throws an exception if any of its inputs does.

Laws:

1. attrsets.longestValidPathPrefix [] x == []
2. hasAttrByPath (attrsets.longestValidPathPrefix p x) x == true

attrPath

A list of strings representing the longest possible path that may be returned.

v

The nested attribute set to check.

Example 6. lib.attrsets.longestValidPathPrefix usage example

```
x = { a = { b = 3; }; }
attrsets.longestValidPathPrefix ["a" "b" "c"] x
```

v: stable -

```
=> ["a" "b"]
attrsets.longestValidPathPrefix ["a"] x
=> ["a"]
attrsets.longestValidPathPrefix ["z" "z"] x
=> []
attrsets.longestValidPathPrefix ["z" "z"] (throw "no need")
=> []
```

Located at [lib/attrsets.nix:107](#) in <nixpkgs>.

lib.attrsets.setAttrByPath

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

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

attrPath

A list of strings representing the attribute path to set

value

The value to set at the location described by **attrPath**

Example 7. lib.attrsets.setAttrByPath usage example

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

Located at [lib/attrsets.nix:150](#) in <nixpkgs>.

lib.attrsets.getAttrFromPath

Type: getAttrFromPath :: [String] -> AttrSet -> Any

Like **attrByPath**, but without a default value. If it doesn't find the path it will throw an error.

attrPath

A list of strings representing the attribute path to get from `set`

set

The nested attribute set to find the value in.

Example 8. `lib.attrsets.getAttrFromPath` usage example

```
x = { a = { b = 3; }; }
getAttrFromPath ["a" "b"] x
=> 3
getAttrFromPath ["z" "z"] x
=> error: cannot find attribute `z.z'
```

Located at [lib/attrsets.nix:176](#) in <nixpkgs>.

lib.attrsets.concatMapAttrs

Type: `concatMapAttrs :: (String -> a -> AttrSet) -> AttrSet -> AttrSet`

Map each attribute in the given set and merge them into a new attribute set.

f

Function argument

v

Function argument

Example 9. `lib.attrsets.concatMapAttrs` usage example

```
concatMapAttrs
  (name: value: {
    ${name} = value;
    ${name + value} = value;
  })
```

v: stable -

```
{ x = "a"; y = "b"; }
=> { x = "a"; xa = "a"; y = "b"; yb = "b"; }
```

Located at [lib/attrsets.nix:198](#) in <nixpkgs>.

lib.attrsets.updateManyAttrsByPath

Type: updateManyAttrsByPath :: [{ path :: [String]; update :: (Any -> Any); }] -> AttrSet -> AttrSet

Update or set specific paths of an attribute set.

Takes a list of updates to apply and an attribute set to apply them to, and returns the attribute set with the updates applied. Updates are represented as { path = ...; update = ...; } values, where **path** is a list of strings representing the attribute path that should be updated, and **update** is a function that takes the old value at that attribute path as an argument and returns the new value it should be.

Properties:

- Updates to deeper attribute paths are applied before updates to more shallow attribute paths
- Multiple updates to the same attribute path are applied in the order they appear in the update list
- If any but the last **path** element leads into a value that is not an attribute set, an error is thrown
- If there is an update for an attribute path that doesn't exist, accessing the argument in the update function causes an error, but intermediate attribute sets are implicitly created as needed

Example 10. lib.attrsets.updateManyAttrsByPath usage example

```
updateManyAttrsByPath [
{
  path = [ "a" "b" ];
  update = old: { d = old.c; };
}
```

v: stable -

```
{  
    path = [ "a" "b" "c" ];  
    update = old: old + 1;  
}  
{  
    path = [ "x" "y" ];  
    update = old: "xy";  
}  
] { a.b.c = 0; }  
=> { a = { b = { d = 1; }; }; x = { y = "xy"; }; }
```

Located at [lib/attrsets.nix:249](#) in <nixpkgs>.

lib.attrsets.attrs

Type: attrs :: [String] -> AttrSet -> [Any]

Return the specified attributes from a set.

nameList

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

set

The set to get attribute values from

Example 11. lib.attrsets.attrs usage example

```
attrs ["a" "b" "c"] as  
=> [as.a as.b as.c]
```

Located at [lib/attrsets.nix:317](#) in <nixpkgs>.

lib.attrsets.attrValues

v: stable -

Type: attrValues :: AttrSet -> [Any]

Return the values of all attributes in the given set, sorted by attribute name.

Example 12. lib.attrsets.attrValues usage example

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

Located at [lib/attrsets.nix:334](#) in <nixpkgs>.

lib.attrsets.getAttrs

Type: getAttrs :: [String] -> AttrSet -> AttrSet

Given a set of attribute names, return the set of the corresponding attributes from the given set.

names

A list of attribute names to get out of **set**

attrs

The set to get the named attributes from

Example 13. lib.attrsets.getAttrs usage example

```
getAttrs [ "a" "b" ] { a = 1; b = 2; c = 3; }  
=> { a = 1; b = 2; }
```

Located at [lib/attrsets.nix:347](#) in <nixpkgs>.

lib.attrsets.catAttrs

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

v: stable -

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

Example 14. `lib.attrsets.catAttrs` usage example

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

Located at [lib/attrsets.nix:363](#) in <nixpkgs>.

lib.attrsets.filterAttrs

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

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

`pred`

Predicate taking an attribute name and an attribute value, which returns `true` to include the attribute, or `false` to exclude the attribute.

`set`

The attribute set to filter

Example 15. `lib.attrsets.filterAttrs` usage example

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

Located at [lib/attrsets.nix:377](#) in <nixpkgs>.

lib.attrsets.filterAttrsRecursive

Type: `filterAttrsRecursive :: (String -> Any -> Bool) -> AttrSet -> AttrSet` v: stable -

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

pred

Predicate taking an attribute name and an attribute value, which returns `true` to include the attribute, or `false` to exclude the attribute.

set

The attribute set to filter

Example 16. lib.attrsets.filterAttrsRecursive usage example

```
filterAttrsRecursive (n: v: v != null) { foo = { bar = null; }; }
=> { foo = {}; }
```

Located at [lib/attrsets.nix:395](#) in <nixpkgs>.

lib.attrsets.foldlAttrs

Type: `foldlAttrs :: (a -> String -> b -> a) -> a -> { ... :: b } -> a`

Like [lib.lists.foldl'](#) but for attribute sets. Iterates over every name-value pair in the given attribute set. The result of the callback function is often called `acc` for accumulator. It is passed between callbacks from left to right and the final `acc` is the return value of `foldlAttrs`.

Attention: There is a completely different function `lib.foldAttrs` which has nothing to do with this function, despite the similar name.

f

Function argument

init

Function argument

set

Function argument

v: stable -

Example 17. lib.attrsets.foldlAttrs usage example

```
foldlAttrs
  (acc: name: value: {
    sum = acc.sum + value;
    names = acc.names ++ [name];
  })
  { sum = 0; names = []; }
  {
    foo = 1;
    bar = 10;
  }
->
{
  sum = 11;
  names = ["bar" "foo"];
}

foldlAttrs
  (throw "function not needed")
123
{};
->
123

foldlAttrs
  (acc: _: _: acc)
3
{ z = throw "value not needed"; a = throw "value not needed"; };
->
3
```

The accumulator doesn't have to be an attrset.
It can be as simple as a number `or` string.

```
foldlAttrs
  (acc: _: v: acc * 10 + v) v: stable -
```

```
1
{ z = 1; a = 2; };
->
121
```

Located at [lib/attrsets.nix:466](#) in <nixpkgs>.

lib.attrsets.foldAttrs

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

Apply fold functions to values grouped by key.

op

A function, given a value and a collector combines the two.

nul

The starting value.

list_of_attrs

A list of attribute sets to fold together by key.

Example 18. lib.attrsets.foldAttrs usage example

```
foldAttrs (item: acc: [item] ++ acc) [] [{ a = 2; } { a = 3; }]
=> { a = [ 2 3 ]; }
```

Located at [lib/attrsets.nix:482](#) in <nixpkgs>.

lib.attrsets.collect

Type: collect :: (AttrSet -> Bool) -> AttrSet -> [x]

Recursively collect sets that verify a given predicate named `pred` from the set `attrs`. The

v: stable -

stopped when the predicate is verified.

pred

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

attrs

The attribute set to recursively collect.

Example 19. lib.attrsets.collect usage example

```
collect isList { a = { b = ["b"]; }; c = [1]; }
=> [[ "b" ] [1]]  
  
collect (x: x ? outPath)
{ a = { outPath = "a/"; }; b = { outPath = "b/"; }; }
=> [{ outPath = "a/"; } { outPath = "b/"; }]
```

Located at [lib/attrsets.nix:511](#) in <nixpkgs>.

lib.attrsets.cartesianProductOfSets

Type: cartesianProductOfSets :: AttrSet -> [AttrSet]

Return the cartesian product of attribute set value combinations.

attrsOfLists

Attribute set with attributes that are lists of values

Example 20. lib.attrsets.cartesianProductOfSets usage example

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

v: stable -

```
{ a = 2; b = 20; }  
]
```

Located at [lib/attrsets.nix:536](#) in <nixpkgs>.

lib.attrsets.nameValuePair

Type: nameValuePair :: String -> Any -> { name :: String; value :: Any; }

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

name

Attribute name

value

Attribute value

Example 21. lib.attrsets.nameValuePair usage example

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

Located at [lib/attrsets.nix:555](#) in <nixpkgs>.

lib.attrsets.mapAttrs

Type: mapAttrs :: (String -> Any -> Any) -> AttrSet -> AttrSet

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

Example 22. lib.attrsets.mapAttrs usage example

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

v: stable -

```
=> { x = "x-foo"; y = "y-bar"; }
```

Located at [lib/attrsets.nix:573](#) in <nixpkgs>.

lib.attrsets.mapAttrs'

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

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`.

`f`

A function, given an attribute's name and value, returns a new `nameValuePair`.

`set`

Attribute set to map over.

Example 23. lib.attrsets.mapAttrs' usage example

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

Located at [lib/attrsets.nix:590](#) in <nixpkgs>.

lib.attrsets.mapAttrsToList

Type: `mapAttrsToList :: (String -> a -> b) -> AttrSet -> [b]`

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

`f`

A function, given an attribute's name and value, returns a new value.

v: stable -

attrs

Attribute set to map over.

Example 24. `lib.attrsets.mapAttrsToList` usage example

```
mapAttrsToList (name: value: name + value)
  { x = "a"; y = "b"; }
=> [ "xa" "yb" ]
```

Located at [lib/attrsets.nix:610](#) in <nixpkgs>.

lib.attrsets.attrsToList

Type: attrsToList :: AttrSet -> [{ name :: String; value :: Any; }]

Deconstruct an attrset to a list of name-value pairs as expected by [builtins.listToAttrs](#). Each element of the resulting list is an attribute set with these attributes:

- **name** (string): The name of the attribute
- **value** (any): The value of the attribute

The following is always true:

```
builtins.listToAttrs (attrsToList attrs) == attrs
```

Warning

The opposite is not always true. In general expect that

```
attrsToList (builtins.listToAttrs list) != list
```

This is because the `listToAttrs` removes duplicate names and doesn't preserve the order of the list.

v: stable -

Example 25. `lib.attrsets.attrsToList` usage example

```
attrsToList { foo = 1; bar = "asdf"; }
=> [ { name = "bar"; value = "asdf"; } { name = "foo"; value = 1; } ]
```

Located at [lib/attrsets.nix:645](#) in <nixpkgs>.

lib.attrsets.mapAttrsRecursive

Type: `mapAttrsRecursive :: ([String] -> a -> b) -> AttrSet -> AttrSet`

Like `mapAttrs`, except that it recursively applies itself to the *leaf* attributes of a potentially-nested attribute set: the second argument of the function will never be an attrset. Also, the first argument of the argument function is a *list* of the attribute names that form the path to the leaf attribute.

For a function that gives you control over what counts as a leaf, see `mapAttrsRecursiveCond`.

`f`

A function, given a list of attribute names and a value, returns a new value.

`set`

Set to recursively map over.

Example 26. `lib.attrsets.mapAttrsRecursive` usage example

```
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-I" }
```

Located at [lib/attrsets.nix:665](#) in <nixpkgs>.

lib.attrsets.mapAttrsRecursiveCond

Type: `mapAttrsRecursiveCond :: (AttrSet -> Bool) -> ([String] -> a v: stable -> AttrSet)`

AttrSet -> AttrSet

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

cond

A function, given the attribute set the recursion is currently at, determine if to recurse deeper into that attribute set.

f

A function, given a list of attribute names and a value, returns a new value.

set

Attribute set to recursively map over.

Example 27. lib.attrsets.mapAttrsRecursiveCond usage example

```
# To prevent recursing into derivations (which are attribute
# sets with the attribute "type" equal to "derivation"):
mapAttrsRecursiveCond
  (as: !(as ? "type" && as.type == "derivation"))
  (x: ... do something ...)
  attrs
```

Located at [lib/attrsets.nix:690](#) in <nixpkgs>.

lib.attrsets.genAttrs

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

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

names

Names of values in the resulting attribute set.

v: stable -

f

A function, given the name of the attribute, returns the attribute's value.

Example 28. `lib.attrsets.genAttrs` usage example

```
genAttrs [ "foo" "bar" ] (name: "x_" + name)
=> { foo = "x_foo"; bar = "x_bar"; }
```

Located at [lib/attrsets.nix:719](#) in <nixpkgs>.

lib.attrsets.isDerivation

Type: `isDerivation :: Any -> Bool`

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

value

Value to check.

Example 29. `lib.attrsets.isDerivation` usage example

```
nixpkgs = import <nixpkgs> {}
isDerivation nixpkgs.ruby
=> true
isDerivation "foobar"
=> false
```

Located at [lib/attrsets.nix:740](#) in <nixpkgs>.

lib.attrsets.toDerivation

Type: `toDerivation :: Path -> Derivation`

v: stable -

Converts a store path to a fake derivation.

path

A store path to convert to a derivation.

Located at [lib/attrsets.nix:749](#) in <nixpkgs>.

lib.attrsets.optionalAttrs

Type: optionalAttrs :: Bool -> AttrSet -> AttrSet

If `cond` is true, return the attribute set `as`, otherwise 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 30. lib.attrsets.optionalAttrs usage example

```
optionalAttrs (true) { my = "set"; }
=> { my = "set"; }
optionalAttrs (false) { my = "set"; }
=> { }
```

Located at [lib/attrsets.nix:777](#) in <nixpkgs>.

lib.attrsets.zipAttrsWithNames

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

Merge sets of attributes and use the function `f` to merge attributes values.

names

v: stable -

List of attribute names to zip.

f

A function, accepts an attribute name, all the values, and returns a combined value.

sets

List of values from the list of attribute sets.

Example 31. `lib.attrsets.zipAttrsWithNames` usage example

```
zipAttrsWithNames ["a"] (name: vs: vs) [{a = "x";} {a = "y"; b = "z";}]  
=> { a = ["x" "y"]; }
```

Located at [lib/attrsets.nix:795](#) in <nixpkgs>.

lib.attrsets.zipAttrsWith

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

Merge sets of attributes and use the function f to merge attribute values. Like

`lib.attrsets.zipAttrsWithNames` with all key names are passed for `names`.

Implementation note: Common names appear multiple times in the list of names, hopefully this does not affect the system because the maximal laziness avoid computing twice the same expression and `listToAttrs` does not care about duplicated attribute names.

Example 32. `lib.attrsets.zipAttrsWith` usage example

```
zipAttrsWith (name: values: values) [{a = "x";} {a = "y"; b = "z";}]  
=> { a = ["x" "y"]; b = ["z"]; }
```

Located at [lib/attrsets.nix:823](#) in <nixpkgs>.

v: stable -

lib.attrsets.zipAttrs

Type: `zipAttrs :: [AttrSet] -> AttrSet`

Merge sets of attributes and combine each attribute value in to a list.

Like `lib.attrsets.zipAttrsWith` with `(name: values: values)` as the function.

sets

List of attribute sets to zip together.

Example 33. lib.attrsets.zipAttrs usage example

```
zipAttrs [{a = "x";} {a = "y"; b = "z";}]
=> { a = ["x" "y"]; b = ["z"]; }
```

Located at [lib/attrsets.nix:838](#) in <nixpkgs>.

lib.attrsets.mergeAttrsList

Type: `mergeAttrsList :: [Attrs] -> Attrs`

Merge a list of attribute sets together using the `//` operator. In case of duplicate attributes, values from later list elements take precedence over earlier ones. The result is the same as `foldl mergeAttrs { }`, but the performance is better for large inputs. For n list elements, each with an attribute set containing m unique attributes, the complexity of this operation is $O(nm \log n)$.

list

Function argument

Example 34. lib.attrsets.mergeAttrsList usage example

```
mergeAttrsList [ { a = 0; b = 1; } { c = 2; d = 3; } ]
=> { a = 0; b = 1; c = 2; d = 3; }
mergeAttrsList [ { a = 0; } { a = 1; } ]
```

v: stable -

```
=> { a = 1; }
```

Located at [lib/attrsets.nix:858](#) in <nixpkgs>.

lib.attrsets.recursiveUpdateUntil

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

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 satisfied, the value of the first attribute set is replaced by the value of the second attribute set.

pred

Predicate, taking the path to the current attribute as a list of strings for attribute names, and the two values at that path from the original arguments.

lhs

Left attribute set of the merge.

rhs

Right attribute set of the merge.

Example 35. lib.attrsets.recursiveUpdateUntil usage example

```
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;
```

v: stable -

```
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  
}
```

Located at [lib/attrsets.nix:910](#) in <nixpkgs>.

lib.attrsets.recursiveUpdate

Type: recursiveUpdate :: AttrSet -> AttrSet -> AttrSet

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

Left attribute set of the merge.

rhs

Right attribute set of the merge.

Example 36. lib.attrsets.recursiveUpdate usage example

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

v: stable -

```
boot.loader.grub.enable = true;  
boot.loader.grub.device = "";  
}
```

Located at [lib/attrsets.nix:950](#) in <nixpkgs>.

lib.attrsets.matchAttrs

Type: `matchAttrs :: AttrSet -> AttrSet -> Bool`

Recurse into every attribute set of the first argument and check that:

- Each attribute path also exists in the second argument.
- If the attribute's value is not a nested attribute set, it must have the same value in the right argument.

pattern

Attribute set structure to match

attrs

Attribute set to check

Example 37. lib.attrsets.matchAttrs usage example

```
matchAttrs { cpu = {}; } { cpu = { bits = 64; }; }  
=> true
```

Located at [lib/attrsets.nix:970](#) in <nixpkgs>.

lib.attrsets.overrideExisting

Type: `overrideExisting :: AttrSet -> AttrSet -> AttrSet`

Override only the attributes that are already present in the old set useful for deep-overriding

v: stable -

old

Original attribute set

new

Attribute set with attributes to override in `old`.

Example 38. `lib.attrsets.overrideExisting` usage example

```
overrideExisting {} { a = 1; }
=> {}
overrideExisting { b = 2; } { a = 1; }
=> { b = 2; }
overrideExisting { a = 3; b = 2; } { a = 1; }
=> { a = 1; b = 2; }
```

Located at [lib/attrsets.nix:1006](#) in `<nixpkgs>`.

lib.attrsets.showAttrPath

Type: `showAttrPath :: [String] -> String`

Turns a list of strings into a human-readable description of those strings represented as an attribute path. The result of this function is not intended to be machine-readable. Create a new attribute set with `value` set at the nested attribute location specified in `attrPath`.

path

Attribute path to render to a string

Example 39. `lib.attrsets.showAttrPath` usage example

```
showAttrPath [ "foo" "10" "bar" ]
=> "foo.\\"10\\".bar"
showAttrPath []
=> "<root attribute path>"
```

v: stable -

Located at [lib/attrsets.nix:1028](#) in <nixpkgs>.

lib.attrsets.getOutput

Type: getOutput :: String -> Derivation -> String

Get a package output. If no output is found, fallback to .out and then to the default.

output

Function argument

pkg

Function argument

Example 40. lib.attrsets.getOutput usage example

```
getOutput "dev" pkgs.openssl
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-1.0.1r-dev"
```

Located at [lib/attrsets.nix:1045](#) in <nixpkgs>.

lib.attrsets.getBin

Type: getBin :: Derivation -> String

Get a package's bin output. If the output does not exist, fallback to .out and then to the default.

Example 41. lib.attrsets.getBin usage example

```
getBin pkgs.openssl
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-1.0.1r"
```

Located at [lib/attrsets.nix:1060](#) in <nixpkgs>.

v: stable -

lib.attrsets.getLib

Type: getLib :: Derivation -> String

Get a package's `lib` output. If the output does not exist, fallback to `.out` and then to the default.

Example 42. lib.attrsets.getLib usage example

```
getLib pkgs.openssl  
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-1.0.1r-lib"
```

Located at [lib/attrsets.nix:1073](#) in <nixpkgs>.

lib.attrsets.getDev

Type: getDev :: Derivation -> String

Get a package's `dev` output. If the output does not exist, fallback to `.out` and then to the default.

Example 43. lib.attrsets.getDev usage example

```
getDev pkgs.openssl  
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-1.0.1r-dev"
```

Located at [lib/attrsets.nix:1086](#) in <nixpkgs>.

lib.attrsets.getMan

Type: getMan :: Derivation -> String

Get a package's `man` output. If the output does not exist, fallback to `.out` and then to the default.

Example 44. lib.attrsets.getMan usage example

v: stable -

```
getMan pkgs.openssl  
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-1.0.1r-man"
```

Located at [lib/attrsets.nix:1099](#) in <nixpkgs>.

lib.attrsets.chooseDevOutputs

Type: chooseDevOutputs :: [Derivation] -> [String]

Pick the outputs of packages to place in buildInputs

drvs

List of packages to pick **dev** outputs from

Located at [lib/attrsets.nix:1106](#) in <nixpkgs>.

lib.attrsets.recurseIntoAttrs

Type: recurseIntoAttrs :: AttrSet -> AttrSet

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 45. lib.attrsets.recurseIntoAttrs usage example

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

v: stable -

```
};  
}
```

Located at [lib/attrsets.nix:1129](#) in <nixpkgs>.

lib.attrsets.dontRecurseIntoAttrs

Type: dontRecurseIntoAttrs :: AttrSet -> AttrSet

Undo the effect of recurseIntoAttrs.

attrs

An attribute set to not scan for derivations.

Located at [lib/attrsets.nix:1139](#) in <nixpkgs>.

lib.attrsets.unionOfDisjoint

Type: unionOfDisjoint :: AttrSet -> AttrSet -> AttrSet

`unionOfDisjoint x y` is equal to `x // y // z` where the attrnames in `z` are the intersection of the attrnames in `x` and `y`, and all values assert with an error message. This operator is commutative, unlike `(//)`.

x

Function argument

y

Function argument

Located at [lib/attrsets.nix:1151](#) in <nixpkgs>.

lib.strings: string manipulation functions

String manipulation functions.

v: stable -

lib.strings.concatStrings

Type: concatStrings :: [string] -> string

Concatenate a list of strings.

Example 46. lib.strings.concatStrings usage example

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

Located at [lib/strings.nix:50](#) in <nixpkgs>.

lib.strings.concatMapStrings

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

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

f

Function argument

list

Function argument

Example 47. lib.strings.concatMapStrings usage example

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

Located at [lib/strings.nix:60](#) in <nixpkgs>.

lib.strings.concatImapStrings

v: stable -

Type: 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 48. lib.strings.concatImapStrings usage example

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

Located at [lib/strings.nix:71](#) in <nixpkgs>.

lib.strings.intersperse

Type: intersperse :: a -> [a] -> [a]

Place an element between each element of a list

separator

Separator to add between elements

list

Input list

Example 49. lib.strings.intersperse usage example

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

Located at [lib/strings.nix:81](#) in <nixpkgs>.

lib.strings.concatStringsSep

Type: concatStringsSep :: string -> [string] -> string

Concatenate a list of strings with a separator between each element

Example 50. lib.strings.concatStringsSep usage example

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

Located at [lib/strings.nix:98](#) in <nixpkgs>.

lib.strings.concatMapStringsSep

Type: concatMapStringsSep :: string -> (a -> string) -> [a] -> 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 51. lib.strings.concatMapStringsSep usage example

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

Located at [lib/strings.nix:111](#) in <nixpkgs>.

lib.strings.concatIMapStringsSep

Type: concatIMapStringsSep :: string -> (int -> a -> string) -> [a] -> 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

v: stable -

list

List of input strings

Example 52. lib.strings.concatImapStringsSep usage example

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

Located at [lib/strings.nix:128](#) in <nixpkgs>.

lib.strings.concatLines

Type: concatLines :: [string] -> string

Concatenate a list of strings, adding a newline at the end of each one. Defined as concatMapStrings (s: s + "\n").

Example 53. lib.strings.concatLines usage example

```
concatLines [ "foo" "bar" ]
=> "foo\nbar\n"
```

Located at [lib/strings.nix:145](#) in <nixpkgs>.

lib.strings.replicate

Type: replicate :: int -> string -> string

Replicate a string n times, and concatenate the parts into a new string.

n

Function argument

s

v: stable -

Function argument

Example 54. `lib.strings.replicate` usage example

```
replicate 3 "v"  
=> "vvv"  
replicate 5 "hello"  
=> "hellohellohellohellohello"
```

Located at [lib/strings.nix:159](#) in `<nixpkgs>`.

lib.strings.makeSearchPath

Type: `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 55. `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:172](#) in `<nixpkgs>`.

v: stable -

lib.strings.makeSearchPathOutput

Type: 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 56. lib.strings.makeSearchPathOutput usage example

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

Located at [lib/strings.nix:190](#) in <nixpkgs>.

lib.strings.makeLibraryPath

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

Example 57. 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/
```

v: stable -

Located at [lib/strings.nix:208](#) in <nixpkgs>.

lib.strings.makeBinPath

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

Example 58. lib.strings.makeBinPath usage example

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

Located at [lib/strings.nix:217](#) in <nixpkgs>.

lib.strings.normalizePath

Type: normalizePath :: string -> string

Normalize path, removing extraneous /s

s

Function argument

Example 59. lib.strings.normalizePath usage example

```
normalizePath "/a//b///c/"
=> "/a/b/c/"
```

Located at [lib/strings.nix:227](#) in <nixpkgs>.

lib.strings.optionalString

Type: optionalString :: bool -> string -> string

v: stable -

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 60. lib.strings.optionalString usage example

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

Located at [lib/strings.nix:253](#) in <nixpkgs>.

lib.strings.hasPrefix

Type: hasPrefix :: string -> string -> bool

Determine whether a string has given prefix.

pref

Prefix to check for

str

Input string

Example 61. lib.strings.hasPrefix usage example

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

v: stable -

```
=> false
```

Located at [lib/strings.nix:269](#) in <nixpkgs>.

lib.strings.hasSuffix

Type: hasSuffix :: string -> string -> bool

Determine whether a string has given suffix.

suffix

Suffix to check for

content

Input string

Example 62. lib.strings.hasSuffix usage example

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

Located at [lib/strings.nix:296](#) in <nixpkgs>.

lib.strings.hasInfix

Type: hasInfix :: string -> string -> bool

Determine whether a string contains the given infix

infix

Function argument

content

v: stable -

s
Function argument

Example 63. `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:333](#) in <nixpkgs>.

lib.strings.stringToCharacters

Type: `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 64. `lib.strings.stringToCharacters` usage example

```
stringToCharacters ""  
=> [ ]  
stringToCharacters "abc"  
=> [ "a" "b" "c" ]  
stringToCharacters "🦄"  
=> [ "🦄" "🦄" "🦄" "🦄" ]
```

Located at [lib/strings.nix:363](#) in <nixpkgs>.

lib.strings.stringAsChars

Type: 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 65. lib.strings.stringAsChars usage example

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

Located at [lib/strings.nix:375](#) in <nixpkgs>.

lib.strings.toInt

Type: charToInt :: string -> int

v: stable -

Convert char to ascii value, must be in printable range

c

Function argument

Example 66. lib.strings.toInt usage example

```
charToInt "A"  
=> 65  
charToInt "("  
=> 40
```

Located at [lib/strings.nix:394](#) in <nixpkgs>.

lib.strings.escape

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

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

list

Function argument

Example 67. lib.strings.escape usage example

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

Located at [lib/strings.nix:405](#) in <nixpkgs>.

lib.strings.escapeC

Type: escapeC = [string] -> string -> string

v: stable -

Escape occurrence of the element of `list` in `string` by converting to its ASCII value and prefixing it with `\x`. Only works for printable ascii characters.

list

Function argument

Example 68. lib.strings.escapeC usage example

```
escapeC [" "] "foo bar"  
=> "foo\\x20bar"
```

Located at [lib/strings.nix:418](#) in <nixpkgs>.

lib.strings.escapeURL

Type: `escapeURL :: string -> string`

Escape the string so it can be safely placed inside a URL query.

Example 69. lib.strings.escapeURL usage example

```
escapeURL "foo/bar baz"  
=> "foo%2Fbar%20baz"
```

Located at [lib/strings.nix:429](#) in <nixpkgs>.

lib.strings.escapeShellArg

Type: `escapeShellArg :: string -> string`

Quote string to be used safely within the Bourne shell.

arg

Function argument

v: stable -

Example 70. `lib.strings.escapeShellArg` usage example

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

Located at [lib/strings.nix:443](#) in <nixpkgs>.

lib.strings.escapeShellArgs

Type: `escapeShellArgs :: [string] -> string`

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

Example 71. `lib.strings.escapeShellArgs` usage example

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

Located at [lib/strings.nix:453](#) in <nixpkgs>.

lib.strings.isValidPosixName

Type: `string -> bool`

Test whether the given name is a valid POSIX shell variable name.

`name`

Function argument

Example 72. `lib.strings.isValidPosixName` usage example

```
isValidPosixName "foo_bar000"  
=> true
```

v: stable -

```
isValidPosixName "0-bad.jpg"
=> false
```

Located at [lib/strings.nix:465](#) in <nixpkgs>.

lib.strings.toShellVar

Type: string -> (string | listOf string | attrsOf string) -> string

Translate a Nix value into a shell variable declaration, with proper escaping.

The value can be a string (mapped to a regular variable), a list of strings (mapped to a Bash-style array) or an attribute set of strings (mapped to a Bash-style associative array). Note that "string" includes string-coercible values like paths or derivations.

Strings are translated into POSIX sh-compatible code; lists and attribute sets assume a shell that understands Bash syntax (e.g. Bash or ZSH).

name

Function argument

value

Function argument

Example 73. lib.strings.toShellVar usage example

```
'''
${toShellVar "foo" "some string"}
[[ "$foo" == "some string" ]]
'''
```

Located at [lib/strings.nix:485](#) in <nixpkgs>.

lib.strings.toShellVars

v: stable -

Type: attrsOf (string | listOf string | attrsOf string) -> string

Translate an attribute set into corresponding shell variable declarations using `toShellVar`.

`vars`

Function argument

Example 74. `lib.strings.toShellVars` usage example

```
let
  foo = "value";
  bar = foo;
in ''
  ${toShellVars { inherit foo bar; }}
  [[ "$foo" == "$bar" ]]
''
```

Located at [lib/strings.nix:513](#) in <nixpkgs>.

lib.strings.escapeNixString

Type: string -> string

Turn a string into a Nix expression representing that string

`s`

Function argument

Example 75. `lib.strings.escapeNixString` usage example

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

Located at [lib/strings.nix:523](#) in <nixpkgs>.

v: stable -

lib.strings.escapeRegex

Type: string → string

Turn a string into an exact regular expression

Example 76. lib.strings.escapeRegex usage example

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

Located at [lib/strings.nix:533](#) in <nixpkgs>.

lib.strings.escapeNixIdentifier

Type: string → string

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

s

Function argument

Example 77. lib.strings.escapeNixIdentifier usage example

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

Located at [lib/strings.nix:545](#) in <nixpkgs>.

lib.strings.escapeXML

Type: string → string

v: stable -

Escapes a string such that it is safe to include verbatim in an XML document.

Example 78. `lib.strings.escapeXML` usage example

```
escapeXML '''test' 'test' < &gt;'  
=> "&quot;test&quot; &apos;test&apos; &lt; &amp;gt;"
```

Located at [lib/strings.nix:559](#) in <nixpkgs>.

lib.strings.toLowerCase

Type: `toLowerCase :: string -> string`

Converts an ASCII string to lower-case.

Example 79. `lib.strings.toLowerCase` usage example

```
toLowerCase "HOME"  
=> "home"
```

Located at [lib/strings.nix:578](#) in <nixpkgs>.

lib.strings.toUpperCase

Type: `toUpperCase :: string -> string`

Converts an ASCII string to upper-case.

Example 80. `lib.strings.toUpperCase` usage example

```
toUpperCase "home"  
=> "HOME"
```

v: stable -

Located at [lib/strings.nix:588](#) in <nixpkgs>.

lib.strings.addContextFrom

Appends string context from another string. This is an implementation detail of Nix and should be used carefully.

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 inputDrv and inputSrcs.

a

Function argument

b

Function argument

Example 81. lib.strings.addContextFrom usage example

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

Located at [lib/strings.nix:603](#) in <nixpkgs>.

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

v: stable -

Example 82. `lib.strings.splitString` usage example

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

Located at [lib/strings.nix:614](#) in <nixpkgs>.

lib.strings.removePrefix

Type: 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 83. `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:630](#) in <nixpkgs>.

lib.strings.removeSuffix

Type: string -> string -> string

v: stable -

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

suffix

Suffix to remove if it matches

str

Input string

Example 84. lib.strings.removeSuffix usage example

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

Located at [lib/strings.nix:663](#) in <nixpkgs>.

lib.strings.versionOlder

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

v1

Function argument

v2

Function argument

Example 85. lib.strings.versionOlder usage example

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

v: stable -

Located at [lib/strings.nix:694](#) in <nixpkgs>.

lib.strings.versionAtLeast

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

v1

Function argument

v2

Function argument

Example 86. 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:706](#) in <nixpkgs>.

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 87. lib.strings.getName usage example

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

v: stable -

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

Located at [lib/strings.nix:718](#) in <nixpkgs>.

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 88. 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:735](#) in <nixpkgs>.

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 89. lib.strings.nameFromURL usage example

v: stable -

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

Located at [lib/strings.nix:751](#) in <nixpkgs>.

lib.strings.cmakeOptionType

Type:

```
cmakeOptionType :: string -> string -> string -> string  
  
  @param feature The feature to be set  
  @param type The type of the feature to be set, as described in  
            https://cmake.org/cmake/help/latest/command/set.html  
            the possible values (case insensitive) are:  
            BOOL FILEPATH PATH STRING INTERNAL  
  @param value The desired value
```

Create a “-D<feature>:<type>=<value>” string that can be passed to typical CMake invocations.

type

Function argument

feature

Function argument

value

Function argument

Example 90. lib.strings.cmakeOptionType usage example

```
cmakeOptionType "string" "ENGINE" "sdl2"
```

v: stable -

```
=> "-DENGINE:STRING=sdl2"
```

Located at [lib/strings.nix:774](#) in <nixpkgs>.

lib.strings.cmakeBool

Type:

```
cmakeBool :: string -> bool -> string
```

@param condition The condition to be made `true` or `false`

@param flag The controlling flag of the condition

Create a `-D<condition>={TRUE,FALSE}` string that can be passed to typical CMake invocations.

condition

Function argument

flag

Function argument

Example 91. lib.strings.cmakeBool usage example

```
cmakeBool "ENABLE_STATIC_LIBS" false  
=> "-DENABLESTATIC_LIBS:BOOL=FALSE"
```

Located at [lib/strings.nix:793](#) in <nixpkgs>.

lib.strings.cmakeFeature

Type:

```
cmakeFeature :: string -> string -> string
```

v: stable -

```
@param condition The condition to be made true or false
@param flag The controlling flag of the condition
```

Create a `-D<feature>:STRING=<value>` string that can be passed to typical CMake invocations. This is the most typical usage, so it deserves a special case.

feature

Function argument

value

Function argument

Example 92. lib.strings.cmakeFeature usage example

```
cmakeFeature "MODULES" "badblock"
=> "-DMODULES:STRING=badblock"
```

Located at [lib/strings.nix:811](#) in `<nixpkgs>`.

lib.strings.mesonOption

Type:

```
mesonOption :: string -> string -> string
```

```
@param feature The feature to be set
@param value The desired value
```

Create a `-D<feature>=<value>` string that can be passed to typical Meson invocations.

feature

Function argument

value

v: stable -

Function argument

Example 93. **lib.strings.mesonOption** usage example

```
mesonOption "engine" "opengl"
=> "-Dengine=opengl"
```

Located at [lib/strings.nix:828](#) in <nixpkgs>.

lib.strings.mesonBool

Type:

```
mesonBool :: string -> bool -> string
```

@param condition The condition to be made `true` or `false`

@param flag The controlling flag of the condition

Create a `-D<condition>={true,false}` string that can be passed to typical Meson invocations.

condition

Function argument

flag

Function argument

Example 94. **lib.strings.mesonBool** usage example

```
mesonBool "hardened" true
=> "-Dhardened=true"
mesonBool "static" false
=> "-Dstatic=false"
```

v: stable -

Located at [lib/strings.nix:847](#) in <nixpkgs>.

lib.strings.mesonEnable

Type:

```
mesonEnable :: string -> bool -> string
```

@param feature The feature to be enabled or disabled

@param flag The controlling flag

Create a -D<feature>={enabled,disabled} string that can be passed to typical Meson invocations.

feature

Function argument

flag

Function argument

Example 95. lib.strings.mesonEnable usage example

```
mesonEnable "docs" true
=> "-Ddocs=enabled"
mesonEnable "savage" false
=> "-Dsavage=disabled"
```

Located at [lib/strings.nix:866](#) in <nixpkgs>.

lib.strings.enableFeature

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

flag

Function argument

v: stable -

feature

Function argument

Example 96. `lib.strings.enableFeature` usage example

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

Located at [lib/strings.nix:880](#) in `<nixpkgs>`.

lib.strings.enableFeatureAs

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

flag

Function argument

feature

Function argument

value

Function argument

Example 97. `lib.strings.enableFeatureAs` usage example

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

Located at [lib/strings.nix:894](#) in <nixpkgs>.

lib.strings.withFeature

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

flag

Function argument

feature

Function argument

Example 98. lib.strings.withFeature usage example

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

Located at [lib/strings.nix:906](#) in <nixpkgs>.

lib.strings.withFeatureAs

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

flag

Function argument

feature

Function argument

value

Function argument

v: stable -

Example 99. `lib.strings.withFeatureAs` usage example

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

Located at [lib/strings.nix:919](#) in <nixpkgs>.

lib.strings.fixedWidthString

Type: `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 100. `lib.strings.fixedWidthString` usage example

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

Located at [lib/strings.nix:934](#) in <nixpkgs>.

lib.strings.fixedWidthNumber

v: stable -

Format a number adding leading zeroes up to fixed width.

width

Function argument

n

Function argument

Example 101. lib.strings.fixedWidthNumber usage example

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

Located at [lib/strings.nix:951](#) in <nixpkgs>.

lib.strings.toFloatToString

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

float

Function argument

Example 102. lib.strings.toFloatToString 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:963](#) in <nixpkgs>.

lib.strings.isCoercibleToString

v: stable -

Soft-deprecated function. While the original implementation is available as `isConvertibleToString`, consider using `isStringLike` instead, if suitable.

Located at [lib/strings.nix:971](#) in `<nixpkgs>`.

lib.strings.isConvertibleToString

Check whether a list or other value can be passed to `toString`.

Many types of value are coercible to string this way, including `int`, `float`, `null`, `bool`, list of similarly coercible values.

`x`

Function argument

Located at [lib/strings.nix:980](#) in `<nixpkgs>`.

lib.strings.isStringLike

Check whether a value can be coerced to a string. The value must be a string, path, or attribute set.

String-like values can be used without explicit conversion in string interpolations and in most functions that expect a string.

`x`

Function argument

Located at [lib/strings.nix:991](#) in `<nixpkgs>`.

lib.strings.isStorePath

Check whether a value is a store path.

`x`

Function argument

Example 103. lib.strings.isStorePath usage example

v: stable -

```
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:1009](#) in <nixpkgs>.

lib.strings.toInt

Type: string -> int

Parse a string as an int. Does not support parsing of integers with preceding zero due to ambiguity between zero-padded and octal numbers. See toIntBase10.

str

Function argument

Example 104. lib.strings.toInt usage example

```
toInt "1337"
=> 1337

toInt "-4"
=> -4

toInt " 123 "
=> 123

toInt "00024"
=> error: Ambiguity in interpretation of 00024 between octal and zero padded
```

v: stable -

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

Located at [lib/strings.nix:1039](#) in <nixpkgs>.

lib.strings.toIntBase10

Type: string -> int

Parse a string as a base 10 int. This supports parsing of zero-padded integers.

str

Function argument

Example 105. lib.strings.toIntBase10 usage example

```
toIntBase10 "1337"
=> 1337

toIntBase10 "-4"
=> -4

toIntBase10 " 123 "
=> 123

toIntBase10 "00024"
=> 24

toIntBase10 "3.14"
=> error: floating point JSON numbers are not supported
```

Located at [lib/strings.nix:1090](#) in <nixpkgs>.

lib.strings.readPathsFromFile

v: stable -

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 106. `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:1133](#) in <nixpkgs>.

lib.strings.fileContents

Type: `fileContents :: path -> string`

Read the contents of a file removing the trailing `\n`

`file`

Function argument

Example 107. `lib.strings.fileContents` usage example

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

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

Located at [lib/strings.nix:1153](#) in <nixpkgs>.

v: stable -

lib.strings.sanitizeDerivationName

Type: sanitizeDerivationName :: String -> String

Creates a valid derivation name from a potentially invalid one.

Example 108. lib.strings.sanitizeDerivationName usage example

```
sanitizeDerivationName ".../hello.bar # foo"  
=> "-hello.bar-foo"  
sanitizeDerivationName ""  
=> "unknown"  
sanitizeDerivationName pkgs.hello  
=> "-nix-store-2g75chlpbxlrqn15zlb2dfh8hr9qwbk-hello-2.10"
```

Located at [lib/strings.nix:1168](#) in <nixpkgs>.

lib.strings.levenshtein

Type: levenshtein :: string -> string -> int

Computes the Levenshtein distance between two strings. Complexity $O(n*m)$ where n and m are the lengths of the strings. Algorithm adjusted from <https://stackoverflow.com/a/9750974/6605742>

a

Function argument

b

Function argument

Example 109. lib.strings.levenshtein usage example

```
levenshtein "foo" "foo"  
=> 0  
levenshtein "book" "hook"  
=> 1
```

v: stable -

```
levenshtein "hello" "Heyo"  
=> 3
```

Located at [lib/strings.nix:1207](#) in <nixpkgs>.

lib.strings.commonPrefixLength

Returns the length of the prefix common to both strings.

a

Function argument

b

Function argument

Located at [lib/strings.nix:1228](#) in <nixpkgs>.

lib.strings.commonSuffixLength

Returns the length of the suffix common to both strings.

a

Function argument

b

Function argument

Located at [lib/strings.nix:1236](#) in <nixpkgs>.

lib.strings.levenshteinAtMost

Type: levenshteinAtMost :: int -> string -> string -> bool

Returns whether the levenshtein distance between two strings is at most some value Complexity is $O(\min(n,m))$ for $k \leq 2$ and $O(n*m)$ otherwise

v: stable -

Example 110. `lib.strings.levenshteinAtMost` usage example

```
levenshteinAtMost 0 "foo" "foo"  
=> true  
levenshteinAtMost 1 "foo" "boa"  
=> false  
levenshteinAtMost 2 "foo" "boa"  
=> true  
levenshteinAtMost 2 "This is a sentence" "this is a sentense."  
=> false  
levenshteinAtMost 3 "This is a sentence" "this is a sentense."  
=> true
```

Located at [lib/strings.nix:1260](#) in <nixpkgs>.

lib.versions: version string functions

Version string functions.

lib.versions.splitVersion

Break a version string into its component parts.

Example 111. `lib.versions.splitVersion` usage example

```
splitVersion "1.2.3"  
=> ["1" "2" "3"]
```

Located at [lib/versions.nix:12](#) in <nixpkgs>.

lib.versions.major

Get the major version string from a string.

v: stable -

v

Function argument

Example 112. `lib.versions.major` usage example

```
major "1.2.3"  
=> "1"
```

Located at [lib/versions.nix:20](#) in <nixpkgs>.

lib.versions.minor

Get the minor version string from a string.

v

Function argument

Example 113. `lib.versions.minor` usage example

```
minor "1.2.3"  
=> "2"
```

Located at [lib/versions.nix:28](#) in <nixpkgs>.

lib.versions.patch

Get the patch version string from a string.

v

Function argument

Example 114. `lib.versions.patch` usage example

v: stable -

```
patch "1.2.3"  
=> "3"
```

Located at [lib/versions.nix:36](#) in <nixpkgs>.

lib.versions.majorMinor

Get string of the first two parts (major and minor) of a version string.

v

Function argument

Example 115. lib.versions.majorMinor usage example

```
majorMinor "1.2.3"  
=> "1.2"
```

Located at [lib/versions.nix:45](#) in <nixpkgs>.

lib.versions.pad

Pad a version string with zeros to match the given number of components.

n

Function argument

version

Function argument

Example 116. lib.versions.pad usage example

```
pad 3 "1.2"  
=> "1.2.0"
```

v: stable -

```
pad 3 "1.3-rc1"
=> "1.3.0-rc1"
pad 3 "1.2.3.4"
=> "1.2.3"
```

Located at [lib/versions.nix:59](#) in <nixpkgs>.

lib.trivial: miscellaneous functions

lib.trivial.id

Type: 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>.

lib.trivial.const

Type: 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 117. lib.trivial.const usage example

v: stable -

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

Located at [lib/trivial.nix:26](#) in <nixpkgs>.

lib.trivial.pipe

Type: 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 118. 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
```

v: stable -

```
(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>`.

lib.trivial.concat

Type: `concat :: [a] -> [a] -> [a]`

Concatenate two lists

x

Function argument

y

Function argument

Example 119. lib.trivial.concat usage example

```
concat [ 1 2 ] [ 3 4 ]  
=> [ 1 2 3 4 ]
```

Located at [lib/trivial.nix:80](#) in `<nixpkgs>`.

lib.trivial.or

boolean “or”

v: stable -

x

Function argument

y

Function argument

Located at [lib/trivial.nix:83](#) in <nixpkgs>.

lib.trivial.and

boolean “and”

x

Function argument

y

Function argument

Located at [lib/trivial.nix:86](#) in <nixpkgs>.

lib.trivial.bitAnd

bitwise “and”

Located at [lib/trivial.nix:89](#) in <nixpkgs>.

lib.trivial.bitOr

bitwise “or”

Located at [lib/trivial.nix:94](#) in <nixpkgs>.

lib.trivial.bitXor

bitwise “xor”

Located at [lib/trivial.nix:99](#) in <nixpkgs>.

v: stable -

lib.trivial.bitNot

bitwise “not”

Located at [lib/trivial.nix:104](#) in <nixpkgs>.

lib.trivial.boolToString

Type: 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>.

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 120. lib.trivial.mergeAttrs usage example

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

v: stable -

Located at [lib/trivial.nix:124](#) in <nixpkgs>.

lib.trivial.flip

Type: 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 121. lib.trivial.flip usage example

```
flip concat [1] [2]
=> [ 2 1 ]
```

Located at [lib/trivial.nix:138](#) in <nixpkgs>.

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 122. **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>.

lib.trivial.version

Returns the current full nixpkgs version number.

Located at [lib/trivial.nix:164](#) in <nixpkgs>.

lib.trivial.release

Returns the current nixpkgs release number as string.

Located at [lib/trivial.nix:167](#) in <nixpkgs>.

lib.trivial.oldestSupportedRelease

The latest release that is supported, at the time of release branch-off, if applicable.

Ideally, out-of-tree modules should be able to evaluate cleanly with all supported Nixpkgs versions (master, release and old release until EOL). So if possible, deprecation warnings should take effect only when all out-of-tree expressions/libs/modules can upgrade to the new way without losing support for supported Nixpkgs versions.

This release number allows deprecation warnings to be implemented such that they take effect as soon as the oldest release reaches end of life.

Located at [lib/trivial.nix:180](#) in <nixpkgs>.

v: stable -

lib.trivial.isInOldestRelease

Whether a feature is supported in all supported releases (at the time of release branch-off, if applicable). See `oldestSupportedRelease`.

release

Release number of feature introduction as an integer, e.g. 2111 for 21.11. Set it to the upcoming release, matching the nixpkgs/.version file.

Located at [lib/trivial.nix:186](#) in <nixpkgs>.

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:198](#) in <nixpkgs>.

lib.trivial.versionSuffix

Returns the current nixpkgs version suffix as string.

Located at [lib/trivial.nix:201](#) in <nixpkgs>.

lib.trivial.revisionWithDefault

Type: `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:212](#) in <nixpkgs>.

lib.trivial.inNixShell

v: stable -

Type: `inNixShell` :: bool

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

Located at [lib/trivial.nix:230](#) in <nixpkgs>.

lib.trivial.inPureEvalMode

Type: `inPureEvalMode` :: bool

Determine whether the function is being called from inside pure-eval mode by seeing whether `builtins` contains `currentSystem`. If not, we must be in pure-eval mode.

Located at [lib/trivial.nix:238](#) in <nixpkgs>.

lib.trivial.min

Return minimum of two numbers.

x

Function argument

y

Function argument

Located at [lib/trivial.nix:243](#) in <nixpkgs>.

lib.trivial.max

Return maximum of two numbers.

x

Function argument

y

Function argument

Located at [lib/trivial.nix:246](#) in <nixpkgs>.

v: stable -

lib.trivial.mod

Integer modulus

base

Function argument

int

Function argument

Example 123. lib.trivial.mod usage example

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

Located at [lib/trivial.nix:256](#) in <nixpkgs>.

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:267](#) in <nixpkgs>.

lib.trivial.splitByAndCompare

Type: (a -> bool) -> (a -> a -> int) -> (a -> a -> int) -> (a -> a v: stable -

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 124. `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:292](#) in <nixpkgs>.

lib.trivial.importJSON

Type: `importJSON :: path -> any`

v: stable -

Reads a JSON file.

path

Function argument

Located at [lib/trivial.nix:312](#) in <nixpkgs>.

lib.trivial.importTOML

Type: importTOML :: path -> any

Reads a TOML file.

path

Function argument

Located at [lib/trivial.nix:319](#) in <nixpkgs>.

lib.trivial.warn

Type: string -> a -> a

Print a warning before returning the second argument. This function behaves like `builtins.trace`, but requires a string message and formats it as a warning, including the `warning:` prefix.

To get a call stack trace and abort evaluation, set the environment variable

`NIX_ABORT_ON_WARN=true` and set the Nix options `--option pure-eval false --show-trace`

Located at [lib/trivial.nix:347](#) in <nixpkgs>.

lib.trivial.warnIf

Type: bool -> string -> a -> a

Like `warn`, but only warn when the first argument is `true`.

cond

v: stable -

Function argument

msg

Function argument

Located at [lib/trivial.nix:357](#) in <nixpkgs>.

lib.trivial.warnIfNot

Type: bool -> string -> a -> a

Like warnIf, but negated (warn if the first argument is `false`).

cond

Function argument

msg

Function argument

Located at [lib/trivial.nix:364](#) in <nixpkgs>.

lib.trivial.throwIfNot

Type: bool -> string -> a -> a

Like the `assert b; e` expression, but with a custom error message and without the semicolon.

If true, return the identity function, `r: r`.

If false, throw the error message.

Calls can be juxtaposed using function application, as `(r: r) a = a`, so `(r: r) (r: r) a = a`, and so forth.

cond

Function argument

msg

Function argument

v: stable -

Example 125. lib.trivial.throwIfNot usage example

```
throwIfNot (lib.isList overlays) "The overlays argument to nixpkgs must be  
lib.foldr (x: throwIfNot (lib.isFunction x) "All overlays passed to nixpkgs  
pkgs
```

Located at [lib/trivial.nix:386](#) in <nixpkgs>.

lib.trivial.throwIf

Type: bool -> string -> a -> a

Like throwIfNot, but negated (throw if the first argument is `true`).

`cond`

Function argument

`msg`

Function argument

Located at [lib/trivial.nix:393](#) in <nixpkgs>.

lib.trivial.checkListOfEnum

Type: String -> List ComparableVal -> List ComparableVal -> a -> a

Check if the elements in a list are valid values from a enum, returning the identity function, or throwing an error message otherwise.

`msg`

Function argument

`valid`

Function argument

`given`

v: stable -

Function argument

Example 126. `lib.trivial.checkListOfEnum` usage example

```
let colorVariants = ["bright" "dark" "black"]
in checkListOfEnum "color variants" [ "standard" "light" "dark" ] colorVariants
=>
error: color variants: bright, black unexpected; valid ones: standard, light
```

Located at [lib/trivial.nix:405](#) in <nixpkgs>.

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:428](#) in <nixpkgs>.

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

v: stable -

Function argument

Located at [lib/trivial.nix:440](#) in <nixpkgs>.

lib.trivial.isFunction

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

f

Function argument

Located at [lib/trivial.nix:448](#) in <nixpkgs>.

lib.trivial.mirrorFunctionArgs

Type: `mirrorFunctionArgs :: (a -> b) -> (a -> c) -> (a -> c)`

`mirrorFunctionArgs f g` creates a new function `g'` with the same behavior as `g` (`g' x == g x`) but its function arguments mirroring `f` (`lib.functionArgs g' == lib.functionArgs f`).

f

Function to provide the argument metadata

Example 127. lib.trivial.mirrorFunctionArgs usage example

```
addab = {a, b}: a + b
addab { a = 2; b = 4; }
=> 6
lib.functionArgs addab
=> { a = false; b = false; }
addab1 = attrs: addab attrs + 1
addab1 { a = 2; b = 4; }
=> 7
lib.functionArgs addab1
=> {}
addab1' = lib.mirrorFunctionArgs addab addab1
addab1' { a = 2; b = 4; }
```

v: stable -

```
=> 7
lib.functionArgs addab1'
=> { a = false; b = false; }
```

Located at [lib/trivial.nix:475](#) in <nixpkgs>.

lib.trivial.fromFunction

Turns any non-callable values into constant functions. Returns callable values as is.

v

Any value

Example 128. lib.fromFunction usage example

```
nix-repl> lib.fromFunction 1 2
1

nix-repl> lib.fromFunction (x: x + 1) 2
3
```

Located at [lib/trivial.nix:497](#) in <nixpkgs>.

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

v: stable -

Function argument

Located at [lib/trivial.nix:513](#) in <nixpkgs>.

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:539](#) in <nixpkgs>.

lib.fixedPoints: explicit recursion functions

lib.fixedPoints.fix

Type: `fix :: (a -> a) -> a`

`fix f` computes the fixed point of the given function `f`. In other words, the return value is `x` in `x = f x`.

`f` must be a lazy function. This means that `x` must be a value that can be partially evaluated, such as an attribute set, a list, or a function. This way, `f` can use one part of `x` to compute another part.

Relation to syntactic recursion

This section explains `fix` by refactoring from syntactic recursion to a call of `fix` instead.

v: stable -

For context, Nix lets you define attributes in terms of other attributes syntactically using the [rec { }](#) syntax.

```
nix-repl> rec {
  foo = "foo";
  bar = "bar";
  foobar = foo + bar;
}
{ bar = "bar"; foo = "foo"; foobar = "foobar"; }
```

This is convenient when constructing a value to pass to a function for example, but an equivalent effect can be achieved with the `let` binding syntax:

```
nix-repl> let self = {
  foo = "foo";
  bar = "bar";
  foobar = self.foo + self.bar;
}; in self
{ bar = "bar"; foo = "foo"; foobar = "foobar"; }
```

But in general you can get more reuse out of `let` bindings by refactoring them to a function.

```
nix-repl> f = self: {
  foo = "foo";
  bar = "bar";
  foobar = self.foo + self.bar;
}
```

This is where `fix` comes in, it contains the syntactic recursion that's not in `f` anymore.

```
nix-repl> fix = f:
  let self = f self; in self;
```

By applying `fix` we get the final result.

v: stable -

```
nix-repl> fix f
{ bar = "bar"; foo = "foo"; foobar = "foobar"; }
```

Such a refactored `f` using `fix` is not useful by itself. See [extends](#) for an example use case. There `self` is also often called `final`.

`f`

Function argument

Example 129. `lib.fixedPoints.fix` usage example

```
fix (self: { foo = "foo"; bar = "bar"; foobar = self.foo + self.bar; })
=> { bar = "bar"; foo = "foo"; foobar = "foobar"; }

fix (self: [ 1 2 (elemAt self 0 + elemAt self 1) ])
=> [ 1 2 3 ]
```

Located at [lib/fixed-points.nix:75](#) in `<nixpkgs>`.

`lib.fixedPoints.fix'`

A variant of `fix` that records the original recursive attribute set in the result, in an attribute named `__unfix__`.

This is useful in combination with the `extends` function to implement deep overriding.

`f`

Function argument

Located at [lib/fixed-points.nix:84](#) in `<nixpkgs>`.

`lib.fixedPoints.converge`

Type: `(a -> a) -> a -> a`

v: stable -

Return the fixpoint that `f` converges to when called iteratively, starting with the input `x`.

```
nix-repl> converge (x: x / 2) 16
0
```

`f`

Function argument

`x`

Function argument

Located at [lib/fixed-points.nix:97](#) in <nixpkgs>.

lib.fixedPoints.extends

Modify the contents of an explicitly recursive attribute set in a way that honors `self`-references. This is accomplished with a function

```
g = self: super: { foo = super.foo + " + "; }
```

that has access to the unmodified input (`super`) as well as the final non-recursive representation of the attribute set (`self`). `extends` differs from the native `//` operator insofar as that it's applied *before* references to `self` are resolved:

```
nix-repl> fix (extends g f)
{ bar = "bar"; foo = "foo + "; foobar = "foo + bar"; }
```

The name of the function is inspired by object-oriented inheritance, i.e. think of it as an infix operator `g extends f` that mimics the syntax from Java. It may seem counter-intuitive to have the “base class” as the second argument, but it's nice this way if several uses of `extends` are cascaded.

To get a better understanding how `extends` turns a function with a fix point (the package set we start with) into a new function with a different fix point (the desired packages set) lets just see, how `extends g f` unfolds with `g` and `f` defined above:

v: stable -

```
extends g f = self: let super = f self; in super // g self super;  
= self: let super = { foo = "foo"; bar = "bar"; foobar = self  
= self: { foo = "foo"; bar = "bar"; foobar = self.foo + self.  
= self: { foo = "foo"; bar = "bar"; foobar = self.foo + self.  
= self: { foo = "foo + "; bar = "bar"; foobar = self.foo + se
```

f

Function argument

rattrs

Function argument

self

Function argument

Located at [lib/fixed-points.nix:141](#) in <nixpkgs>.

lib.fixedPoints.composeExtensions

Compose two extending functions of the type expected by ‘extends’ into one where changes made in the first are available in the ‘super’ of the second

f

Function argument

g

Function argument

final

Function argument

prev

Function argument

Located at [lib/fixed-points.nix:148](#) in <nixpkgs>.

lib.fixedPoints.composeManyExtensions

v: stable -

Compose several extending functions of the type expected by ‘extends’ into one where changes made in preceding functions are made available to subsequent ones.

```
composeManyExtensions : [packageSet -> packageSet -> packageSet] -> packageSet
    ^final          ^prev           ^overrides      ^final
```

Located at [lib/fixed-points.nix:164](#) in <nixpkgs>.

lib.fixedPoints.makeExtensible

Create an overridable, recursive attribute set. For example:

```
nix-repl> obj = makeExtensible (self: { })
nix-repl> obj
{ __unfix__ = «lambda»; extend = «lambda»; }

nix-repl> obj = obj.extend (self: super: { foo = "foo"; })

nix-repl> obj
{ __unfix__ = «lambda»; extend = «lambda»; foo = "foo"; }

nix-repl> obj = obj.extend (self: super: { foo = super.foo + " + "; bar =
nix-repl> obj
{ __unfix__ = «lambda»; bar = "bar"; extend = «lambda»; foo = "foo + "; fo
```

Located at [lib/fixed-points.nix:187](#) in <nixpkgs>.

lib.fixedPoints.makeExtensibleWithCustomName

Same as `makeExtensible` but the name of the extending attribute is customized.

`extenderName`

Function argument

v: stable -

rattr

Function argument

Located at [lib/fixed-points.nix:193](#) in <nixpkgs>.

lib.lists: list manipulation functions

General list operations.

lib.lists.singleton

Type: 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 130. lib.lists.singleton usage example

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

Located at [lib/lists.nix:23](#) in <nixpkgs>.

lib.lists.forEach

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

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

xs

Function argument

f

v: stable -

Function argument

Example 131. `lib.lists.forEach` usage example

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

Located at [lib/lists.nix:36](#) in `<nixpkgs>`.

lib.lists.foldr

Type: `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 132. `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 ]
```

v: stable -

```
=> "2345a"
```

Located at [lib/lists.nix:53](#) in <nixpkgs>.

lib.lists.fold

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

Located at [lib/lists.nix:64](#) in <nixpkgs>.

lib.lists.foldl

Type: `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 133. 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:81](#) in `<nixpkgs>`.

lib.lists.foldl'

Type: `foldl' :: (acc -> x -> acc) -> acc -> [x] -> acc`

Reduce a list by applying a binary operator from left to right, starting with an initial accumulator.

Before each application of the operator, the accumulator value is evaluated. This behavior makes this function stricter than [foldl](#).

v: stable -

Unlike [`builtins.foldl'`](#), the initial accumulator argument is evaluated before the first iteration.

A call like

```
foldl' op acc0 [ x0 x1 x2 ... xn-1 xn ]
```

is (denotationally) equivalent to the following, but with the added benefit that `foldl'` itself will never overflow the stack.

```
let
  acc1 = builtins.seq acc0 (op acc0 x0);
  acc2 = builtins.seq acc1 (op acc1 x1);
  acc3 = builtins.seq acc2 (op acc2 x2);
  ...
  accn = builtins.seq accn-1 (op accn-1 xn-1);
  accn+1 = builtins.seq accn (op accn xn);
in
accn+1

# Or ignoring builtins.seq
op (op (... (op (op (op acc0 x0) x1) x2) ...) xn-1) xn
```

op

The binary operation to run, where the two arguments are:

1. acc: The current accumulator value: Either the initial one for the first iteration, or the result of the previous iteration
2. x: The corresponding list element for this iteration

acc

The initial accumulator value

list

The list to fold

Example 134. `lib.lists.foldl'` usage example

v: stable -

```
foldl' (acc: x: acc + x) 0 [1 2 3]
=> 6
```

Located at [lib/lists.nix:129](#) in <nixpkgs>.

lib.lists imap0

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

Map with index starting from 0

f

Function argument

list

Function argument

Example 135. lib.lists imap0 usage example

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

Located at [lib/lists.nix:155](#) in <nixpkgs>.

lib.lists imap1

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

Map with index starting from 1

f

Function argument

list

v: stable -

Function argument

Example 136. lib.lists imap1 usage example

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

Located at [lib/lists.nix:165](#) in <nixpkgs>.

lib.lists.concatMap

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

Map and concatenate the result.

Example 137. lib.lists.concatMap usage example

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

Located at [lib/lists.nix:175](#) in <nixpkgs>.

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 138. lib.lists.flatten usage example

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

v: stable -

```
flatten 1  
=> [1]
```

Located at [lib/lists.nix:186](#) in <nixpkgs>.

lib.lists.remove

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

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

`e`

Element to remove from the list

Example 139. lib.lists.remove usage example

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

Located at [lib/lists.nix:199](#) in <nixpkgs>.

lib.lists.findSingle

Type: `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.

v: stable -

multiple

Default value to return if more than one element was found

list

Input list

Example 140. 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:217](#) in <nixpkgs>.

lib.lists.findIndex

Type: `findFirstIndex :: (a -> Bool) -> b -> [a] -> (Int | b)`

Find the first index 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 141. lib.lists.findIndex usage example

v: stable -

```
findFirstIndex (x: x > 3) null [ 0 6 4 ]
=> 1
findFirstIndex (x: x > 9) null [ 0 6 4 ]
=> null
```

Located at [lib/lists.nix:242](#) in <nixpkgs>.

lib.lists.findFirst

Type: `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 142. 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:293](#) in <nixpkgs>.

lib.lists.any

v: stable -

Type: any :: ($a \rightarrow \text{bool}$) $\rightarrow [a] \rightarrow \text{bool}$

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

Example 143. lib.lists.any usage example

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

Located at [lib/lists.nix:319](#) in <nixpkgs>.

lib.lists.all

Type: all :: ($a \rightarrow \text{bool}$) $\rightarrow [a] \rightarrow \text{bool}$

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

Example 144. 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:332](#) in <nixpkgs>.

lib.lists.count

Type: count :: ($a \rightarrow \text{bool}$) $\rightarrow [a] \rightarrow \text{int}$

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

v: stable -

pred

Predicate

Example 145. lib.lists.count usage example

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

Located at [lib/lists.nix:343](#) in <nixpkgs>.

lib.lists.optional

Type: 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 146. lib.lists.optional usage example

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

Located at [lib/lists.nix:359](#) in <nixpkgs>.

lib.lists.optionals

v: stable -

Type: `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 147. lib.lists.optionals usage example

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

Located at [lib/lists.nix:371](#) in <nixpkgs>.

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 148. lib.lists.toList usage example

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

Located at [lib/lists.nix:388](#) in <nixpkgs>.

lib.lists.range

Type: 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 149. lib.lists.range usage example

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

Located at [lib/lists.nix:400](#) in <nixpkgs>.

lib.lists.replicate

Type: replicate :: int -> a -> [a]

Return a list with `n` copies of an element.

`n`

Function argument

`elem`

Function argument

Example 150. lib.lists.replicate usage example

v: stable -

```
replicate 3 "a"  
=> [ "a" "a" "a" ]  
replicate 2 true  
=> [ true true ]
```

Located at [lib/lists.nix:420](#) in <nixpkgs>.

lib.lists.partition

Type: (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 151. 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:431](#) in <nixpkgs>.

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`

v: stable -

Function argument

`lst`

Function argument

Example 152. `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 = "icewmbg"; script = "icewmbg &";}
                     {name = "mate";   script = "gnome-session &";}
]
=> { icewm = [ { name = "icewm"; script = "icewm &"; }
        { name = "icewmbg"; 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:460](#) in `<nixpkgs>`.

lib.lists.zipListsWith

Type: `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`

v: stable -

First list

 snd

 Second list

Example 153. `lib.lists.zipListsWith` usage example

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

Located at [lib/lists.nix:480](#) in <nixpkgs>.

lib.lists.zipLists

Type: `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 154. `lib.lists.zipLists` usage example

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

Located at [lib/lists.nix:499](#) in <nixpkgs>.

lib.lists.reverseList

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

Reverse the order of the elements of a list.

xs

Function argument

v: stable -

Example 155. `lib.lists.reverseList` usage example

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

Located at [lib/lists.nix:510](#) in `<nixpkgs>`.

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 156. 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
      rest    = [ "/home" "other" ]; # everything else
```

Located at [lib/lists.nix:532](#) in `<nixpkgs>`.

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.

v: stable -

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

`before`

Function argument

`list`

Function argument

Example 157. `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:571](#) in `<nixpkgs>`.

lib.lists.sort

Type: `sort :: (a -> a -> Bool) -> [a] -> [a]`

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.

See also [sortOn](#), which applies the default comparison on a function-derived property, and may be more efficient.

Example 158. lib.lists.sort usage example

```
sort (p: q: p < q) [ 5 3 7 ]  
=> [ 3 5 7 ]
```

Located at [lib/lists.nix:605](#) in `<nixpkgs>`.

lib.lists.sortOn

Type: `sortOn :: (a -> b) -> [a] -> [a]`, for comparable b

Sort a list based on the default comparison of a derived property b.

The items are returned in b-increasing order.

Performance:

The passed function f is only evaluated once per item, unlike an unprepared [sort](#) using `f p < f q`.

Laws:

```
sortOn f == sort (p: q: f p < f q)
```

f

Function argument

list

Function argument

Example 159. lib.lists.sortOn usage example

```
sortOn stringLength [ "aa" "b" "cccc" ]  
=> [ "b" "aa" "cccc" ]
```

v: stable -

Located at [lib/lists.nix:645](#) in <nixpkgs>.

lib.lists.compareLists

Compare two lists element-by-element.

cmp

Function argument

a

Function argument

b

Function argument

Example 160. 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:670](#) in <nixpkgs>.

lib.lists.naturalSort

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

lst

Function argument

v: stable -

Example 161. 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:693](#) in <nixpkgs>.

lib.lists.take

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

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

count

Number of elements to take

Example 162. lib.lists.take usage example

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

Located at [lib/lists.nix:711](#) in <nixpkgs>.

lib.lists.drop

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

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

v: stable -

count

Number of elements to drop

list

Input list

Example 163. lib.lists.drop usage example

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

Located at [lib/lists.nix:725](#) in <nixpkgs>.

lib.lists.hasPrefix

Type: hasPrefix :: [a] -> [a] -> bool

Whether the first list is a prefix of the second list.

list1

Function argument

list2

Function argument

Example 164. lib.lists.hasPrefix usage example

```
hasPrefix [ 1 2 ] [ 1 2 3 4 ]
=> true
hasPrefix [ 0 1 ] [ 1 2 3 4 ]
=> false
```

Located at [lib/lists.nix:741](#) in <nixpkgs>.

lib.lists.removePrefix

Type: `removePrefix :: [a] -> [a] -> [a]`

Remove the first list as a prefix from the second list. Error if the first list isn't a prefix of the second list.

`list1`

Function argument

`list2`

Function argument

Example 165. lib.lists.removePrefix usage example

```
removePrefix [ 1 2 ] [ 1 2 3 4 ]
=> [ 3 4 ]
removePrefix [ 0 1 ] [ 1 2 3 4 ]
=> <error>
```

Located at [lib/lists.nix:757](#) in <nixpkgs>.

lib.lists.sublist

Type: `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`

v: stable -

Input list

Example 166. `lib.lists.sublist` usage example

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

Located at [lib/lists.nix:776](#) in <nixpkgs>.

lib.lists.commonPrefix

Type: `commonPrefix :: [a] -> [a] -> [a]`

The common prefix of two lists.

list1

Function argument

list2

Function argument

Example 167. `lib.lists.commonPrefix` usage example

```
commonPrefix [ 1 2 3 4 5 6 ] [ 1 2 4 8 ]
=> [ 1 2 ]
commonPrefix [ 1 2 3 ] [ 1 2 3 4 5 ]
=> [ 1 2 3 ]
commonPrefix [ 1 2 3 ] [ 4 5 6 ]
=> [ ]
```

Located at [lib/lists.nix:802](#) in <nixpkgs>.

v: stable -

lib.lists.last

Type: last :: [a] -> a

Return the last element of a list.

This function throws an error if the list is empty.

list

Function argument

Example 168. lib.lists.last usage example

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

Located at [lib/lists.nix:826](#) in <nixpkgs>.

lib.lists.init

Type: init :: [a] -> [a]

Return all elements but the last.

This function throws an error if the list is empty.

list

Function argument

Example 169. lib.lists.init usage example

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

Located at [lib/lists.nix:840](#) in <nixpkgs>.

v: stable -

lib.lists.crossLists

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

Example 170. 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:851](#) in <nixpkgs>.

lib.lists.unique

Type: unique :: [a] -> [a]

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

Example 171. lib.lists.unique usage example

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

Located at [lib/lists.nix:864](#) in <nixpkgs>.

lib.lists.allUnique

Type: allUnique :: [a] -> bool

Check if list contains only unique elements. O(n^2) complexity.

list

Function argument

Example 172. lib.lists.allUnique usage example

v: stable -

```
allUnique [ 3 2 3 4 ]
=> false
allUnique [ 3 2 4 1 ]
=> true
```

Located at [lib/lists.nix:876](#) in <nixpkgs>.

lib.lists.intersectLists

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

e

Function argument

Example 173. lib.lists.intersectLists usage example

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

Located at [lib/lists.nix:885](#) in <nixpkgs>.

lib.lists.subtractLists

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

e

Function argument

Example 174. lib.lists.subtractLists usage example

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

Located at [lib/lists.nix:893](#) in <nixpkgs>.

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:898](#) in <nixpkgs>.

lib.debug: debugging functions

Collection of functions useful for debugging broken nix expressions.

- **trace**-like functions take two values, print the first to stderr and return the second.
- **traceVal**-like functions take one argument which both printed and returned.
- **traceSeq**-like functions fully evaluate their traced value before printing (not just to "wea v: stable -

normal form” like trace does by default).

- Functions that end in `-Fn` take an additional function as their first argument, which is applied to the traced value before it is printed.

lib.debug.traceIf

Type: `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 175. lib.debug.traceIf usage example

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

Located at [lib/debug.nix:44](#) in `<nixpkgs>`.

lib.debug.traceValFn

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

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

`f`

Function to apply

v: stable -

x

Value to trace and return

Example 176. `lib.debug.traceValFn` usage example

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

Located at [lib/debug.nix:62](#) in <nixpkgs>.

lib.debug.traceVal

Type: `traceVal :: a -> a`

Trace the supplied value and return it.

Example 177. `lib.debug.traceVal` usage example

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

Located at [lib/debug.nix:77](#) in <nixpkgs>.

lib.debug.traceSeq

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

`builtins.trace`, but the value is `builtins.deepSeqed` first.

x

The value to trace

v: stable -

y

The value to return

Example 178. `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:91](#) in <nixpkgs>.

lib.debug.traceSeqN

Type: `traceSeqN :: Int -> a -> b -> b`

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 179. lib.debug.traceSeqN usage example

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

Located at [lib/debug.nix:108](#) in <nixpkgs>.

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:125](#) in <nixpkgs>.

v: stable -

lib.debug.traceValSeq

A combination of `traceVal` and `traceSeq`.

Located at [lib/debug.nix:132](#) in <nixpkgs>.

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:136](#) in <nixpkgs>.

lib.debug.traceValSeqN

A combination of `traceVal` and `traceSeqN`.

Located at [lib/debug.nix:144](#) in <nixpkgs>.

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`

v: stable -

Function argument

f

Function argument

v

Function argument

Example 180. `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:157](#) in <nixpkgs>.

lib.debug.runTests

Type:

```
runTests :: {
  tests = [ String ];
  ${testName} :: {
    expr :: a;
    expected :: a;
  };
}
->
[
  {
    name :: String;
    expected :: a;
    result :: a;
  }
]
```

v: stable -

Evaluates 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, result}`,

- expected
 - What was passed as `expected`
- result
 - The actual `result` of the test

Used for regression testing of the functions in lib; see tests.nix for more examples.

Important: Only attributes that start with `test` are executed.

- If you want to run only a subset of the tests add the attribute `tests = ["testName"];`

tests

Tests to run

Example 181. `lib.debug.runTests` usage example

```
runTests {  
    testAndOk = {  
        expr = lib.and true false;  
        expected = false;  
    };  
    testAndFail = {  
        expr = lib.and true false;  
        expected = true;  
    };  
}  
->  
[  
    {  
        name = "testAndFail";  
    }  
]
```

v: stable -

```
    expected = true;
    result = false;
}
]
```

Located at [lib/debug.nix:229](#) in <nixpkgs>.

lib.debug.testAllTrue

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

`expr`

Function argument

Example 182. lib.debug.testAllTrue usage example

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

Located at [lib/debug.nix:245](#) in <nixpkgs>.

lib.options: NixOS / nixpkgs option handling

Nixpkgs/NixOS option handling.

lib.options.isOption

Type: `isOption :: a -> bool`

Returns true when the given argument is an option

Example 183. lib.options.isOption usage example

```
isOption 1          // => false
```

v: stable -

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

Located at [lib/options.nix:56](#) in <nixpkgs>.

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.

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.

v: stable -

visible

Whether the option shows up in the manual. Default: true. Use false to hide the option and any sub-options from submodules. Use “shallow” to hide only sub-options.

readOnly

Whether the option can be set only once

Example 184. `lib.options.mkOption` usage example

```
mkOption { } // => { _type = "option"; }
mkOption { default = "foo"; } // => { _type = "option"; default = "foo"; }
```

Located at [lib/options.nix:66](#) in <nixpkgs>.

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 185. `lib.options.mkEnableOption` usage example

```
mkEnableOption "foo"
=> { _type = "option"; default = false; description = "Whether to enable ·
```

Located at [lib/options.nix:98](#) in <nixpkgs>.

lib.options.mkPackageOption

Type: `mkPackageOption :: pkgs -> (string|[string]) -> { nullable? :: bool, default? :: string|[string], example? :: null|string|[string], extraDescription? :: string, pkgsText? :: string } -> option`

v: stable -

Creates an Option attribute set for an option that specifies the package a module should use for some purpose.

The package is specified in the third argument under `default` as a list of strings representing its attribute path in nixpkgs (or another package set). Because of this, you need to pass nixpkgs itself (usually `pkgs` in a module; alternatively to nixpkgs itself, another package set) as the first argument.

If you pass another package set you should set the `pkgsText` option. This option is used to display the expression for the package set. It is "`pkgs`" by default. If your expression is complex you should parenthesize it, as the `pkgsText` argument is usually immediately followed by an attribute lookup `(.)`.

The second argument may be either a string or a list of strings. It provides the display name of the package in the description of the generated option (using only the last element if the passed value is a list) and serves as the fallback value for the `default` argument.

To include extra information in the description, pass `extraDescription` to append arbitrary text to the generated description.

You can also pass an `example` value, either a literal string or an attribute path.

The `default` argument can be omitted if the provided name is an attribute of `pkgs` (if `name` is a string) or a valid attribute path in `pkgs` (if `name` is a list). You can also set `default` to just a string in which case it is interpreted as an attribute name (a singleton attribute path, if you will).

If you wish to explicitly provide no default, pass `null` as `default`.

If you want users to be able to set no package, pass `nullable = true`. In this mode a `default = null` will not be interpreted as no default and is interpreted literally.

pkgs

Package set (an instantiation of nixpkgs such as `pkgs` in modules or another package set)

name

Name for the package, shown in option description

structured function argument

nullable

v: stable -

Whether the package can be null, for example to disable installing a package altogether (defaults to false)

default

The attribute path where the default package is located (may be omitted, in which case it is copied from name)

example

A string or an attribute path to use as an example (may be omitted)

extraDescription

Additional text to include in the option description (may be omitted)

pkgsText

Representation of the package set passed as pkgs (defaults to "pkgs")

Example 186. `lib.options.mkPackageOption` usage example

```
mkPackageOption pkgs "hello" { }
=> { ...; default = pkgs.hello; defaultText = literalExpression "pkgs.hel"
}

mkPackageOption pkgs "GHC" {
  default = [ "ghc" ];
  example = "pkgs.haskell.packages.ghc92.ghc.withPackages (hks: [ hks.p
}
=> { ...; default = pkgs.ghc; defaultText = literalExpression "pkgs.ghc";
}

mkPackageOption pkgs [ "python3Packages" "pytorch" ] {
  extraDescription = "This is an example and doesn't actually do anything
}
=> { ...; default = pkgs.python3Packages.pytorch; defaultText = literalEx

mkPackageOption pkgs "nushell" {
  nullable = true;
}
```

v: stable -

```
}

=> { ...; default = pkgs.nushell; defaultText = literalExpression "pkgs.nu

mkPackageOption pkgs "coreutils" {
  default = null;
}
=> { ...; description = "The coreutils package to use."; type = package;

mkPackageOption pkgs "dbus" {
  nullable = true;
  default = null;
}
=> { ...; default = null; description = "The dbus package to use."; type = package;

mkPackageOption pkgs.javaPackages "OpenJFX" {
  default = "openjfx20";
  pkgsText = "pkgs.javaPackages";
}
=> { ...; default = pkgs.javaPackages.openjfx20; defaultText = literalExp
```

Located at [lib/options.nix:185](#) in <nixpkgs>.

lib.options.mkPackageOptionMD

Alias of mkPackageOption. Previously used to create options with markdown documentation, which is no longer required.

Located at [lib/options.nix:226](#) in <nixpkgs>.

lib.options.mkSinkUndeclaredOptions

This option accepts anything, but it does not produce any result.

v: stable -

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:233](#) in <nixpkgs>.

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:266](#) in <nixpkgs>.

lib.options.getValues

Type: getValues :: [{ value :: a; }] -> [a]

Extracts values of all “value” keys of the given list.

Example 187. lib.options.getValues usage example

```
getValues [ { value = 1; } { value = 2; } ] // => [ 1 2 ]
getValues [ ]                                // => [ ]
```

Located at [lib/options.nix:286](#) in <nixpkgs>.

lib.options.GetFiles

v: stable -

Type: `getFiles :: [{ file :: a; }] -> [a]`

Extracts values of all “file” keys of the given list

Example 188. `lib.options.getFiles` usage example

```
getFiles [ { file = "file1"; } { file = "file2"; } ] // => [ "file1" "file2"  
getFiles [ ] // => [ ]
```

Located at [lib/options.nix:296](#) in <nixpkgs>.

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.

This function was made obsolete by `renderOptionValue` and is kept for compatibility with out-of-tree code.

`x`

Function argument

Located at [lib/options.nix:354](#) in <nixpkgs>.

lib.options.renderOptionValue

Ensures that the given option value (default or example) is a `_typed` string by rendering Nix values to `literalExpressions`.

`v`

Function argument

Located at [lib/options.nix:365](#) in <nixpkgs>.

v: stable -

lib.options.literalExpression

For use in the `defaultText` and `example` option attributes. Causes the given string to be rendered verbatim in the documentation as Nix code. This is necessary for complex values, e.g. functions, or values that depend on other values or packages.

text

Function argument

Located at [lib/options.nix:378](#) in <nixpkgs>.

lib.options.mdDoc

Transition marker for documentation that's already migrated to markdown syntax. This is a no-op and no longer needed.

Located at [lib/options.nix:387](#) in <nixpkgs>.

lib.options.literalMD

For use in the `defaultText` and `example` option attributes. Causes the given MD text to be inserted verbatim in the documentation, for when a `literalExpression` would be too hard to read.

text

Function argument

Located at [lib/options.nix:393](#) in <nixpkgs>.

lib.options.showOption

Convert an option, described as a list of the option parts to a human-readable version.

parts

Function argument

Example 189. lib.options.showOption usage example

v: stable -

```
(showOption ["foo" "bar" "baz"]) == "foo.bar.baz"  
(showOption ["foo" "bar.baz" "tux"]) == "foo.\\"bar.baz\\\".tux"  
(showOption ["windowManager" "2bwm" "enable"]) == "windowManager.\\"2bwm"
```

Placeholders will not be quoted as they are not actual values:

```
(showOption ["foo" "*" "bar"]) == "foo.*.bar"  
(showOption ["foo" "<name>" "bar"]) == "foo.<name>.bar"
```

Located at [lib/options.nix:411](#) in <nixpkgs>.

lib.path: path functions

Functions for working with path values.

lib.path.append

Type: append :: Path -> String -> Path

Append a subpath string to a path.

Like `path + ("/" + string)` but safer, because it errors instead of returning potentially surprising results. More specifically, it checks that the first argument is a [path value type](#), and that the second argument is a [valid subpath string](#).

Laws:

- Not influenced by subpath [normalisation](#):

```
append p s == append p (subpath.normalise s)
```

path

The absolute path to append to

subpath

The subpath string to append

v: stable -

Example 190. `lib.path.append` usage example

```
append /foo "bar/baz"
=> /foo/bar/baz

# subpaths don't need to be normalised
append /foo "./bar//baz/./"
=> /foo/bar/baz

# can append to root directory
append /. "foo/bar"
=> /foo/bar

# first argument needs to be a path value type
append "/foo" "bar"
=> <error>

# second argument needs to be a valid subpath string
append /foo /bar
=> <error>
append /foo ""
=> <error>
append /foo "/bar"
=> <error>
append /foo "../bar"
=> <error>
```

Located at [lib/path/default.nix:192](#) in `<nixpkgs>`.

`lib.path.hasPrefix`

Type: `hasPrefix :: Path -> Path -> Bool`

Whether the first path is a component-wise prefix of the second path.

Laws:

v: stable -

- `hasPrefix p q` is only true if `q == append p s` for some `subpath s`.
- `hasPrefix` is a [non-strict partial order](#) over the set of all path values.

path1

Function argument

Example 191. `lib.path.hasPrefix` usage example

```
hasPrefix /foo /foo/bar
=> true
hasPrefix /foo /foo
=> true
hasPrefix /foo/bar /foo
=> false
hasPrefix /. /foo
=> true
```

Located at [lib/path/default.nix:226](#) in <nixpkgs>.

lib.path.removePrefix

Type: `removePrefix :: Path -> Path -> String`

Remove the first path as a component-wise prefix from the second path. The result is a [normalised subpath string](#).

Laws:

- Inverts [append](#) for [normalised subpath string](#):

```
removePrefix p (append p s) == subpath.normalise s
```

path1

Function argument

v: stable -

Example 192. `lib.path.removePrefix` usage example

```
removePrefix /foo /foo/bar/baz
=> "./bar/baz"
removePrefix /foo /foo
=> "./."
removePrefix /foo/bar /foo
=> <error>
removePrefix /. /foo
=> "./foo"
```

Located at [lib/path/default.nix:271](#) in <nixpkgs>.

lib.path.splitRoot

Type: `splitRoot :: Path -> { root :: Path, subpath :: String }`

Split the filesystem root from a [path](#). The result is an attribute set with these attributes:

- **root**: The filesystem root of the path, meaning that this directory has no parent directory.
- **subpath**: The [normalised subpath string](#) that when [appended](#) to **root** returns the original path.

Laws:

- [Appending](#) the **root** and **subpath** gives the original path:

```
p ==
  append
    (splitRoot p).root
    (splitRoot p).subpath
```

- Trying to get the parent directory of **root** using [readDir](#) returns **root** itself:

```
dirOf (splitRoot p).root == (splitRoot p).root
```

v: stable -

path

The path to split the root off of

Example 193. `lib.path.splitRoot` usage example

```
splitRoot /foo/bar
=> { root = ./; subpath = "./foo/bar"; }

splitRoot .
=> { root = ./; subpath = "./."; }

# Nix neutralises `..` path components for all path values automatically
splitRoot /foo/../../bar
=> { root = ./; subpath = "./bar"; }

splitRoot "/foo/bar"
=> <error>
```

Located at [lib/path/default.nix:336](#) in [<nixpkgs>](#).

lib.path.hasStorePathPrefix

Type: `hasStorePathPrefix :: Path -> Bool`

Whether a [path](#) has a [store path](#) as a prefix.

Note

As with all functions of this `lib.path` library, it does not work on paths in strings, which is how you'd typically get store paths.

Instead, this function only handles path values themselves, which occur when Nix files in the store use relative path expressions.

path

v: stable -

Function argument

Example 194. `lib.path.hasStorePathPrefix` usage example

```
# Subpaths of derivation outputs have a store path as a prefix
hasStorePathPrefix /nix/store/nvl9ic0pj1fpyln3zaqrf4cclbqdfn1j-foo/bar/baz
=> true

# The store directory itself is not a store path
hasStorePathPrefix /nix/store
=> false

# Derivation outputs are store paths themselves
hasStorePathPrefix /nix/store/nvl9ic0pj1fpyln3zaqrf4cclbqdfn1j-foo
=> true

# Paths outside the Nix store don't have a store path prefix
hasStorePathPrefix /home/user
=> false

# Not all paths under the Nix store are store paths
hasStorePathPrefix /nix/store/.links/10gg8k3rmbw8p7gszarbk7qyd9jwxhcfq9i6s
=> false

# Store derivations are also store paths themselves
hasStorePathPrefix /nix/store/nvl9ic0pj1fpyln3zaqrf4cclbqdfn1j-foo.drv
=> true
```

Located at [lib/path/default.nix:390](#) in <nixpkgs>.

lib.path.subpath.isValid

Type: `subpath.isValid :: String -> Bool`

Whether a value is a valid subpath string.

v: stable -

A subpath string points to a specific file or directory within an absolute base directory. It is a stricter form of a relative path that excludes `..` components, since those could escape the base directory.

- The value is a string.
- The string is not empty.
- The string doesn't start with a `/`.
- The string doesn't contain any `..` path components.

value

The value to check

Example 195. `lib.path.subpath.isValid` usage example

```
# Not a string
subpath.isValid null
=> false

# Empty string
subpath.isValid ""
=> false

# Absolute path
subpath.isValid "/foo"
=> false

# Contains a `..` path component
subpath.isValid "../foo"
=> false

# Valid subpath
subpath.isValid "foo/bar"
=> true

# Doesn't need to be normalised
subpath.isValid "./foo//bar/"
```

v: stable -

```
=> true
```

Located at [lib/path/default.nix:447](#) in <nixpkgs>.

lib.path.subpath.join

Type: `subpath.join :: [String] -> String`

Join subpath strings together using `/`, returning a normalised subpath string.

Like `concatStringsSep "/"` but safer, specifically:

- All elements must be [valid subpath strings](#).
- The result gets [normalised](#).
- The edge case of an empty list gets properly handled by returning the neutral subpath `". ./."`.

Laws:

- Associativity:

```
subpath.join [ x (subpath.join [ y z ]) ] == subpath.join [ (subpath.joi
```

- Identity - `". ./."` is the neutral element for normalised paths:

```
subpath.join [ ] == ". ./."
subpath.join [ (subpath.normalise p) ". ./." ] == subpath.normalise p
subpath.join [ ". ./." (subpath.normalise p) ] == subpath.normalise p
```

- Normalisation - the result is [normalised](#):

```
subpath.join ps == subpath.normalise (subpath.join ps)
```

- For non-empty lists, the implementation is equivalent to [normalising](#) the result of

v: stable -

`concatStringsSep "/"`. Note that the above laws can be derived from this one:

```
ps != [] -> subpath.join ps == subpath.normalise (concatStringsSep "/" ps)
```

subpaths

The list of subpaths to join together

Example 196. `lib.path.subpath.join` usage example

```
subpath.join [ "foo" "bar/baz" ]
=> "./foo/bar/baz"

# normalise the result
subpath.join [ "./foo" ".." "bar//./baz/" ]
=> "./foo/bar/baz"

# passing an empty list results in the current directory
subpath.join [ ]
=> "./"

# elements must be valid subpath strings
subpath.join [ /foo ]
=> <error>
subpath.join [ "" ]
=> <error>
subpath.join [ "/foo" ]
=> <error>
subpath.join [ "../foo" ]
=> <error>
```

Located at [lib/path/default.nix:510](#) in `<nixpkgs>`.

lib.path.subpath.components

v: stable -

Type: `subpath.components :: String -> [String]`

Split [a subpath](#) into its path component strings. Throw an error if the subpath isn't valid. Note that the returned path components are also [valid subpath strings](#), though they are intentionally not [normalised](#).

Laws:

- Splitting a subpath into components and [joining](#) the components gives the same subpath but [normalised](#):

```
subpath.join (subpath.components s) == subpath.normalise s
```

`subpath`

The subpath string to split into components

Example 197. `lib.path.subpath.components` usage example

```
subpath.components "."
=> []

subpath.components "./foo//bar//baz/"
=> [ "foo" "bar" "baz" ]

subpath.components "/foo"
=> <error>
```

Located at [lib/path/default.nix:552](#) in `<nixpkgs>`.

`lib.path.subpath.normalise`

Type: `subpath.normalise :: String -> String`

Normalise a subpath. Throw an error if the subpath isn't [valid](#).

- Limit repeating / to a single one.

v: stable -

- Remove redundant `.` components.
- Remove trailing `/` and `/..`.
- Add leading `./`.

Laws:

- Idempotency - normalising multiple times gives the same result:

```
subpath.normalise (subpath.normalise p) == subpath.normalise p
```

- Uniqueness - there's only a single normalisation for the paths that lead to the same file system node:

```
subpath.normalise p != subpath.normalise q -> $(realpath ${p}) != $(real
```

- Don't change the result when appended to a Nix path value:

```
append base p == append base (subpath.normalise p)
```

- Don't change the path according to `realpath`:

```
$(realpath ${p}) == $(realpath ${subpath.normalise p})
```

- Only error on invalid subpaths:

```
builtins.tryEval (subpath.normalise p)).success == subpath.isValid p
```

subpath

The subpath string to normalise

Example 198. lib.path.subpath.normalise usage example

```
# limit repeating `^` to a single one
subpath.normalise "foo//bar"
```

v: stable -

```
=> "./foo/bar"

# remove redundant `.` components
subpath.normalise "foo./bar"
=> "./foo/bar"

# add leading `./`
subpath.normalise "foo/bar"
=> "./foo/bar"

# remove trailing `/`
subpath.normalise "foo/bar/"
=> "./foo/bar"

# remove trailing `./`
subpath.normalise "foo/bar/."
=> "./foo/bar"

# Return the current directory as `./`
subpath.normalise "."
=> "./"

# error on `..` path components
subpath.normalise "foo/../../bar"
=> <error>

# error on empty string
subpath.normalise ""
=> <error>

# error on absolute path
subpath.normalise "/foo"
=> <error>
```

Located at [lib/path/default.nix:633](#) in `<nixpkgs>`.

v: stable -

lib.filesystem: filesystem functions

Functions for querying information about the filesystem without copying any files to the Nix store.

lib.filesystem.pathType

Type: pathType :: Path -> String

The type of a path. The path needs to exist and be accessible. The result is either “directory” for a directory, “regular” for a regular file, “symlink” for a symlink, or “unknown” for anything else.

Example 199. lib.filesystem.pathType usage example

```
pathType /.  
=> "directory"  
  
pathType /some/file.nix  
=> "regular"
```

Located at [lib/filesystem.nix:35](#) in <nixpkgs>.

lib.filesystem.pathIsDirectory

Type: pathIsDirectory :: Path -> Bool

Whether a path exists and is a directory.

path

Function argument

Example 200. lib.filesystem.pathIsDirectory usage example

```
pathIsDirectory /.  
=> true
```

v: stable -

```
pathIsDirectory /this/does/not/exist
=> false
```

```
pathIsDirectory /some/file.nix
=> false
```

Located at [lib/filesystem.nix:67](#) in <nixpkgs>.

lib.filesystem.pathIsRegularFile

Type: pathIsRegularFile :: Path -> Bool

Whether a path exists and is a regular file, meaning not a symlink or any other special file type.

path

Function argument

Example 201. lib.filesystem.pathIsRegularFile usage example

```
pathIsRegularFile /.
=> false
```

```
pathIsRegularFile /this/does/not/exist
=> false
```

```
pathIsRegularFile /some/file.nix
=> true
```

Located at [lib/filesystem.nix:86](#) in <nixpkgs>.

lib.filesystem.haskellPathsInDir

Type: Path -> Map String Path

v: stable -

A map of all haskell packages defined in the given path, identified by having a cabal file with the same name as the directory itself.

root

The directory within to search

Located at [lib/filesystem.nix:96](#) in <nixpkgs>.

lib.filesystem.locateDominatingFile

Type: RegExp -> Path -> Nullable { path : Path; matches : [MatchResults]; }

Find the first directory containing a file matching ‘pattern’ upward from a given ‘file’. Returns ‘null’ if no directories contain a file matching ‘pattern’.

pattern

The pattern to search for

file

The file to start searching upward from

Located at [lib/filesystem.nix:119](#) in <nixpkgs>.

lib.filesystem.listFilesRecursive

Type: Path -> [Path]

Given a directory, return a flattened list of all files within it recursively.

dir

The path to recursively list

Located at [lib/filesystem.nix:147](#) in <nixpkgs>.

lib.fileset: file set functions

The [lib.fileset](#) library allows you to work with *file sets*. A file set is a (mathematical) set of files.

files that can be added to the Nix store for use in Nix derivations. File sets are easy and safe to use, providing obvious and composable semantics with good error messages to prevent mistakes.

Overview

Basics:

- [Implicit coercion from paths to file sets](#)
- [lib.fileset.maybeMissing](#):

Create a file set from a path that may be missing.

- [lib.fileset.trace/lib.fileset.traceVal](#):

Pretty-print file sets for debugging.

- [lib.fileset.toSource](#):

Add files in file sets to the store to use as derivation sources.

Combinators:

- [lib.fileset.union/lib.fileset.unions](#):

Create a larger file set from all the files in multiple file sets.

- [lib.fileset.intersection](#):

Create a smaller file set from only the files in both file sets.

- [lib.fileset.difference](#):

Create a smaller file set containing all files that are in one file set, but not another one.

Filtering:

- [lib.fileset.fileFilter](#):

Create a file set from all files that satisfy a predicate in a directory.

Utilities:

v: stable -

- [lib.fileset.fromSource](#):

Create a file set from a `lib.sources`-based value.

- [lib.fileset.gitTracked/lib.fileset.gitTrackedWith](#):

Create a file set from all tracked files in a local Git repository.

If you need more file set functions, see [this issue](#) to request it.

Implicit coercion from paths to file sets

All functions accepting file sets as arguments can also accept [paths](#) as arguments. Such path arguments are implicitly coerced to file sets containing all files under that path:

- A path to a file turns into a file set containing that single file.
- A path to a directory turns into a file set containing all files *recursively* in that directory.

If the path points to a non-existent location, an error is thrown.

Note

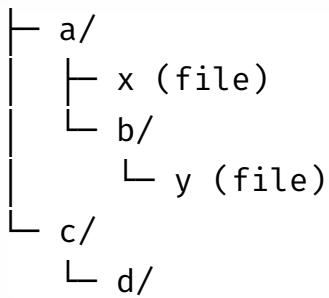
Just like in Git, file sets cannot represent empty directories. Because of this, a path to a directory that contains no files (recursively) will turn into a file set containing no files.

Note

File set coercion does *not* add any of the files under the coerced paths to the store. Only the [toSource](#) function adds files to the Nix store, and only those files contained in the [fileset](#) argument. This is in contrast to using [paths in string interpolation](#), which does add the entire referenced path to the store.

Example

Assume we are in a local directory with a file hierarchy like this:



Here's a listing of which files get included when different path expressions get coerced to file sets:

- `./.` as a file set contains both `a/x` and `a/b/y` (`c/` does not contain any files and is therefore omitted).
- `./a` as a file set contains both `a/x` and `a/b/y`.
- `./a/x` as a file set contains only `a/x`.
- `./a/b` as a file set contains only `a/b/y`.
- `./c` as a file set is empty, since neither `c` nor `c/d` contain any files.

lib.fileset.maybeMissing

Type: `maybeMissing :: Path -> FileSet`

Create a file set from a path that may or may not exist:

- If the path does exist, the path is [coerced to a file set](#).
- If the path does not exist, a file set containing no files is returned.

`path`

Function argument

Example 202. lib.fileset.maybeMissing usage example

```
# All files in the current directory, but excluding main.o if it exists  
difference ./.. (maybeMissing ./main.o)
```

Located at [lib/fileset/default.nix:167](#) in `<nixpkgs>`.

lib.fileset.trace

Type: `trace :: FileSet -> Any -> Any`

Incrementally evaluate and trace a file set in a pretty way. This function is only intended for debugging purposes. The exact tracing format is unspecified and may change.

This function takes a final argument to return. In comparison, [traceVal](#) returns the given file set argument.

This variant is useful for tracing file sets in the Nix repl.

`fileset`

The file set to trace.

This argument can also be a path, which gets [implicitly coerced to a file set](#).

v: stable -

Example 203. `lib.fileset.trace` usage example

```
trace (unions [ ./Makefile ./src ./tests/run.sh ]) null
=>
trace: /home/user/src/myProject
trace: - Makefile (regular)
trace: - src (all files in directory)
trace: - tests
trace:   - run.sh (regular)
null
```

Located at [lib/fileset/default.nix:205](#) in <nixpkgs>.

lib.fileset.traceVal

Type: `traceVal :: FileSet -> FileSet`

Incrementally evaluate and trace a file set in a pretty way. This function is only intended for debugging purposes. The exact tracing format is unspecified and may change.

This function returns the given file set. In comparison, [trace](#) takes another argument to return.

This variant is useful for tracing file sets passed as arguments to other functions.

fileset

The file set to trace and return.

This argument can also be a path, which gets [implicitly coerced to a file set](#).

Example 204. `lib.fileset.traceVal` usage example

```
toSource {
  root = ./;
  fileset = traceVal (unions [
    ./Makefile
    ./src
  ]) v: stable -
```

```
./tests/run.sh
]);
}
=>
trace: /home/user/src/myProject
trace: - Makefile (regular)
trace: - src (all files in directory)
trace: - tests
trace:   - run.sh (regular)
"/nix/store/...-source"
```

Located at [lib/fileset/default.nix:252](#) in `<nixpkgs>`.

lib.fileset.toSource

Type:

```
toSource :: {
  root :: Path,
  fileset :: FileSet,
} -> SourceLike
```

Add the local files contained in `fileset` to the store as a single [store path](#) rooted at `root`.

The result is the store path as a string-like value, making it usable e.g. as the `src` of a derivation, or in string interpolation:

```
stdenv.mkDerivation {
  src = lib.fileset.toSource { ... };
  # ...
}
```

The name of the store path is always `source`.

structured function argument

v: stable -

root

(required) The local directory [path](#) that will correspond to the root of the resulting store path. Paths in [strings](#), including Nix store paths, cannot be passed as `root`. `root` has to be a directory.

Note

Changing `root` only affects the directory structure of the resulting store path, it does not change which files are added to the store. The only way to change which files get added to the store is by changing the `fileset` attribute.

fileset

(required) The file set whose files to import into the store. File sets can be created using other functions in this library. This argument can also be a path, which gets [implicitly coerced to a file set](#).

Note

If a directory does not recursively contain any file, it is omitted from the store path contents.

Example 205. lib.fileset.toSource usage example

```
# Import the current directory into the store
# but only include files under ./src
toSource {
  root = ./;
  fileset = ./src;
}
=> "/nix/store/...-source"

# Import the current directory into the store
# but only include ./Makefile and all files under ./src
toSource {
  root = ./;
```

v: stable -

```
fileset = union
  ./Makefile
  ./src;
}
=> "/nix/store/...-source"

# Trying to include a file outside the root will fail
toSource {
  root = ./.;
  fileset = unions [
    ./Makefile
    ./src
    ../LICENSE
  ];
}
=> <error>

# The root needs to point to a directory that contains all the files
toSource {
  root = ../../;
  fileset = unions [
    ./Makefile
    ./src
    ../LICENSE
  ];
}
=> "/nix/store/...-source"

# The root has to be a local filesystem path
toSource {
  root = "/nix/store/...-source";
  fileset = ./.;
}
=> <error>
```

lib.fileset.union

Type: `union :: FileSet -> FileSet -> FileSet`

The file set containing all files that are in either of two given file sets. This is the same as [unions](#), but takes just two file sets instead of a list. See also [Union \(set theory\)](#).

The given file sets are evaluated as lazily as possible, with the first argument being evaluated first if needed.

`fileset1`

The first file set. This argument can also be a path, which gets [implicitly coerced to a file set](#).

`fileset2`

The second file set. This argument can also be a path, which gets [implicitly coerced to a file set](#).

Example 206. lib.fileset.union usage example

```
# Create a file set containing the file `Makefile`  
# and all files recursively in the `src` directory  
union ./Makefile ./src  
  
# Create a file set containing the file `Makefile`  
# and the LICENSE file from the parent directory  
union ./Makefile ../LICENSE
```

Located at [lib/fileset/default.nix:436](#) in [<nixpkgs>](#).

lib.fileset.unions

Type: `unions :: [FileSet] -> FileSet`

The file set containing all files that are in any of the given file sets. This is the same as [union](#), but takes a list of file sets instead of just two. See also [Union \(set theory\)](#).

The given file sets are evaluated as lazily as possible, with earlier elements being evaluated v: stable -

needed.

filesets

A list of file sets. The elements can also be paths, which get [implicitly coerced to file sets](#).

Example 207. lib.fileset.unions usage example

```
# Create a file set containing selected files
unions [
  # Include the single file `Makefile` in the current directory
  # This errors if the file doesn't exist
  ./Makefile

  # Recursively include all files in the `src/code` directory
  # If this directory is empty this has no effect
  ./src/code

  # Include the files `run.sh` and `unit.c` from the `tests` directory
  ./tests/run.sh
  ./tests/unit.c

  # Include the `LICENSE` file from the parent directory
  ../LICENSE
]
```

Located at [lib/fileset/default.nix:488](#) in <nixpkgs>.

lib.fileset.intersection

Type: intersection :: FileSet -> FileSet -> FileSet

The file set containing all files that are in both of two given file sets. See also [Intersection \(set theory\)](#).

The given file sets are evaluated as lazily as possible, with the first argument being evaluated first if needed.

v: stable -

fileset1

The first file set. This argument can also be a path, which gets [implicitly coerced to a file set](#).

fileset2

The second file set. This argument can also be a path, which gets [implicitly coerced to a file set](#).

Example 208. lib.fileset.intersection usage example

```
# Limit the selected files to the ones in ./, so only ./src and ./Makefile
intersection ./. (unions [ ..LICENSE ./src ./Makefile ])
```

Located at [lib/fileset/default.nix:521](#) in <nixpkgs>.

lib.fileset.difference

Type: union :: FileSet -> FileSet -> FileSet

The file set containing all files from the first file set that are not in the second file set. See also [Difference \(set theory\)](#).

The given file sets are evaluated as lazily as possible, with the first argument being evaluated first if needed.

positive

The positive file set. The result can only contain files that are also in this file set. This argument can also be a path, which gets [implicitly coerced to a file set](#).

negative

The negative file set. The result will never contain files that are also in this file set. This argument can also be a path, which gets [implicitly coerced to a file set](#).

Example 209. lib.fileset.difference usage example

```
# Create a file set containing all files from the current directory,
# except ones under ./tests
```

v: stable -

```
difference ./.. ./tests

let
  # A set of Nix-related files
  nixFiles = unions [ ./default.nix ./nix ./tests/default.nix ];
in
# Create a file set containing all files under ./tests, except ones in `n:
# meaning only without ./tests/default.nix
difference ./tests nixFiles
```

Located at [lib/fileset/default.nix:569](#) in <nixpkgs>.

lib.fileset.fileFilter

Type:

```
fileFilter ::  
({  
  name :: String,  
  type :: String,  
  hasExt :: String -> Bool,  
  ...  
} -> Bool)  
-> Path  
-> FileSet
```

Filter a file set to only contain files matching some predicate.

predicate

The predicate function to call on all files contained in given file set. A file is included in the resulting file set if this function returns true for it.

This function is called with an attribute set containing these attributes:

- **name** (String): The name of the file

v: stable -

- **type** (String, one of "regular", "symlink" or "unknown"): The type of the file. This matches result of calling [`builtins.readFileType`](#) on the file's path.
- **hasExt** (String -> Bool): Whether the file has a certain file extension. `hasExt ext` is true only if `hasSuffix ".${ext}" name`.

This also means that e.g. for a file with name `.gitignore`, `hasExt "gitignore"` is true.

Other attributes may be added in the future.

path

The path whose files to filter

Example 210. `lib.fileset.fileFilter` usage example

```
# Include all regular `default.nix` files in the current directory
fileFilter (file: file.name == "default.nix") ./.

# Include all non-Nix files from the current directory
fileFilter (file: ! file.hasExt "nix") ./.

# Include all files that start with a "." in the current directory
fileFilter (file: hasPrefix "." file.name) ./.

# Include all regular files (not symlinks or others) in the current directory
fileFilter (file: file.type == "regular") ./.
```

Located at [lib/fileset/default.nix:625](#) in `<nixpkgs>`.

lib.fileset.fromSource

Type: `fromSource :: SourceLike -> FileSet`

Create a file set with the same files as a `lib.sources`-based value. This does not import any of the files into the store.

v: stable -

This can be used to gradually migrate from `lib.sources`-based filtering to `lib.fileset`.

A file set can be turned back into a source using [`toSource`](#).

Note

File sets cannot represent empty directories. Turning the result of this function back into a source using `toSource` will therefore not preserve empty directories.

source

Function argument

Example 211. `lib.fileset.fromSource` usage example

```
# There's no cleanSource-like function for file sets yet,
# but we can just convert cleanSource to a file set and use it that way
toSource {
    root = ./;
    fileset = fromSource (lib.sources.cleanSource ./);
}

# Keeping a previous sourceByRegex (which could be migrated to `lib.fileset`)
# but removing a subdirectory using file set functions
difference
  (fromSource (lib.sources.sourceByRegex ./ [ "README*.md$"
    # This regex includes everything in ./doc
    "^doc(/.*)?$")
  ])
  ./doc/generated

# Use cleanSource, but limit it to only include ./Makefile and files under
intersection
  (fromSource (lib.sources.cleanSource ./))
  (unions [
    ./Makefile
    ./src
```

v: stable -

```
]);
```

Located at [lib/fileset/default.nix:707](#) in <nixpkgs>.

lib.fileset.gitTracked

Type: gitTracked :: Path -> FileSet

Create a file set containing all [Git-tracked files](#) in a repository.

This function behaves like [gitTrackedWith { }](#) - using the defaults.

path

The [path](#) to the working directory of a local Git repository. This directory must contain a `.git` file or subdirectory.

Example 212. lib.fileset.gitTracked usage example

```
# Include all files tracked by the Git repository in the current directory
gitTracked ./.

# Include only files tracked by the Git repository in the parent directory
# that are also in the current directory
intersection ./. (gitTracked ../../)
```

Located at [lib/fileset/default.nix:750](#) in <nixpkgs>.

lib.fileset.gitTrackedWith

Type: gitTrackedWith :: { recurseSubmodules :: Bool ? false } -> Path -> FileSet

Create a file set containing all [Git-tracked files](#) in a repository. The first argument allows configuration with an attribute set, while the second argument is the path to the Git working tree.

v: stable -

`gitTrackedWith` does not perform any filtering when the path is a [Nix store path](#) and not a repository. In this way, it accommodates the use case where the expression that makes the `gitTracked` call does not reside in an actual git repository anymore, and has presumably already been fetched in a way that excludes untracked files. Fetchers with such equivalent behavior include `builtins.fetchGit`, `builtins.fetchTree` (experimental), and `pkgs.fetchgit` when used without `leaveDotGit`.

If you don't need the configuration, you can use [`gitTracked`](#) instead.

This is equivalent to the result of [`unions`](#) on all files returned by [`git ls-files`](#) (which uses [`--cached`](#) by default).

Warning

Currently this function is based on [`builtins.fetchGit`](#). As such, this function causes all Git-tracked files to be unnecessarily added to the Nix store, without being re-usable by [`toSource`](#).

This may change in the future.

structured function argument

`reurseSubmodules`

(optional, default: `false`) Whether to recurse into [`Git submodules`](#) to also include their tracked files.

If `true`, this is equivalent to passing the [`-recurse-submodules`](#) flag to [`git ls-files`](#).

`path`

The [`path`](#) to the working directory of a local Git repository. This directory must contain a `.git` file or subdirectory.

Example 213. `lib.fileset.gitTrackedWith` usage example

```
# Include all files tracked by the Git repository in the current directory
# and any submodules under it
gitTracked { recurseSubmodules = true; } ./
```

v: stable -

Located at [lib/fileset/default.nix:794](#) in <nixpkgs>.

lib.sources: source filtering functions

Functions for copying sources to the Nix store.

lib.sources.commitIdFromRepo

Get the commit id of a git repo.

path

Function argument

Example 214. lib.sources.commitIdFromRepo usage example

```
commitIdFromRepo <nixpkgs/.git>
```

Located at [lib/sources.nix:271](#) in <nixpkgs>.

lib.sources.cleanSource

Filters a source tree removing version control files and directories using cleanSourceFilter.

src

Function argument

Example 215. lib.sources.cleanSource usage example

```
cleanSource ./.
```

Located at [lib/sources.nix:271](#) in <nixpkgs>.

v: stable -

lib.sources.cleanSourceWith

Like `builtins.filterSource`, except it will compose with itself, allowing you to chain multiple calls together without any intermediate copies being put in the nix store.

structured function argument

`src`

A path or `cleanSourceWith` result to filter and/or rename.

`filter`

Optional with default value: constant true (include everything) The function will be combined with the `&&` operator such that `src.filter` is called lazily. For implementing a filter, see <https://nixos.org/nix/manual/#builtin-filterSource> Type: A function (path -> type -> bool)

`name`

Optional name to use as part of the store path. This defaults to `src.name` or otherwise "source".

Example 216. lib.sources.cleanSourceWith usage example

```
lib.cleanSourceWith {  
    filter = f;  
    src = lib.cleanSourceWith {  
        filter = g;  
        src = ./.;  
    };  
}  
# Succeeds!  
  
builtins.filterSource f (builtins.filterSource g ./.)  
# Fails!
```

Located at [lib/sources.nix:271](#) in <nixpkgs>.

v: stable -

lib.sources.cleanSourceFilter

A basic filter for `cleanSourceWith` that removes directories of version control system, backup files (`*~`) and some generated files.

`name`

Function argument

`type`

Function argument

Located at [lib/sources.nix:271](#) in `<nixpkgs>`.

lib.sources.sourceByRegex

Filter sources by a list of regular expressions.

`src`

Function argument

`regexes`

Function argument

Example 217. lib.sources.sourceByRegex usage example

```
src = sourceByRegex ./my-subproject [".*\.\py$" "^database\.sql$"]
```

Located at [lib/sources.nix:271](#) in `<nixpkgs>`.

lib.sources.sourceFilesBySuffices

Type: `sourceLike -> [String] -> Source`

Get all files ending with the specified suffices from the given source directory or its descendants, omitting files that do not match any suffix. The result of the example below will include files like `v: stable -`

./dir/module.c and ./dir/subdir/doc.xml if present.

SRC

Path or source containing the files to be returned

exts

A list of file suffix strings

Example 218. lib.sources.sourceFilesBySuffixes usage example

```
sourceFilesBySuffixes ./ [ ".xml" ".c" ]
```

Located at lib/sources.nix:271 in <nixpkgs>.

lib.sources.trace

Type: sources.trace :: sourceLike -> Source

Add logging to a source, for troubleshooting the filtering behavior.

src

Source to debug. The returned source will behave like this source, but also log its filter invocations

Located at lib/sources.nix:271 in <nixpkgs>.

lib.cli: command-line serialization functions

libcli.toGNUCommandLineShell

Automatically convert an attribute set to command-line options.

This helps protect against malformed command lines and also to reduce boilerplate related to command-line construction for simple use cases.

`toGNULinux` returns a list of pix strings. `toGNULinuxShell` returns an `Object`.

shell string.

options

Function argument

attrs

Function argument

Example 219. `lib.cli.toGNUCommandLineShell` usage example

```
cli.toCommandLine {} {  
    data = builtins.toJSON { id = 0; };  
    X = "PUT";  
    retry = 3;  
    retry-delay = null;  
    url = [ "https://example.com/foo" "https://example.com/bar" ];  
    silent = false;  
    verbose = true;  
}  
=> [  
    "-X" "PUT"  
    "--data" "{\"id\":0}"  
    "--retry" "3"  
    "--url" "https://example.com/foo"  
    "--url" "https://example.com/bar"  
    "--verbose"  
]  
  
cli.toGNUCommandLineShell {} {  
    data = builtins.toJSON { id = 0; };  
    X = "PUT";  
    retry = 3;  
    retry-delay = null;  
    url = [ "https://example.com/foo" "https://example.com/bar" ];  
    silent = false;  
    verbose = true;  
}
```

v: stable -

```
=> "'-X' 'PUT' '--data' '{\"id\":0}' '--retry' '3' '--url' 'https://example.com/api/v1/items/0'
```

Located at [lib/cli.nix:42](#) in <nixpkgs>.

lib.gvariant: GVariant formatted string serialization functions

A partial and basic implementation of GVariant formatted strings. See [GVariant Format Strings](#) for details.

Warning

This API is not considered fully stable and it might therefore change in backwards incompatible ways without prior notice.

lib.gvariant.isGVariant

Type: `isGVariant :: Any -> Bool`

Check if a value is a GVariant value

v

Function argument

Located at [lib/gvariant.nix:54](#) in <nixpkgs>.

lib.gvariant.mkValue

Type: `mkValue :: Any -> gvariant`

Returns the GVariant value that most closely matches the given Nix value. If no GVariant value can be found unambiguously then error is thrown.

v

Function argument

Located at [lib/gvariant.nix:62](#) in <nixpkgs>.

v: stable -

lib.gvariant.mkArray

Type: `mkArray :: [Any] -> gvariant`

Returns the GVariant array from the given type of the elements and a Nix list.

`elems`

Function argument

Example 220. lib.gvariant.mkArray usage example

```
# Creating a string array
lib.gvariant.mkArray [ "a" "b" "c" ]
```

Located at [lib/gvariant.nix:85](#) in <nixpkgs>.

lib.gvariant.mkEmptyArray

Type: `mkEmptyArray :: gvariant.type -> gvariant`

Returns the GVariant array from the given empty Nix list.

`elemType`

Function argument

Example 221. lib.gvariant.mkEmptyArray usage example

```
# Creating an empty string array
lib.gvariant.mkEmptyArray (lib.gvariant.type.string)
```

Located at [lib/gvariant.nix:106](#) in <nixpkgs>.

lib.gvariant.mkVariant

v: stable -

Type: `mkVariant :: Any -> gvariant`

Returns the GVariant variant from the given Nix value. Variants are containers of different GVariant type.

`elem`

Function argument

Example 222. lib.gvariant.mkVariant usage example

```
lib.gvariant.mkArray [
  (lib.gvariant.mkVariant "a string")
  (lib.gvariant.mkVariant (lib.gvariant.mkInt32 1))
]
```

Located at [lib/gvariant.nix:123](#) in <nixpkgs>.

lib.gvariant.mkDictionaryEntry**Type:** `mkDictionaryEntry :: String -> Any -> gvariant`

Returns the GVariant dictionary entry from the given key and value.

`name`

The key of the entry

`value`

The value of the entry

Example 223. lib.gvariant.mkDictionaryEntry usage example

```
# A dictionary describing an Epiphany's search provider
[
  (lib.gvariant.mkDictionaryEntry "url" (lib.gvariant.mkVariant "https://duckduckgo.com/search?q={query}"))
  (lib.gvariant.mkDictionaryEntry "bang" (lib.gvariant.mkVariant "!d"))
  (lib.gvariant.mkDictionaryEntry "name" (lib.gvariant.mkVariant "DuckDuckGo"))
]
```

Located at [lib/gvariant.nix:142](#) in <nixpkgs>.

lib.gvariant.mkMaybe

Type: `mkMaybe :: gvariant.type -> Any -> gvariant`

Returns the GVariant maybe from the given element type.

`elementType`

Function argument

`elem`

Function argument

Located at [lib/gvariant.nix:161](#) in <nixpkgs>.

lib.gvariant.mkNothing

Type: `mkNothing :: gvariant.type -> gvariant`

Returns the GVariant nothing from the given element type.

`elementType`

Function argument

Located at [lib/gvariant.nix:175](#) in <nixpkgs>.

lib.gvariant.mkJust

v: stable -

Type: `mkJust :: Any -> gvariant`

Returns the GVariant just from the given Nix value.

`elem`

Function argument

Located at [lib/gvariant.nix:182](#) in <nixpkgs>.

lib.gvariant.mkTuple

Type: `mkTuple :: [Any] -> gvariant`

Returns the GVariant tuple from the given Nix list.

`elems`

Function argument

Located at [lib/gvariant.nix:189](#) in <nixpkgs>.

lib.gvariant.mkBoolean

Type: `mkBoolean :: Bool -> gvariant`

Returns the GVariant boolean from the given Nix bool value.

`v`

Function argument

Located at [lib/gvariant.nix:204](#) in <nixpkgs>.

lib.gvariant.mkString

Type: `mkString :: String -> gvariant`

Returns the GVariant string from the given Nix string value.

`v`

Function argument

v: stable -

Located at [lib/gvariant.nix:214](#) in <nixpkgs>.

lib.gvariant.mkObjectpath

Type: `mkObjectpath :: String -> gvariant`

Returns the GVariant object path from the given Nix string value.

v

Function argument

Located at [lib/gvariant.nix:225](#) in <nixpkgs>.

lib.gvariant.mkUchar

Type: `mkUchar :: Int -> gvariant`

Returns the GVariant uchar from the given Nix int value.

Located at [lib/gvariant.nix:235](#) in <nixpkgs>.

lib.gvariant.mkInt16

Type: `mkInt16 :: Int -> gvariant`

Returns the GVariant int16 from the given Nix int value.

Located at [lib/gvariant.nix:242](#) in <nixpkgs>.

lib.gvariant.mkUint16

Type: `mkUint16 :: Int -> gvariant`

Returns the GVariant uint16 from the given Nix int value.

Located at [lib/gvariant.nix:249](#) in <nixpkgs>.

lib.gvariant.mkInt32

v: stable -

Type: `mkInt32 :: Int -> gvariant`

Returns the GVariant int32 from the given Nix int value.

v

Function argument

Located at [lib/gvariant.nix:256](#) in <nixpkgs>.

lib.gvariant.mkUInt32

Type: `mkUInt32 :: Int -> gvariant`

Returns the GVariant uint32 from the given Nix int value.

Located at [lib/gvariant.nix:266](#) in <nixpkgs>.

lib.gvariant.mkInt64

Type: `mkInt64 :: Int -> gvariant`

Returns the GVariant int64 from the given Nix int value.

Located at [lib/gvariant.nix:273](#) in <nixpkgs>.

lib.gvariant.mkUInt64

Type: `mkUInt64 :: Int -> gvariant`

Returns the GVariant uint64 from the given Nix int value.

Located at [lib/gvariant.nix:280](#) in <nixpkgs>.

lib.gvariant.mkDouble

Type: `mkDouble :: Float -> gvariant`

Returns the GVariant double from the given Nix float value.

v: stable -

v

Function argument

Located at [lib/gvariant.nix:287](#) in <nixpkgs>.

lib.customisation: Functions to customise (derivation-related) functions, derivatons, or attribute sets

lib.customisation.overrideDerivation

Type: `overrideDerivation :: Derivation -> (Derivation -> AttrSet) -> Derivation`

`overrideDerivation drv f` takes a derivation (i.e., the result of a call to the builtin function `derivation`) and returns a new derivation in which the attributes of the original are overridden according to the function `f`. The function `f` is called with the original derivation attributes.

`overrideDerivation` allows certain “ad-hoc” customisation scenarios (e.g. in `~/.config/nixpkgs/config.nix`). For instance, if you want to “patch” the derivation returned by a package function in Nixpkgs to build another version than what the function itself provides.

For another application, see `build-support/vm`, where this function is used to build arbitrary derivations inside a QEMU virtual machine.

Note that in order to preserve evaluation errors, the new derivation’s `outPath` depends on the old one’s, which means that this function cannot be used in circular situations when the old derivation also depends on the new one.

You should in general prefer `drv.overrideAttrs` over this function; see the nixpkgs manual for more information on overriding.

`drv`

Function argument

`f`

Function argument

Example 224. `lib.customisation.overrideDerivation` usage example

```
mySed = overrideDerivation pkgs.gnused (oldAttrs: {
  name = "sed-4.2.2-pre";
  src = fetchurl {
    url = ftp://alpha.gnu.org/gnu/sed/sed-4.2.2-pre.tar.bz2;
    hash = "sha256-MxBJRcM2rYzQYwJ5XKhXTQByvSg5jZc5cSHEZoB2IY=";
  };
  patches = [];
});
```

Located at [lib/customisation.nix:43](#) in <nixpkgs>.

lib.customisation.makeOverridable

Type: `makeOverridable :: (AttrSet -> a) -> AttrSet -> a`

`makeOverridable` takes a function from attribute set to attribute set and injects `override` attribute which can be used to override arguments of the function.

Please refer to documentation on [`<pkg>.overrideDerivation`](#) to learn about `overrideDerivation` and caveats related to its use.

f

Function argument

Example 225. `lib.customisation.makeOverridable` usage example

```
nix-repl> x = {a, b}: { result = a + b; }

nix-repl> y = lib.makeOverridable x { a = 1; b = 2; }

nix-repl> y
{ override = «lambda»; overrideDerivation = «lambda»; result = 3; }
```

v: stable -

```
nix-repl> y.override { a = 10; }
{ override = «lambda»; overrideDerivation = «lambda»; result = 12; }
```

Located at [lib/customisation.nix:79](#) in <nixpkgs>.

lib.customisation.callPackageWith

Type: callPackageWith :: AttrSet -> ((AttrSet -> a) | Path) -> AttrSet -> a

Call the package function in the file `fn` with the required arguments automatically. The function is called with the arguments `args`, but any missing arguments are obtained from `autoArgs`. This function is intended to be partially parameterised, e.g.,

```
callPackage = callPackageWith pkgs;
pkgs = {
  libfoo = callPackage ./foo.nix { };
  libbar = callPackage ./bar.nix { };
};
```

If the `libbar` function expects an argument named `libfoo`, it is automatically passed as an argument. Overrides or missing arguments can be supplied in `args`, e.g.

```
libbar = callPackage ./bar.nix {
  libfoo = null;
  enableX11 = true;
};
```

`autoArgs`

Function argument

`fn`

Function argument

`args`

v: stable -

Function argument

Located at [lib/customisation.nix:141](#) in <nixpkgs>.

lib.customisation.callPackagesWith

Type: callPackagesWith :: AttrSet -> ((AttrSet -> AttrSet) | Path) -> AttrSet -> AttrSet

Like callPackage, but for a function that returns an attribute set of derivations. The override function is added to the individual attributes.

autoArgs

Function argument

fn

Function argument

args

Function argument

Located at [lib/customisation.nix:202](#) in <nixpkgs>.

lib.customisation.extendDerivation

Type: extendDerivation :: Bool -> Any -> Derivation -> Derivation

Add attributes to each output of a derivation without changing the derivation itself and check a given condition when evaluating.

condition

Function argument

passthru

Function argument

drv

Function argument

v: stable -

Located at [lib/customisation.nix:223](#) in <nixpkgs>.

lib.customisation.hydraJob

Type: hydraJob :: (Derivation | Null) -> (Derivation | Null)

Strip a derivation of all non-essential attributes, returning only those needed by hydra-eval-jobs. Also strictly evaluate the result to ensure that there are no thunks kept alive to prevent garbage collection.

drv

Function argument

Located at [lib/customisation.nix:261](#) in <nixpkgs>.

lib.customisation.makeScope

Type: makeScope :: (AttrSet -> ((AttrSet -> a) | Path) -> AttrSet -> a) -> (AttrSet -> AttrSet) -> AttrSet

Make a set of packages with a common scope. All packages called with the provided `callPackage` will be evaluated with the same arguments. Any package in the set may depend on any other. The `overrideScope`' function allows subsequent modification of the package set in a consistent way, i.e. all packages in the set will be called with the overridden packages. The package sets may be hierarchical: the packages in the set are called with the scope provided by `newScope` and the set provides a `newScope` attribute which can form the parent scope for later package sets.

newScope

Function argument

f

Function argument

Located at [lib/customisation.nix:303](#) in <nixpkgs>.

lib.customisation.makeScopeWithSplicing

backward compatibility with old uncurried form; deprecated

v: stable -

`splicePackages`

Function argument

`newScope`

Function argument

`otherSplices`

Function argument

`keep`

Function argument

`extra`

Function argument

`f`

Function argument

Located at [lib/customisation.nix:317](#) in <nixpkgs>.

lib.customisation.makeScopeWithSplicing'

Type:

```
makeScopeWithSplicing' ::  
{ splicePackages :: Splice -> AttrSet  
, newScope :: AttrSet -> ((AttrSet -> a) | Path) -> AttrSet -> a  
}  
-> { otherSplices :: Splice, keep :: AttrSet -> AttrSet, extra :: AttrSet  
-> AttrSet  
  
Splice ::  
{ pkgsBuildBuild :: AttrSet  
, pkgsBuildHost :: AttrSet  
, pkgsBuildTarget :: AttrSet  
, pkgsHostHost :: AttrSet  
, pkgsHostTarget :: AttrSet
```

v: stable -

```
, pkgsTargetTarget :: AttrSet
}
```

Like makeScope, but aims to support cross compilation. It's still ugly, but hopefully it helps a little bit.

structured function argument

splicePackages

Function argument

newScope

Function argument

structured function argument

otherSplices

Function argument

keep

Function argument

extra

Function argument

f

Function argument

Located at [lib/customisation.nix:343](#) in <nixpkgs>.

lib.meta: functions for derivation metadata

Some functions for manipulating meta attributes, as well as the name attribute.

lib.meta.addAttribute

Add to or override the meta attributes of the given derivation.

newAttrs

v: stable -

Function argument

drv

Function argument

Example 226. `lib.meta.addMetaAttrs` usage example

```
addMetaAttrs {description = "Bla blah";} somePkg
```

Located at [lib/meta.nix:20](#) in <nixpkgs>.

lib.meta.dontDistribute

Disable Hydra builds of given derivation.

drv

Function argument

Located at [lib/meta.nix:26](#) in <nixpkgs>.

lib.meta.setName

Change the symbolic name of a package for presentation purposes (i.e., so that nix-env users can tell them apart).

name

Function argument

drv

Function argument

Located at [lib/meta.nix:32](#) in <nixpkgs>.

lib.meta.updateName

v: stable -

Like `setName`, but takes the previous name as an argument.

updater

Function argument

drv

Function argument

Example 227. `lib.meta.updateName` usage example

```
updateName (oldName: oldName + "-experimental") somePkg
```

Located at [lib/meta.nix:40](#) in `<nixpkgs>`.

lib.meta.appendToName

Append a suffix to the name of a package (before the version part).

suffix

Function argument

Located at [lib/meta.nix:45](#) in `<nixpkgs>`.

lib.meta.mapDerivationAttrset

Apply a function to each derivation and only to derivations in an attrset.

f

Function argument

set

Function argument

Located at [lib/meta.nix:51](#) in `<nixpkgs>`.

lib.meta.setPrio

Set the nix-env priority of the package.

priority

Function argument

Located at [lib/meta.nix:55](#) in <nixpkgs>.

lib.meta.lowPrio

Decrease the nix-env priority of the package, i.e., other versions/variants of the package will be preferred.

Located at [lib/meta.nix:60](#) in <nixpkgs>.

lib.meta.lowPrioSet

Apply lowPrio to an attrset with derivations

set

Function argument

Located at [lib/meta.nix:64](#) in <nixpkgs>.

lib.meta.hiPrio

Increase the nix-env priority of the package, i.e., this version/variant of the package will be preferred.

Located at [lib/meta.nix:70](#) in <nixpkgs>.

lib.meta.hiPrioSet

Apply hiPrio to an attrset with derivations

set

Function argument

v: stable -

Located at [lib/meta.nix:74](#) in <nixpkgs>.

lib.meta.platformMatch

Check to see if a platform is matched by the given `meta.platforms` element.

A `meta.platform` pattern is either

1. (legacy) a system string.
2. (modern) a pattern for the entire platform structure (see `lib.systems.inspect.platformPatterns`).
3. (modern) a pattern for the platform `parsed` field (see `lib.systems.inspect.patterns`).

We can inject these into a pattern for the whole of a structured platform, and then match that.

platform

Function argument

elem

Function argument

Located at [lib/meta.nix:91](#) in <nixpkgs>.

lib.meta.availableOn

Check if a package is available on a given platform.

A package is available on a platform if both

1. One of `meta.platforms` pattern matches the given platform, or `meta.platforms` is not present.
2. None of `meta.badPlatforms` pattern matches the given platform.

platform

Function argument

v: stable -

pkg

Function argument

Located at [lib/meta.nix:116](#) in <nixpkgs>.

lib.meta.getLicenseFromSpdxId

Type: getLicenseFromSpdxId :: str -> AttrSet

Get the corresponding attribute in lib.licenses from the SPDX ID. For SPDX IDs, see <https://spdx.org/licenses>

Example 228. lib.meta.getLicenseFromSpdxId usage example

```
lib.getLicenseFromSpdxId "MIT" == lib.licenses.mit
=> true
lib.getLicenseFromSpdxId "mIt" == lib.licenses.mit
=> true
lib.getLicenseFromSpdxId "MY LICENSE"
=> trace: warning: getLicenseFromSpdxId: No license matches the given SPDX ID
=> { shortName = "MY LICENSE"; }
```

Located at [lib/meta.nix:137](#) in <nixpkgs>.

lib.meta.getExe

Type: getExe :: package -> string

Get the path to the main program of a package based on meta.mainProgram

x

Function argument

Example 229. lib.meta.getExe usage example

v: stable -

```
getExe pkgs.hello
=> "/nix/store/g124820p9hlv4lj8qplzxw1c44dxaw1k-hello-2.12/bin/hello"
getExe pkgs.mustache-go
=> "/nix/store/am9ml4f4ywvivxnkiaqwr0hyxka1xjsf-mustache-go-1.3.0/bin/mustache-go"
```

Located at [lib/meta.nix:157](#) in <nixpkgs>.

lib.meta.getExe'

Type: getExe' :: derivation -> string -> string

Get the path of a program of a derivation.

x

Function argument

y

Function argument

Example 230. lib.meta.getExe' usage example

```
getExe' pkgs.hello "hello"
=> "/nix/store/g124820p9hlv4lj8qplzxw1c44dxaw1k-hello-2.12/bin/hello"
getExe' pkgs.imagemagick "convert"
=> "/nix/store/5rs48jamq7k6sal98ymj9l4k2bnwq515-imagemagick-7.1.1-15/bin/convert"
```

Located at [lib/meta.nix:177](#) in <nixpkgs>.

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` v: stable -

`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 delegates 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"
```

v: stable -

```
[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`.

Debugging Nix Expressions

Nix is a untyped, 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 running. 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.

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
v: stable -
```

EOF

pkgs.nix-gitignore

Usage

gitignore files in subdirectories

`pkgs.nix-gitignore` is a function that acts similarly to `builtins.filterSource` but also allows filtering with the help of the gitignore format.

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 ignore lines.

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", ~/.gitignore]
  # It also accepts a list (of strings and paths) that will be concatenated
  # once the paths are turned to strings via readFile.
```

These functions are derived from the `Filter` functions by setting the first filter argument to `pkgs.nix-gitignore`.

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 if 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
```

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);
```

Module System

Table of Contents

[Introduction](#)

[lib.evalModules](#)

Introduction

v: stable -

The module system is a language for handling configuration, implemented as a Nix library.

Compared to plain Nix, it adds documentation, type checking and composition or extensibility.

Note

This chapter is new and not complete yet. For a gentle introduction to the module system, in the context of NixOS, see [Writing NixOS Modules](#) in the NixOS manual.

lib.evalModules

[Parameters](#)

[Return value](#)

Evaluate a set of modules. This function is typically only used once per application (e.g. once in NixOS, once in Home Manager, ...).

Parameters

modules

A list of modules. These are merged together to form the final configuration.

specialArgs

An attribute set of module arguments that can be used in `imports`.

This is in contrast to `config._module.args`, which is only available after all `imports` have been resolved.

class

If the `class` attribute is set and non-null, the module system will reject `imports` with a message like:

class declaration.

The `class` value should be a string in lower [camel case](#).

If applicable, the `class` should match the “prefix” of the attributes used in (experimental) [flakes](#). Some examples are:

- `nixos` as in [flake.nixosModules](#)
- `nixosTest`: modules that constitute a [NixOS VM test](#)

prefix

A list of strings representing the location at or below which all options are evaluated. This is used by `types.submodule` to improve error reporting and find the implicit `name` module argument.

Return value

The result is an attribute set with the following attributes:

options

The nested attribute set of all option declarations.

config

The nested attribute set of all option values.

type

A module system type. This type is an instance of `types.submoduleWith` containing the current [modules](#).

The option definitions that are typed with this type will extend the current set of modules, like [extendModules](#).

However, the value returned from the type is just the [config](#), like any submodule.

v: stable -

If you're familiar with prototype inheritance, you can think of this `evalModules` invocation as the prototype, and usages of this type as the instances.

This type is also available to the [`modules`](#) as the module argument `moduleType`.

extendModules

A function similar to `evalModules` but building on top of the already passed [`modules`](#). Its arguments, `modules` and `specialArgs` are added to the existing values.

If you're familiar with prototype inheritance, you can think of the current, actual `evalModules` invocation as the prototype, and the return value of `extendModules` as the instance.

This functionality is also available to modules as the `extendModules` module argument.

Note

Evaluation Performance

`extendModules` returns a configuration that shares very little with the original `evalModules` invocation, because the module arguments may be different.

So if you have a configuration that has been (or will be) largely evaluated, almost none of the computation is shared with the configuration returned by `extendModules`.

The real work of module evaluation happens while computing the values in `config` and `options`, so multiple invocations of `extendModules` have a particularly small cost, as long as only the final `config` and `options` are evaluated.

If you do reference multiple `config` (or `options`) from before and after `extendModules`, evaluation performance is the same as with multiple `evalModules` invocations, because the new modules' ability to override existing configuration fundamentally requires constructing a new `config` and `options` fixpoint.

_module

A portion of the configuration tree which is elided from `config`.

v: stable -

_type

A nominal type marker, always "configuration".

class

The [class](#) argument.

Standard environment

Table of Contents

[The Standard Environment](#)

[Meta-attributes](#)

[Multiple-output packages](#)

[Cross-compilation](#)

[Platform Notes](#)

The Standard Environment

Table of Contents

[Using stdenv](#)

[Tools provided by stdenv](#)

[Specifying dependencies](#)

[Attributes](#)

[Phases](#)

[Shell functions and utilities](#)

v: stable -

[Package setup hooks](#)

[Purity in Nixpkgs](#)

[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.

Using `stdenv`

[Building a `stdenv` package in `nix-shell`](#)

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";  
    hash = "sha256-tWxU/LANbQE32my+9AXyt3nCT7NBVfJ45CX757EMT3Q=";  
  };  
}
```

(`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:

v: stable -

```
stdenv.mkDerivation rec {
  pname = "libfoo";
  version = "1.2.3";
  src = fetchurl {
    url = "http://example.org/libfoo-source-${version}.tar.bz2";
    hash = "sha256-tWxU/LANbQE32my+9AXyt3nCT7NBVfJ45CX757EMT3Q=";
  };
}
```

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 {
  pname = "libfoo";
  version = "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 [the section called “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 [the section called “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 {
  pname = "fnord";
  version = "4.5";
  ...
  buildPhase = ''
    gcc foo.c -o foo
```

v: stable -

```
'';
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 [the section called "Attributes".](#)

While the standard environment provides a generic builder, you can still supply your own build script:

```
stdenv.mkDerivation {
  pname = "libfoo";
  version = "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. by resetting `PATH` and populating it from build inputs). 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
}
```

v: stable -

```
installPhase() {
  mkdir -p $out/bin
  cp foo $out/bin
}

genericBuild
```

Building a stdenv package in nix-shell

To build a stdenv package in a [nix-shell](#), enter a shell, find the [phases](#) you wish to build, then invoke `genericBuild` manually:

Go to an empty directory, invoke `nix-shell` with the desired package, and from inside the shell, set the output variables to a writable directory:

```
cd "$(mktemp -d)"
nix-shell '<nixpkgs>' -A some_package
export out=$(pwd)/out
```

Next, invoke the desired parts of the build. First, run the phases that generate a working copy of the sources, which will change directory to the sources for you:

```
phases="${prePhases[*]:-} unpackPhase patchPhase" genericBuild
```

Then, run more phases up until the failure is reached. If the failure is in the build or check phase, the following phases would be required:

```
phases="${preConfigurePhases[*]:-} configurePhase ${preBuildPhases[*]:-} |
```

Use this command to run all install phases:

```
phases="${preInstallPhases[*]:-} installPhase ${preFixupPhases[*]:-} fixup
```

Single phase can be re-run as many times as necessary to examine the failure like so:

v: stable -

```
phases="buildPhase" genericBuild
```

To modify a [phase](#), first print it with

```
echo "$buildPhase"
```

Or, if that is empty, for instance, if it is using a function:

```
type buildPhase
```

then change it in a text editor, and paste it back to the terminal.

Note

This method may have some inconsistencies in environment variables and behaviour compared to a normal build within the [Nix build sandbox](#). The following is a non-exhaustive list of such differences:

- `TMP`, `TMPDIR`, and similar variables likely point to non-empty directories that the build might conflict with files in.
- Output store paths are not writable, so the variables for outputs need to be overridden to writable paths.
- Other environment variables may be inconsistent with a `nix-build` either due to `nix-shell`'s initialization script or due to the use of `nix-shell` without the `--pure` option.

If the build fails differently inside the shell than in the sandbox, consider using [`breakpointHook`](#) and invoking `nix-build` instead. The [`--keep-failed`](#) option for `nix-build` may also be useful to examine the build directory of a failed build.

Tools provided by `stdenv`

The standard environment provides the following packages:

v: stable -

- 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.

Specifying dependencies

[Overview](#)

[Reference](#)

Build systems often require more dependencies than just what `stdenv` provides. This section describes attributes accepted by `stdenv.mkDerivation` that can be used to make these dependencies available to the build system.

Overview

v: stable -

A full reference of the different kinds of dependencies is provided in [the section called “Reference”](#), but here is an overview of the most common ones. It should cover most use cases.

Add dependencies to `nativeBuildInputs` if they are executed during the build:

- those which are needed on `$PATH` during the build, for example `cmake` and `pkg-config`
- [setup hooks](#), for example [`makeWrapper`](#)
- interpreters needed by [`patchShebangs`](#) for build scripts (with the `--build` flag), which can be the case for e.g. `perl`

Add dependencies to `buildInputs` if they will end up copied or linked into the final output or otherwise used at runtime:

- libraries used by compilers, for example `zlib`,
- interpreters needed by [`patchShebangs`](#) for scripts which are installed, which can be the case for e.g. `perl`

Note

These criteria are independent.

For example, software using Wayland usually needs the `wayland` library at runtime, so `wayland` should be added to `buildInputs`. But it also executes the `wayland-scanner` program as part of the build to generate code, so `wayland` should also be added to `nativeBuildInputs`.

Dependencies needed only to run tests are similarly classified between native (executed during build) and non-native (executed at runtime):

- `nativeCheckInputs` for test tools needed on `$PATH` (such as `ctest`) and [setup hooks](#) (for example [`pytestCheckHook`](#))
- `checkInputs` for libraries linked into test executables (for example the `qcheck` OCaml package)

These dependencies are only injected when [`doCheck`](#) is set to `true`.

Example

Consider for example this simplified derivation for `solo5`, a sandboxing tool:

```
stdenv.mkDerivation rec {
  pname = "solo5";
  version = "0.7.5";

  src = fetchurl {
    url = "https://github.com/Solo5/solo5/releases/download/v${version}/solo5-virtio-mkimage.tar.gz";
    hash = "sha256-viwrS9lnaU8sTGuzK/+L/P1MM/xRRtgVuK5pixVeDEw=";
  };

  nativeBuildInputs = [ makeWrapper pkg-config ];
  buildInputs = [ libseccomp ];

  postInstall = ''
    substituteInPlace $out/bin/solo5-virtio-mkimage \
      --replace "/usr/lib/syslinux" "${syslinux}/share/syslinux" \
      --replace "/usr/share/syslinux" "${syslinux}/share/syslinux" \
      --replace "cp" "cp --no-preserve=mode"

    wrapProgram $out/bin/solo5-virtio-mkimage \
      --prefix PATH : ${lib.makeBinPath [ dosfstools mtools parted syslinux ]};
  '';

  doCheck = true;
  nativeCheckInputs = [ util-linux qemu ];
  checkPhase = '' [elided] '';
}
```

- `makeWrapper` is a setup hook, i.e., a shell script sourced by the generic builder of `stdenv`. It is thus executed during the build and must be added to `nativeBuildInputs`.
- `pkg-config` is a build tool which the configure script of `solo5` expects to be on `$PATH` during the build: therefore, it must be added to `nativeBuildInputs`.

- `libseccomp` is a library linked into `$out/bin/solo5-elftool`. As it is used at runtime, it must be added to `buildInputs`.
- Tests need `qemu` and `getopt` (from `util-linux`) on `$PATH`, these must be added to `nativeCheckInputs`.
- Some dependencies are injected directly in the shell code of phases: `syslinux`, `dosfstools`, `mtools`, and `parted`. In this specific case, they will end up in the output of the derivation (`$out` here). As Nix marks dependencies whose absolute path is present in the output as runtime dependencies, adding them to `buildInputs` is not required.

For more complex cases, like libraries linked into an executable which is then executed as part of the build system, see [the section called “Reference”](#).

Reference

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 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 [the section called “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 [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.

v: stable -

A dependency is said to be **propagated** when some of its other-transitive (non-immediate) downstream dependencies also need it as an immediate dependency. [\[3\]](#)

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. To determine the exact rules for dependency propagation, we start by assigning to each dependency a couple of ternary numbers (-1 for **build**, 0 for **host**, and 1 for **target**) representing its [dependency type](#), which captures how its host and target platforms are each “offset” from the depending derivation’s host and target platforms. The following table summarize the different combinations that can be obtained:

host → target	attribute name	offset
build --> build	depsBuildBuild	-1, -1
build --> host	nativeBuildInputs	-1, 0
build --> target	depsBuildTarget	-1, 1
host --> host	depsHostHost	0, 0
host --> target	buildInputs	0, 1
target --> target	depsTargetTarget	1, 1

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 is sort of 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! [\[4\]](#) 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)
```

v: stable -

```

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, t0, A, B)
propagated-dep(h1, t1, B, C)
h0 + h1 in {-1, 0, 1}
h0 + t1 in {-1, 0, -1}
----- Take immediate dependencies' propagated dependencies
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 $\sum_{h \in \text{host}} h$.

offset of the transitive dependency. The target offset is the transitive dependency is 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 depending 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 unforgeable 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.

Variables specifying dependencies

`depsBuildBuild`

A list of dependencies whose host and target platforms are the new derivation's build platform. 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

v: stable -

run-time dependencies. This isn't currently enforced, but could be in the future.

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. 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.

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. 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, `-1` and `1`, 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.

dependencies. This isn't currently enforced, but could be in the future.

depsHostHost

A list of dependencies whose host and target platforms match the new derivation's host platform. 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.

buildInputs

A list of dependencies whose host platform and target platform match the new derivation's. 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.

depsTargetTarget

A list of dependencies whose host platform matches the new derivation's target platform. 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.

depsBuildBuildPropagated

The propagated equivalent of **depsBuildBuild**. This perhaps never ought to be used, but it is included for consistency [see below for the others].

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.

depsBuildTargetPropagated

The propagated equivalent of `depsBuildTarget`. This is prefixed for the same reason of alerting potential users.

depsHostHostPropagated

The propagated equivalent of `depsHostHost`.

propagatedBuildInputs

The propagated equivalent of `buildInputs`. This would be called `depsHostTargetPropagated` but for historical continuity.

depsTargetTargetPropagated

The propagated equivalent of `depsTargetTarget`. This is prefixed for the same reason of alerting potential users.

Attributes

Variables affecting `stdenv` initialisation

Attributes affecting build properties

Special variables

v: stable -

Fixed-point arguments of `mkDerivation`

Variables affecting `stdenv` initialisation

NIX_DEBUG

A number between 0 and 7 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.

Attributes affecting build properties

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`.

Special variables

passthru

This is an attribute set which can be filled with arbitrary values. For example:

```
passthru = {
  foo = "bar";
  baz = {
    value1 = 4;
    value2 = 5;
```

v: stable -

```
};  
}
```

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.

passthru.updateScript

A script to be run by `maintainers/scripts/update.nix` when the package is matched. The attribute can contain one of the following:

- 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=$(curl -sI https://zoom.us/client/latest/zoom_x86_64.tar.xz |  
    update-source-version zoom-us "$version"  
  );
```

- a list, a script followed by arguments to be passed to it:

```
passthru.updateScript = [ ../../update.sh pname "--requested-release=unstable" ];
```

- an attribute set containing:

- **command** – a string or list in the [format expected by passthru.updateScript](#).
- **attrPath** (optional) – a string containing the canonical attribute path for the package. If present, it will be passed to the update script instead of the attribute path on which the package was discovered during Nixpkgs traversal.
- **supportedFeatures** (optional) – a list of the [extra features](#) the script supports.

```
passthru.updateScript = {  
    command = [ ../../update.sh pname ];  
    attrPath = pname;  
    supportedFeatures = [ ... ];  
};
```

Tip

A common pattern is to use the [nix-update-script](#) attribute provided in Nixpkgs, which runs [nix-update](#):

```
passthru.updateScript = nix-update-script {};
```

For simple packages, this is often enough, and will ensure that the package is updated automatically by [nixpkgs-update](#) when a new version is released. The [update bot](#) runs periodically to attempt to automatically update packages, and will run [passthru.updateScript](#) if set. While not strictly necessary if the project is listed on [Repology](#), using [nix-update-script](#) allows the package to update via many more sources (e.g. GitHub releases).

How update scripts are executed?

v: stable -

Update scripts are to be invoked by `maintainers/scripts/update.nix`. You can run `nix-shell maintainers/scripts/update.nix` in the root of Nixpkgs repository for information on how to use it. `update.nix` offers several modes for selecting packages to update (e.g. select by attribute path, traverse Nixpkgs and filter by maintainer, etc.), and it will execute update scripts for all matched packages that have an `updateScript` attribute.

Each update script will be passed the following environment variables:

- `UPDATE_NIX_NAME` – content of the `name` attribute of the updated package.
- `UPDATE_NIX_PNAME` – content of the `pname` attribute of the updated package.
- `UPDATE_NIX_OLD_VERSION` – content of the `version` attribute of the updated package.
- `UPDATE_NIX_ATTR_PATH` – attribute path the `update.nix` discovered the package on (or the canonical `attrPath` when available). Example: `pantheon.elementary-terminal`

Note

An update script will be usually run from the root of the Nixpkgs repository but you should not rely on that. Also note that `update.nix` executes update scripts in parallel by default so you should avoid running `git commit` or any other commands that cannot handle that.

Tip

While update scripts should not create commits themselves, `maintainers/scripts/update.nix` supports automatically creating commits when running it with `--argstr commit true`. If you need to customize commit message, you can have the update script implement `commit` feature.

Supported features

commit

This feature allows update scripts to ask `update.nix` to create Git commits.

When support of this feature is declared, whenever the update script exits with 0 return sta v: stable -

expected to print a JSON list containing an object described below for each updated attribute to standard output.

When `update.nix` is run with `--argstr commit true` arguments, it will create a separate commit for each of the objects. An empty list can be returned when the script did not update any files, for example, when the package is already at the latest version.

The commit object contains the following values:

- `attrPath` – a string containing attribute path.
- `oldVersion` – a string containing old version.
- `newVersion` – a string containing new version.
- `files` – a non-empty list of file paths (as strings) to add to the commit.
- `commitBody` (optional) – a string with extra content to be appended to the default commit message (useful for adding changelog links).
- `commitMessage` (optional) – a string to use instead of the default commit message.

If the returned array contains exactly one object (e.g. `[{}]`), all values are optional and will be determined automatically.

Example 231. Standard output of an update script using commit feature

```
[  
 {  
   "attrPath": "volume_key",  
   "oldVersion": "0.3.11",  
   "newVersion": "0.3.12",  
   "files": [  
     "/path/to/nixpkgs/pkgs/development/libraries/volume-key/default.nix"  
   ]  
 }  
]
```

v: stable -

Fixed-point arguments of `mkDerivation`

If you pass a function to `mkDerivation`, it will receive as its argument the final arguments, including the overrides when reinvoked via `overrideAttrs`. For example:

```
mkDerivation (finalAttrs: {
  pname = "hello";
  withFeature = true;
  configureFlags =
    lib.optionals finalAttrs.withFeature ["--with-feature"];
})
```

Note that this does not use the `rec` keyword to reuse `withFeature` in `configureFlags`. The `rec` keyword works at the syntax level and is unaware of overriding.

Instead, the definition references `finalAttrs`, allowing users to change `withFeature` consistently with `overrideAttrs`.

`finalAttrs` also contains the attribute `finalPackage`, which includes the output paths, etc.

Let's look at a more elaborate example to understand the differences between various bindings:

```
# `pkg` is the _original_ definition (for illustration purposes)
let pkg =
  mkDerivation (finalAttrs: {
    # ...
    # An example attribute
    packages = [];

    # `passthru.tests` is a commonly defined attribute.
    passthru.tests.simple = f finalAttrs.finalPackage;

    # An example of an attribute containing a function
```

v: stable -

```
passthru.appendPackages = packages':  
  finalAttrs.finalPackage.overrideAttrs (newSelf: super: {  
    packages = super.packages ++ packages';  
  });  
  
  # For illustration purposes; referenced as  
  # `(pkg.overrideAttrs(x)).finalAttrs` etc in the text below.  
  passthru.finalAttrs = finalAttrs;  
  passthru.original = pkg;  
};  
in pkg
```

Unlike the `pkg` binding in the above example, the `finalAttrs` parameter always references the final attributes. For instance `(pkg.overrideAttrs(x)).finalAttrs.finalPackage` is identical to `pkg.overrideAttrs(x)`, whereas `(pkg.overrideAttrs(x)).original` is the same as the original `pkg`.

See also the section about [passthru.tests](#).

Phases

[Controlling phases](#)

[The unpack phase](#)

[The patch phase](#)

[The configure phase](#)

[The build phase](#)

[The check phase](#)

[The install phase](#)

[The fixup phase](#)

[The installCheck phase](#)

v: stable -

The distribution phase

`stdenv.mkDerivation` sets the Nix [derivation](#)'s builder to a script that loads the `stdenv setup.sh` bash library and calls `genericBuild`. Most packaging functions rely on this default builder.

This generic command invokes 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).

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`.

When overriding a phase, for example `installPhase`, it is important to start with `runHook preInstall` and end it with `runHook postInstall`, otherwise `preInstall` and `postInstall` will not be run. Even if you don't use them directly, it is good practice to do so anyways for downstream users who would want to add a `postInstall` by overriding your derivation.

While inside an interactive `nix-shell`, if you wanted to run all phases in the order they would be run in an actual build, you can invoke `genericBuild` yourself.

Controlling phases

There are a number of variables that control what phases are executed and in what order:

Variables affecting phase control

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 unpacl`

v: stable -

```
patchPhase $preConfigurePhases configurePhase $preBuildPhases buildPhase  
checkPhase $preInstallPhases installPhase fixupPhase installCheckPhase  
$preDistPhases distPhase $postPhases.
```

It is discouraged to set this variable, as it is easy to miss some important functionality hidden in some of the less obviously needed phases (like `fixupPhase` which patches the shebang of scripts). Usually, if you just want to add a few phases, it's more convenient to set one of the variables below (such as `preInstallPhases`).

prePhases

Additional phases executed before any of the default phases.

preConfigurePhases

Additional phases executed just before the configure phase.

preBuildPhases

Additional phases executed just before the build phase.

preInstallPhases

Additional phases executed just before the install phase.

preFixupPhases

Additional phases executed just before the fixup phase.

preDistPhases

Additional phases executed just before the distribution phase.

postPhases

v: stable -

Additional phases executed after any of the default phases.

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:

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`).

Zip files

Zip files are unpacked using `unzip`. However, `unzip` is not in the standard environment, so you should add it to `nativeBuildInputs` yourself.

Directories in the Nix store

These are 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).

Variables controlling the unpack phase

`srcs / src`

The list of source files or directories to be unpacked or copied. One of these must be set. Note that if you use `srcs`, you should also set `sourceRoot` or `setSourceRoot`.

`sourceRoot`

After unpacking all of `src` and `srcs`, if neither of `sourceRoot` and `setSourceRoot` are set, `unpackPhase` of the generic builder checks that the unpacking produced a single director v: stable - ;

the current working directory into it.

If `unpackPhase` produces multiple source directories, you should set `sourceRoot` to the name of the intended directory. You can also set `sourceRoot = ".;"`; if you want to control it yourself in a later phase.

For example, if you want your build to start in a sub-directory inside your sources, and you are using `fetchzip`-derived `src` (like `fetchFromGitHub` or similar), you need to set `sourceRoot = "${src.name}/my-sub-directory"`.

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`.

For example, if you are using `fetchurl` on an archive file that gets unpacked into a single directory the name of which changes between package versions, and you want your build to start in its sub-directory, you need to set `setSourceRoot = "sourceRoot=$(echo */my-sub-directory)"`; or in the case of multiple sources, you could use something more specific, like `setSourceRoot = "sourceRoot=$(echo ${pname}-*/my-sub-directory)"`.

preUnpack

Hook executed at the start of the unpack phase.

postUnpack

Hook executed at the end of the unpack phase.

dontUnpack

Set to true to skip the unpack phase.

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.

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.

The patch phase

The patch phase applies the list of patches defined in the `patches` variable.

Variables controlling the patch phase

dontPatch

Set to true to skip the patch phase.

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)`.

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.

prePatch

Hook executed at the start of the patch phase.

postPatch

Hook executed at the end of the patch phase.

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.

Variables controlling the configure phase

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`).

configureFlags

A list of strings passed as additional arguments to the configure script.

dontConfigure

Set to true to skip the configure phase.

configureFlagsArray

A shell array containing additional arguments passed to the configure script. You must use this instead of `configureFlags` if the arguments contain spaces.

dontAddPrefix

By default, the flag `--prefix=$prefix` is added to the configure flags. If this is undesirable, set this variable to true.

prefix

The prefix under which the package must be installed, passed via the `--prefix` option to the configure script. It defaults to `$out`.

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.

dontAddStaticConfigureFlags

By default, when building statically, stdenv will try to add build system appropriate configure flags to try to enable static builds.

If this is undesirable, set this variable to true.

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.

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 [\[5\]](#). If this is undesirable, set this variable to true.

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. It is automatically set to true when building statically, for example through `pkgsStatic`.

v: stable -

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. [\[6\]](#)

preConfigure

Hook executed at the start of the configure phase.

postConfigure

Hook executed at the end of the configure phase.

The build phase

The build phase is responsible for actually building the package (e.g. compiling it). The default `buildPhase` 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.

Variables controlling the build phase

dontBuild

Set to true to skip the build phase.

makefile

The file name of the Makefile.

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.

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.

buildFlags / buildFlagsArray

A list of strings passed as additional flags to `make`. Like `makeFlags` and `makeFlagsArray`, but only used by the build phase.

preBuild

Hook executed at the start of the build phase.

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.

The check phase

The check phase checks whether the package was built correctly by running its test suite. The default `checkPhase` calls `make $checkTarget`, but only if the [`doCheck` variable](#) is enabled.

Variables controlling the check phase

`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.

`makeFlags / makeFlagsArray / makefile`

See the [`build phase`](#) for details.

`checkTarget`

The `make` target that runs the tests. If unset, use `check` if it exists, otherwise `test`; if neither is found, do nothing.

`checkFlags / checkFlagsArray`

A list of strings passed as additional flags to `make`. Like `makeFlags` and `makeFlagsArr`:
v: stable -

used by the check phase.

checkInputs

A list of host dependencies used by the phase, usually libraries linked into executables built during tests. This gets included in `buildInputs` when `doCheck` is set.

nativeCheckInputs

A list of native dependencies used by the phase, notably tools needed on `$PATH`. This gets included in `nativeBuildInputs` when `doCheck` is set.

preCheck

Hook executed at the start of the check phase.

postCheck

Hook executed at the end of the check phase.

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`.

Variables controlling the install phase

dontInstall

Set to true to skip the install phase.

makeFlags / makeFlagsArray / makefile

See the [build phase](#) for details.

v: stable -

installTargets

The make targets that perform the installation. Defaults to `install`. Example:

```
installTargets = "install-bin install-doc";
```

installFlags / installFlagsArray

A list of strings passed as additional flags to `make`. Like `makeFlags` and `makeFlagsArray`, but only used by the install phase.

preInstall

Hook executed at the start of the install phase.

postInstall

Hook executed at the end of the install phase.

The fixup phase

The fixup phase performs (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`. See [the section called “patch-shebangs.sh”](#) for details.

Variables controlling the fixup phase

dontFixup

Set to true to skip the fixup phase.

dontStrip

If set, libraries and executables are not stripped. By default, they are.

dontStripHost

Like `dontStrip`, but only affects the `strip` command targeting the package's host platform. Useful when supporting cross compilation, but otherwise feel free to ignore.

dontStripTarget

Like `dontStrip`, but only affects the `strip` command targeting the packages' target platform. Useful when supporting cross compilation, but otherwise feel free to ignore.

dontMoveSbin

If set, files in `$out/sbin` are not moved to `$out/bin`. By default, they are.

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.

stripAllListTarget

Like `stripAllList`, but only applies to packages' target platform. By default, it's empty. Useful when supporting cross compilation.

v: stable -

stripAllFlags

Flags passed to the `strip` command applied to the files in the directories listed in `stripAllList`.

Defaults to `-s` (i.e. `--strip-all`).

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`.

stripDebugListTarget

Like `stripDebugList`, but only applies to packages' target platform. By default, it's empty. Useful when supporting cross compilation.

stripDebugFlags

Flags passed to the `strip` command applied to the files in the directories listed in `stripDebugList`.

Defaults to `-S` (i.e. `--strip-debug`).

stripExclude

A list of filenames or path patterns to avoid stripping. A file is excluded if its name or path (from the derivation root) matches.

This example prevents all `*.rlib` files from being stripped:

```
stdenv.mkDerivation {  
  # ...  
  stripExclude = [ "*.rlib" ]  
}
```

This example prevents files within certain paths from being stripped:

v: stable -

```
stdenv.mkDerivation {  
  # ...  
  stripExclude = [ "lib/modules/*/build/*" ]  
}
```

dontPatchELF

If set, the `patchelf` command is not used to remove unnecessary RPATH entries. Only applies to Linux.

dontPatchShebangs

If set, scripts starting with `#!` do not have their interpreter paths rewritten to paths in the Nix store. See [the section called “patch-shebangs.sh”](#) on how patching shebangs works.

dontPruneLibtoolFiles

If set, libtool `.la` files associated with shared libraries won’t have their `dependency_libs` field cleared.

forceShare

The list of directories that must be moved from `$out` to `$out/share`. Defaults to `man doc info`.

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`.

prefixup

Hook executed at the start of the fixup phase.

v: stable -

postFixup

Hook executed at the end of the fixup phase.

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 `<XXXXXXX...>` 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.

Example 232. Enable debug symbols for use with GDB

To make GDB find debug information for the `socat` package and its dependencies, you can use the following `shell.nix`:

```
let
  pkgs = import ./. {
    config = {};
    overlays = [
      (final: prev: {
        ncurses = prev.ncurses.overrideAttrs { separateDebugInfo = true;
        readline = prev.readline.overrideAttrs { separateDebugInfo = true
        })
      });
    ];
  };

  myDebugInfoDirs = pkgs.symlinkJoin {
    name = "myDebugInfoDirs";
    paths = with pkgs; [
      glibc.debug
      ncurses.debug
      openssl.debug
      readline.debug
    ];
  };

```

v: stable -

```
];
};

in

pkgs.mkShell {

  NIX_DEBUG_INFO_DIRS = "${pkgs.lib.getLib myDebugInfoDirs}/lib/debug";

  packages = [
    pkgs.gdb
    pkgs.socat
  ];

  shellHook = ''
    ${pkgs.lib.getBin pkgs.gdb}/bin/gdb ${pkgs.lib.getBin pkgs.socat}/b:
  '';
}

}
```

This setup works as follows:

- Add overlays to the package set, since debug symbols are disabled for **ncurses** and **readline** by default.
- Create a derivation to combine all required debug symbols under one path with symlinkJoin.
- Set the environment variable **NIX_DEBUG_INFO_DIRS** in the shell. Nixpkgs patches **gdb** to use it for looking up debug symbols.
- Run **gdb** on the **socat** binary on shell startup in the shellHook. Here we use lib.getBin to ensure that the correct derivation output is selected rather than the default one.

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`.

It is often better to add tests that are not part of the source distribution to `passthru.tests` -

section called “[tests](#)”). This avoids adding overhead to every build and enables us to run them independently.

Variables controlling the installCheck phase

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.

installCheckTarget

The make target that runs the install tests. Defaults to `installcheck`.

installCheckFlags / installCheckFlagsArray

A list of strings passed as additional flags to `make`. Like `makeFlags` and `makeFlagsArray`, but only used by the installCheck phase.

installCheckInputs

A list of host dependencies used by the phase, usually libraries linked into executables built during tests. This gets included in `buildInputs` when `doInstallCheck` is set.

nativeInstallCheckInputs

A list of native dependencies used by the phase, notably tools needed on `$PATH`. This gets included in v: stable -

`nativeBuildInputs` when `doInstallCheck` is set.

preInstallCheck

Hook executed at the start of the `installCheck` phase.

postInstallCheck

Hook executed at the end of the `installCheck` phase.

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.

Variables controlling the distribution phase

doDist

If set, the distribution phase is executed.

distTarget

The make target that produces the distribution. Defaults to `dist`.

distFlags / distFlagsArray

Additional flags passed to `make`.

tarballs

The names of the source distribution files to be copied to `$out/tarballs/`. It can contain shell wildcards. The default is `*.tar.gz`.

v: stable -

don'tCopyDist

If set, no files are copied to `$out/tarballs/`.

preDist

Hook executed at the start of the distribution phase.

postDist

Hook executed at the end of the distribution phase.

Shell functions and utilities

[makeWrapper <executable> <wrapperfile> <args>](#)

[remove-references-to -t <storepath> \[-t <storepath> ... \] <file> ...](#)

[substitute <infile> <outfile> <subs>](#)

[substituteInPlace <multiple files> <subs>](#)

[substituteAll <infile> <outfile>](#)

[substituteAllInPlace <file>](#)

[stripHash <path>](#)

[wrapProgram <executable> <makeWrapperArgs>](#)

[prependToVar <variableName> <elements...>](#)

[appendToVar <variableName> <elements...>](#)

The standard environment provides a number of useful functions.

makeWrapper <executable> <wrapperfile> <args>

Constructs a wrapper for a program with various possible arguments. It is defined as part of 2 setup-hooks named `makeWrapper` and `makeBinaryWrapper` that implement the same bash functions. Hence, to use it you have to add `makeWrapper` to your `nativeBuildInputs`. Here's an example usage:

```
# adds `FOOBAR=bar` to `$out/bin/foo`'s environment
makeWrapper $out/bin/foo $wrapperfile --set FOOBAR bar

# Prefixes the binary paths of `hello` and `git`
# and suffixes the binary path of `xdg-utils`.
# 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 [ hello git ]} \
  --suffix PATH : ${lib.makeBinPath [ xdg-utils ]}
```

Packages may expect or require other utilities to be available at runtime. `makeWrapper` can be used to add packages to a `PATH` environment variable local to a wrapper.

Use `--prefix` to explicitly set dependencies in `PATH`.

Note

`--prefix` essentially hard-codes dependencies into the wrapper. They cannot be overridden without rebuilding the package.

If dependencies should be resolved at runtime, use `--suffix` to append fallback values to `PATH`.

There's many more kinds of arguments, they are documented in `nixpkgs/pkgs/build-support/setup-hooks/make-wrapper.sh` for the `makeWrapper` implementation and in `nixpkgs/pkgs/build-support/setup-hooks/make-binary-wrapper/make-binary-wrapper.sh` for the `makeBinaryWrapper` implementation.

`wrapProgram` is a convenience function you probably want to use most of the time, imple

v: stable -

both `makeWrapper` and `makeBinaryWrapper`.

Using the `makeBinaryWrapper` implementation is usually preferred, as it creates a tiny *compiled* wrapper executable, that can be used as a shebang interpreter. This is needed mostly on Darwin, where shebangs cannot point to scripts, [due to a limitation with the execve-syscall](#). Compiled wrappers generated by `makeBinaryWrapper` can be inspected with `less <path-to-wrapper>` - by scrolling past the binary data you should be able to see the shell command that generated the executable and there see the environment variables that were injected into the wrapper.

remove-references-to -t <storepath> [-t <storepath> ...] <file> ...

Removes the references of the specified files to the specified store files. This is done without changing the size of the file by replacing the hash by `eeeeeeeeeeeeeeeeeeeeeeeeeeee`, and should work on compiled executables. This is meant to be used to remove the dependency of the output on inputs that are known to be unnecessary at runtime. Of course, reckless usage will break the patched programs. To use this, add `removeReferencesTo` to `nativeBuildInputs`.

As `remove-references-to` is an actual executable and not a shell function, it can be used with `find`. Example removing all references to the compiler in the output:

```
postInstall = ''
  find "$out" -type f -exec remove-references-to -t ${stdenv.cc} '{}' +
'';
```

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:

--replace <s1> <s2>

Replace every occurrence of the string `<s1>` by `<s2>`.

--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.

--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
```

substituteInPlace <multiple files> <subs>

Like `substitute`, but performs the substitutions in place on the files passed.

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
```

v: stable -

```
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).

substituteAllInPlace <file>

Like `substituteAll`, but performs the substitutions in place on the file `<file>`.

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)
```

wrapProgram <executable> <makeWrapperArgs>

Convenience function for `makeWrapper` that replaces `<executable>` with a wrapper that executes the original program. It takes all the same arguments as `makeWrapper`, except for `--inherit-argv0` (used by the `makeBinaryWrapper` implementation) and `--argv0` (used by both `makeWrapper` and `makeBinaryWrapper` wrapper implementations).

If you will apply it multiple times, it will overwrite the wrapper file and you will end up with d v: stable -

wrapping, which should be avoided.

prependToVar <variableName> <elements...>

Prepend elements to a variable.

Example:

```
$ configureFlags="--disable-static"
$ prependToVar configureFlags --disable-dependency-tracking --enable-foo
$ echo $configureFlags
--disable-dependency-tracking --enable-foo --disable-static
```

appendToVar <variableName> <elements...>

Append elements to a variable.

Example:

```
$ configureFlags="--disable-static"
$ appendToVar configureFlags --disable-dependency-tracking --enable-foo
$ echo $configureFlags
--disable-static --disable-dependency-tracking --enable-foo
```

Package setup hooks

[move-docs.sh](#)

[compress-man-pages.sh](#)

[strip.sh](#)

[patch-shebangs.sh](#)

[audit-tmpdir.sh](#)

[multiple-outputs.sh](#)

[move-sbin.sh](#)

[move-lib64.sh](#)

[move-systemd-user-units.sh](#)

[set-source-date-epoch-to-latest.sh](#)

[Bintools Wrapper and hook](#)

[CC Wrapper and hook](#)

[Other hooks](#)

[Compiler and Linker wrapper 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 latter isn't. For example, if a derivation path is mentioned more than once, Nix itself doesn't care and 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 t . . . v: stable -

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` or when using a custom builder that has `source $stdenv/setup`. Some of these are platform specific, so they may run on Linux but not Darwin or vice-versa.

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.

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.

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.

patch-shebangs.sh

This setup hook patches installed scripts to add Nix store paths to their shebang interpreter as found in the build environment. The [shebang](#) line tells a Unix-like operating system which interpreter to use to execute the script's contents.

Note

The [generic builder](#) populates PATH from inputs of the derivation.

Invocation

Multiple paths can be specified.

```
patchShebangs [--build | --host] PATH...
```

Flags

--build

Look up commands available at build time

v: stable -

--host

Look up commands available at run time

Examples

```
patchShebangs --host /nix/store/<hash>-hello-1.0/bin
```

```
patchShebangs --build configure
```

`#!/bin/sh` will be rewritten to `#!/nix/store/<hash>-some-bash/bin/sh`.

`#!/usr/bin/env` gets special treatment: `#!/usr/bin/env python` is rewritten to `/nix/store/<hash>/bin/python`.

Interpreter paths that point to a valid Nix store location are not changed.

Note

A script file must be marked as executable, otherwise it will not be considered.

This mechanism ensures that the interpreter for a given script is always found and is exactly the one specified by the build.

It can be disabled by setting [dontPatchShebangs](#):

```
stdenv.mkDerivation {  
  # ...  
  dontPatchShebangs = true;  
  # ...  
}
```

The file [patch-shebangs.sh](#) defines the [patchShebangs](#) function. It is used to implement [patchShebangsAuto](#), the [setup hook](#) that is registered to run during the [fixup phase](#) by default.

If you need to run [patchShebangs](#) at build time, it must be called explicitly within [one of t](#) v: stable -

[phases.](#)

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.

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 [Multiple-output packages](#) for more information.

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.

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.

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.

This hook only runs when compiling for Linux.

set-source-date-epoch-to-latest.sh

v: stable -

This sets `SOURCE_DATE_EPOCH` to the modification time of the most recent file.

Bintools Wrapper and hook

The Bintools Wrapper wraps the binary utilities for a bunch of miscellaneous purposes. These are GNU Binutils when targeting 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. ^[7] `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 w/ v: stable -

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"`.

CC Wrapper and hook

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 elaborate comments describing the exact 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.

Other hooks

Many other packages provide hooks, that are not part of `stdenv`. You can find these in the [Hooks Reference](#).

Compiler and Linker wrapper hooks

If the file `${cc}/nix-support/cc-wrapper-hook` exists, it will be run at the end of the [compiler wrapper](#). If the file `${binutils}/nix-support/post-link-hook` exists, it will be run at the end of the linker wrapper. These hooks allow a user to inject code into the wrappers. As an example these hooks can be used to extract `extraBefore`, `params` and `extraAfter` which store all th v: stable -

line arguments passed to the compiler and linker respectively.

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.

Hardening in Nixpkgs

[Hardening flags enabled by default](#)

[Hardening flags disabled by default](#)

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.

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](#).

Hardening flags enabled by default

The following flags are enabled by default and might require disabling with `hardeningDisable` if the program to package is incompatible.

format

v: stable -

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.c
    printf(help_message);
               ^
cc1plus: some warnings being treated as errors
```

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/firr
```

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, crusty 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 `env.NIX_CFLAGS_COMPILE` to `-Wv: stable -`

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

strup.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 attribut
```

pic

Adds the `-fPIC` compiler options. This option 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 in 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'
```

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 g_v stable -

as if signed integer overflows were defined.

This flag should not trigger any build or runtime errors.

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.

bindnow

Adds the `-z now` 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
```

Hardening flags disabled by default

The following flags are disabled by default and should be enabled with `hardeningEnable` for packages that take untrusted input like network services.

pie

v: stable -

This flag is disabled by default for normal `glibc` based NixOS package builds, but enabled by default for `musl` based package builds.

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.

Static libraries need to be compiled with `-fPIE` so that executables can link them in with the `-pie` linker option. If the libraries lack `-fPIE`, you will get the error `recompile with -fPIE`.

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.^[1]

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`.^[2]

Nix itself already takes a package's transitive dependencies into account, but this propagation ensures nixpkgs-specific infrastructure like `setup hooks` also are run as if it were a propagated dependency.^[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 mu v: stable -

wrappings are needed whether or not the same underlying binaries can target multiple platforms.[\[7\]](#)

Meta-attributes

Table of Contents

[Standard meta-attributes](#)

[Licenses](#)

[Source provenance](#)

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 = with 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.

Standard meta-attributes

[description](#)

v: stable -

[longDescription](#)

[branch](#)

[homepage](#)

[downloadPage](#)

[changelog](#)

[license](#)

[maintainers](#)

[mainProgram](#)

[priority](#)

[platforms](#)

[badPlatforms](#)

[tests](#)

[timeout](#)

[hydraPlatforms](#)

[broken](#)

It is expected that each meta-attribute is one of the following:

description

A short (one-line) description of the package. This is displayed on search.nixos.org.

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.

v: stable -

Wrong: "libpng is a library that allows you to decode PNG images."

Right: "A library for decoding PNG images"

longDescription

An arbitrarily long description of the package in [CommonMark](#) Markdown.

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.

homepage

The package's homepage. Example: <https://www.gnu.org/software/hello/manual/>

downloadPage

The page where a link to the current version can be found. Example: <https://ftp.gnu.org/gnu/hello/>

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}](https://git.savannah.gnu.org/cgit/hello.git/plain/NEWS?h=v${version})"

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 attri

v: stable -

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](#).

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`” in the same pull request, and reference maintainers with `maintainers = with lib.maintainers; [alice bob]`.

mainProgram

The name of the main binary for the package. This affects the binary `nix run` executes. Example:
`"rg"`

priority

The *priority* of the package, used by `nix-env` to resolve file name conflicts between packages. See the [manual page for nix-env](#) for details. Example: `"10"` (a low-priority package).

platforms

v: stable -

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.

badPlatforms

The list of Nix [platform types](#) on which the package is known not to be buildable. Hydra will never create prebuilt binaries for these platform types, even if they are in `meta.platforms`. In general it is preferable to set `meta.platforms = lib.platforms.all` and then exclude any platforms on which the package is known not to build. For example, a package which requires dynamic linking and cannot be linked statically could use this:

```
meta.platforms = lib.platforms.all;
meta.badPlatforms = [ lib.systems.inspect.patterns.isStatic ];
```

The `lib.meta.availableOn` function can be used to test whether or not a package is available (i.e. buildable) on a given platform. Some packages use this to automatically detect the maximum set of features with which they can be built. For example, `systemd` [requires dynamic linking](#), and [has a `meta.badPlatforms` setting](#) similar to the one above. Packages which can be built with or without `systemd` support will use `lib.meta.availableOn` to detect whether or not `systemd` is available on the [hostPlatform](#) for which they are being built; if it is not available (e.g. due to a statically-linked host platform like `pkgsStatic`) this support will be disabled by default.

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 tests as values. A test is a derivation that builds when the test passes and fails to build otherwise.

You can run these tests with:

```
$ cd path/to/nixpkgs  
$ nix-build -A your-package.tests
```

Package tests

Tests that are part of the source package are often executed in the `installCheckPhase`.

Prefer `passthru.tests` for tests that are introduced in nixpkgs because:

- `passthru.tests` tests the ‘real’ package, independently from the environment in which it was built
- we can run `passthru.tests` independently
- `installCheckPhase` adds overhead to each build

For more on how to write and run package tests, see [the section called “Package tests”](#).

NixOS tests

The NixOS tests are available as `nixosTests` in parameters of derivations. For instance, the OpenSMTPD derivation includes lines similar to:

```
{ /* ... */, nixosTests }:  
{  
  # ...  
  passthru.tests = {  
    v: stable -
```

```
basic-functionality-and-dovecot-integration = nixosTests.opensmtpd;
};

}
```

NixOS tests run in a VM, so they are slower than regular package tests. For more information see [NixOS module tests](#).

Alternatively, you can specify other derivations as tests. You can make use of the optional parameter to inject the correct package without relying on non-local definitions, even in the presence of `overrideAttrs`. Here that's `finalAttrs.finalPackage`, but you could choose a different name if `finalAttrs` already exists in your scope.

`(mypkg.overrideAttrs f).passthru.tests` will be as expected, as long as the definition of `tests` does not rely on the original `mypkg` or overrides it in all places.

```
# my-package/default.nix
{ stdenv, callPackage }:
stdenv.mkDerivation (finalAttrs: {
  # ...
  passthru.tests.example = callPackage ./example.nix { my-package = finalAttrs });
})
```

```
# my-package/example.nix
{ runCommand, lib, my-package, ... }:
runCommand "my-package-test" {
  nativeBuildInputs = [ my-package ];
  src = lib.sources.sourcesByRegex ./ [ "*.in" "*.expected" ];
}
```
my-package --help
my-package <example.in> example.actual
diff -U3 --color=auto example.expected example.actual
mkdir $out
```

```

timeout

v: stable -

A timeout (in seconds) for building the derivation. If the derivation takes longer than this time to build, Hydra will fail it 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`.

`meta` attributes are not stored in the instantiated derivation. Therefore, this setting may be lost when the package is used as a dependency. To be effective, it must be presented directly to an evaluation process that handles the `meta.timeout` attribute.

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 = [];
```

broken

If set to `true`, the package is marked as “broken”, meaning that it won’t show up in [search.nixos.org](#), and cannot be built or installed unless the environment variable [`NIXPKGS_ALLOW_BROKEN`](#) is set. Such unconditionally-broken packages should be removed from Nixpkgs eventually unless they are fixed.

The value of this attribute can depend on a package’s arguments, including `stdenv`. This means that `broken` can be used to express constraints, for example:

- Does not cross compile

```
meta.broken = !(stdenv.buildPlatform.canExecute stdenv.hostPlatform)
```

- Broken if all of a certain set of its dependencies are broken

```
meta.broken = lib.all (map (p: p.meta.broken) [ glibc musl ])
```

This makes `broken` strictly more powerful than `meta.badPlatforms`. However `meta.availableOn` currently examines only `meta.platforms` and `meta.badPlatforms`, so `meta.broken` does not influence the default values for optional dependencies.

Licenses

[lib.licenses.free, "free"](#)

[lib.licenses.unfreeRedistributable, "unfree-redistributable"](#)

[lib.licenses.unfree, "unfree"](#)

[lib.licenses.unfreeRedistributableFirmware, "unfree-redistributable-firmware"](#)

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:

lib.licenses.free, "free"

Catch-all for free software licenses not listed above.

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 build v: stable -

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.

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.

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.

Source provenance

[lib.sourceTypes.fromSource](#)

[lib.sourceTypes.binaryNativeCode](#)

[lib.sourceTypes.binaryFirmware](#)

[lib.sourceTypes.binaryBytecode](#)

The value of a package's `meta.sourceProvenance` attribute specifies the provenance of the package's derivation outputs.

If a package contains elements that are not built from the original source by a nixpkgs derivation, the `meta.sourceProvenance` attribute should be a list containing one or more value from `lib.sourceTypes` defined in [nixpkgs/lib/source-types.nix](#).

Adding this information helps users who have needs related to build transparency and supply-chain security to gain some visibility into their installed software or set policy to allow or disallow

v: stable -

based on source provenance.

The presence of a particular `sourceType` in a package's `meta.sourceProvenance` list indicates that the package contains some components falling into that category, though the *absence* of that `sourceType` does not guarantee the absence of that category of `sourceType` in the package's contents. A package with no `meta.sourceProvenance` set implies it has no *known* `sourceTypes` other than `fromSource`.

The meaning of the `meta.sourceProvenance` attribute does not depend on the value of the `meta.license` attribute.

lib.sourceTypes.fromSource

Package elements which are produced by a nixpkgs derivation which builds them from source code.

lib.sourceTypes.binaryNativeCode

Native code to be executed on the target system's CPU, built by a third party. This includes packages which wrap a downloaded AppImage or Debian package.

lib.sourceTypes.binaryFirmware

Code to be executed on a peripheral device or embedded controller, built by a third party.

lib.sourceTypes.binaryBytecode

Code to run on a VM interpreter or JIT compiled into bytecode by a third party. This includes packages which download Java `.jar` files from another source.

Multiple-output packages

Table of Contents

[Using a split package](#)

[Writing a split derivation](#)

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: **bin**, or **out**, or the first specified output; as well as **man** if that is specified.

Using a split package

v: stable -

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 [the section called “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.)

Writing a split derivation

[“Binaries first”](#)

[File type groups](#)

[Common caveats](#)

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

v: stable -

There is a special handling of the `debug` output, described at the section called [`separateDebugInfo`](#).

“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. `${glibc}/lib/ld-linux-x86-64.so.2`). The executables provided by `glibc` can be accessed via its `bin` attribute (e.g. `${lib.getBin stdenv.cc.libc}/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).

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.

\$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.

v: stable -

\$outputBin

is meant for user-facing binaries, typically residing in `bin/`. They go to `bin` or `out` by default.

\$outputLib

is meant for libraries, typically residing in `lib/` and `libexec/`. They go to `lib` or `out` by default.

\$outputDoc

is for user documentation, typically residing in `share/doc/`. It goes to `doc` or `out` by default.

\$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.

\$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.

\$outputDevman

is for section 3 man pages, typically residing in `share/man/man[0-9]/`. They go to `devman` or `$outputMan` by default.

\$outputInfo

is for info pages, typically residing in `share/info/`. They go to `info` or `$outputBin` by default.

Common caveats

- Some configure scripts don't like some of the parameters passed by default by the framework: `v: stable -`

--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.

Cross-compilation

Table of Contents

[Introduction](#)

[Packaging in a cross-friendly manner](#)

[Cross-building packages](#)

[Cross-compilation infrastructure](#)

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.

v: stable -

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.

Packaging in a cross-friendly manner

[Platform parameters](#)

[Theory of dependency categorization](#)

[Cross packaging cookbook](#)

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 -

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 you 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!

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 [the section called “Specifying dependencies”](#). For a description of how everything is implemented, see [the section called “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 that we have dependency types of the form “`X → E`”, which means that the dependency executes on `X` and emits code for `E`; each of `X` and `E` can be `build`, `host`, or `target`, and `E` can be `*` to indicate that the dependency is not a compiler-like package.

Dependency types describe the relationships that a package has with each of its transitive dependencies. You could think of attaching one or more dependency types to each of the formal parameters at the top of a package’s `.nix` file, as well as to all of *their* formal parameters, and so on. Triples like `(foo, bar, baz)`, on the other hand, are a property of an instantiated derivation – you could attach a triple `(mips-linux, mips-linux, sparc-solaris)` to a `.drv` file in `/nix/store`.

Only nine dependency types matter in practice:

Possible dependency types

Dependency type	Dependency’s host platform	Dependency’s target platform
<code>build → *</code>	<code>build</code>	<code>(none)</code> v: stable -

Dependency type	Dependency's host platform	Dependency's target platform
build → build	build	build
build → host	build	host
build → target	build	target
host → *	host	(none)
host → host	host	host
host → target	host	target
target → *	target	(none)
target → target	target	target

Let's use `g++` as an example to make this table clearer. `g++` is a C++ compiler written in C. Suppose we are building `g++` with a (`build`, `host`, `target`) platform triple of (`foo`, `bar`, `baz`). This means we are using a `foo`-machine to build a copy of `g++` which will run on a `bar`-machine and emit binaries for the `baz`-machine.

- `g++` links against the host platform's `libc` C library, which is a "host→ *" dependency with a triple of (`bar`, `bar`, `*`). Since it is a library, not a compiler, it has no "target".
- Since `g++` is written in C, the `gcc` compiler used to compile it is a "build→ host" dependency of `g++` with a triple of (`foo`, `foo`, `bar`). This compiler runs on the build platform and emits code for the host platform.
- `gcc` links against the build platform's `libc` C library, which is a "build→ *" dependency with a triple of (`foo`, `foo`, `*`). Since it is a library, not a compiler, it has no "target".
- This `gcc` is itself compiled by an *earlier* copy of `gcc`. This earlier copy of `gcc` is a "build→ build" dependency of `g++` with a triple of (`foo`, `foo`, `foo`). This "early `gcc`" runs on the build platform and emits code for the build platform.
- `g++` is bundled with `libgcc`, which includes a collection of target-machine routines for exception handling and software floating point emulation. `libgcc` would be a "target→ *" dependency.

triple (`foo, baz, *`), because it consists of machine code which gets linked against the output of the compiler that we are building. It is a library, not a compiler, so it has no target of its own.

- `libgcc` is written in C and compiled with `gcc`. The `gcc` that compiles it will be a “build→target” dependency with triple (`foo, foo, baz`). It gets compiled *and run* at `g++`-build-time (on platform `foo`), but must emit code for the `baz`-platform.
- `g++` allows inline assembler code, so it depends on access to a copy of the `gas` assembler. This would be a “host→target” dependency with triple (`foo, bar, baz`).
- `g++` (and `gcc`) include a library `libgccjit.so`, which wrap the compiler in a library to create a just-in-time compiler. In nixpkgs, this library is in the `libgccjit` package; if C++ required that programs have access to a JIT, `g++` would need to add a “target→target” dependency for `libgccjit` with triple (`foo, baz, baz`). This would ensure that the compiler ships with a copy of `libgccjit` which both executes on and generates code for the `baz`-platform.
- If `g++` itself linked against `libgccjit.so` (for example, to allow compile-time-evaluated C++ expressions), then the `libgccjit` package used to provide this functionality would be a “host→host” dependency of `g++`: it is code which runs on the `host` and emits code for execution on the `host`.

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!

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" ];
```

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
```

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 ];
```

My package's testsuite needs to run host platform code.

Add the following to your `mkDerivation` invocation.

```
doCheck = stdenv.buildPlatform.canExecute stdenv.hostPlatform;
```

Package using Meson needs to run binaries for the host platform during build.

Add `mesonEmulatorHook` to `nativeBuildInputs` conditionally on if the target binaries can be executed.

e.g.

```
nativeBuildInputs = [
  meson
] ++ lib.optionals (!stdenv.buildPlatform.canExecute stdenv.hostPlatform)
  mesonEmulatorHook
];
```

Example of an error which this fixes.

```
[Errno 8] Exec format error: './gdk3-scan'
```

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
```

Note

Eventually we would like to make these platform examples an unnecessary convenience so that

```
$ nix-build '<nixpkgs>' --arg crossSystem '{ config = "<arch>-<os>-<
```

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 set those configuration parameters in some vaguely sane manner on the users behalf. Issue [#34274](#) tracks this inconvenience along with its root cause in crusty 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 `*Platforms` are always non-null. `localSystem` is always non-null.

Cross-compilation infrastructure

[Implementation of dependencies](#)

[Bootstrapping](#)

Implementation of dependencies

The categories of dependencies developed in [the section called “Theory of dependency categorization”](#) are specified as lists of derivations given to `mkDerivation`, as documented in [the section called “Specifying dependencies”](#). In short, each list of dependencies for “host → target” is called `deps<host><target>` (where `host`, and `target` values are either `build`, `host`, or `target`), with exceptions for backwards compatibility that `depsBuildHost` is instead called `nativeBuildInputs` and `depsHostTarget` is instead called `buildInputs`. Nixpkgs is now structured so that each `deps<host><target>` is automatically taken from `pkgs<host><target>`. (These `pkgs<host><target>`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 `pkgs<host><target>` 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.

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 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:

v: stable -

(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 problem: `v: stable -`

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 `.__spliced.buildHost` or `.__spliced.hostTarget`.

Platform Notes

Table of Contents

[Darwin \(macOS\)](#)

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"  
  '';  
}
```

v: stable -

- 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,$(  
}
```

- 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/7hnmbscpayzxrxixrgxvvlifzlxsdir-jc  
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 `xctool` 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.

- x86_64-darwin uses the 10.12 SDK by default, but some software is not compatible with that version of the SDK. In that case, the 11.0 SDK used by aarch64-darwin is available for use on x86_64-darwin. To use it, reference `apple_sdk_11_0` instead of `apple_sdk` in your derivation and use `pkgs.darwin.apple_sdk_11_0.callPackage` instead of `pkgs.callPackage`. On Linux, this will have the same effect as `pkgs.callPackage`, so you can use `pkgs.darwin.apple_sdk_11_0.callPackage` regardless of platform.

Build helpers

A build helper is a function that produces derivations.

Warning

This is not to be confused with the [builder](#) argument of the Nix `derivation` primitive, which refers to the executable that produces the build result, or [remote builder](#), which refers to a remote machine that could run such an executable.

Such a function is usually designed to abstract over a typical workflow for a given programming language or framework. This allows declaring a build recipe by setting a limited number of options relevant to the particular use case instead of using the `derivation` function directly.

[stdenv.mkDerivation](#) is the most widely used build helper, and serves as a basis for m v: stable -

In addition, it offers various options to customize parts of the builds.

There is no uniform interface for build helpers. [Trivial build helpers](#) and [fetchers](#) have various input types for convenience. [Language- or framework-specific build helpers](#) usually follow the style of `stdenv.mkDerivation`, which accepts an attribute set or a fixed-point function taking an attribute set.

Table of Contents

[Fetchers](#)

[Trivial build helpers](#)

[Testers](#)

[Special build helpers](#)

[Images](#)

[Hooks reference](#)

[Languages and frameworks](#)

[Packages](#)

Fetchers

Table of Contents

[Caveats](#)

[fetchurl and fetchzip](#)

[fetchpatch](#)

[fetchDebianPatch](#)

[fetchsvn](#)

[fetchgit](#)[fetchfossil](#)[fetchcvs](#)[fetchhg](#)[fetchFromGitea](#)[fetchFromGitHub](#)[fetchFromGitLab](#)[fetchFromGitiles](#)[fetchFromBitbucket](#)[fetchFromSavannah](#)[fetchFromRepoOrCz](#)[fetchFromSourcehut](#)[requireFile](#)[fetchtorrent](#)

Building software with Nix often requires downloading source code and other files from the internet. To this end, Nixpkgs provides *fetchers*: functions to obtain remote sources via various protocols and services.

Nixpkgs fetchers differ from built-in fetchers such as [builtins.fetchTarball](#):

- A built-in fetcher will download and cache files at evaluation time and produce a [store path](#). A Nixpkgs fetcher will create a ([fixed-output](#)) [derivation](#), and files are downloaded at build time.
- Built-in fetchers will invalidate their cache after [tarball-ttl](#) expires, and will require network activity to check if the cache entry is up to date. Nixpkgs fetchers only re-download if the v: stable -

hash changes or the store object is not otherwise available.

- Built-in fetchers do not use [substituters](#). Derivations produced by Nixpkgs fetchers will use any configured binary cache transparently.

This significantly reduces the time needed to evaluate the entirety of Nixpkgs, and allows [Hydra](#) to retain and re-distribute sources used by Nixpkgs in the [public binary cache](#). For these reasons, built-in fetchers are not allowed in Nixpkgs source code.

The following table shows an overview of the differences:

Fetchers	Download	Output	Cache	Re-download when
<code>builtins.fetch*</code>	evaluation time	store path	<code>/nix/store</code> , <code>~/.cache/nix</code>	<code>tarball-ttl</code> expires, cache miss in <code>~/.cache/nix</code> , output store object not in local store
<code>pkgs.fetch*</code>	build time	derivation	<code>/nix/store</code> , substituters	output store object not available

Caveats

The fact that the hash belongs to the Nix derivation output and not the file itself can lead to confusion. For example, consider the following fetcher:

```
fetchurl {
  url = "http://www.example.org/hello-1.0.tar.gz";
  hash = "sha256-lTeyxzJNQeMdu1IVdovNMtgn77jRIhSybLdMbTkf2Ww=";
};
```

A common mistake is to update a fetcher's URL, or a version parameter, without updating the hash.

```
fetchurl {
  url = "http://www.example.org/hello-1.1.tar.gz";
```

v: stable -

```
hash = "sha256-lTeyxzJNQeMdu1IVdovNMtgn77jRIhSybLdMbTkf2Ww=";  
};
```

This will reuse the old contents. Remember to invalidate the hash argument, in this case by setting the hash attribute to an empty string.

```
fetchurl {  
  url = "http://www.example.org/hello-1.1.tar.gz";  
  hash = "";  
};
```

Use the resulting error message to determine the correct hash.

```
error: hash mismatch in fixed-output derivation '/path/to/my.drv':  
      specified: sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=  
      got:     sha256-lTeyxzJNQeMdu1IVdovNMtgn77jRIhSybLdMbTkf2Ww=
```

A similar problem arises while testing changes to a fetcher's implementation. If the output of the derivation already exists in the Nix store, test failures can go undetected. The [invalidateFetcherByDrvHash](#) function helps prevent reusing cached derivations.

fetchurl and fetchzip

Two basic fetchers are `fetchurl` and `fetchzip`. Both of these have two required arguments, a URL and a hash. The hash is typically `hash`, although many more hash algorithms are supported. Nixpkgs contributors are currently recommended to use `hash`. 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";
```

v: stable -

```
    hash = "sha256-BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB=";  
};  
}
```

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

`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.

- **relative**: Similar to using `git-diff`'s `--relative` flag, only keep changes inside the specified directory, making paths relative to it.
- **stripLen**: Remove the first `stripLen` components of pathnames in the patch.
- **decode**: Pipe the downloaded data through this command before processing it as a patch.
- **extraPrefix**: Prefix pathnames by this string.
- **excludes**: Exclude files matching these patterns (applies after the above arguments).
- **includes**: Include only files matching these patterns (applies after the above arguments).
- **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 `hash` argument is changed as well.

Most other fetchers return a directory rather than a single file.

fetchDebianPatch

A wrapper around `fetchpatch`, which takes:

- `patch` and `hash`: the patch's filename, and its hash after normalization by `fetchpatch`;
- `pname`: the Debian source package's name ;
- `version`: the upstream version number ;
- `debianRevision`: the [Debian revision number](#) if applicable ;
- the `area` of the Debian archive: `main` (default), `contrib`, or `non-free`.

Here is an example of `fetchDebianPatch` in action:

```
{ lib
, fetchDebianPatch
, buildPythonPackage
}:

buildPythonPackage rec {
  pname = "pysimplesoap";
  version = "1.16.2";
  src = ...;

  patches = [
    (fetchDebianPatch {
      inherit pname version;
      debianRevision = "5";
      name = "Add-quotes-to-SOAPAction-header-in-SoapClient.patch";
      hash = "sha256-xA8Wnrpr31H8wy3zHSNfezFNjUJt1HbSXn3qUMzeKc0=";
    })
  ];
}

...
```

Patches are fetched from `sources.debian.org`, and so must come from a package version that was uploaded to the Debian archive. Packages may be removed from there once that specific version isn't in any suite anymore (stable, testing, unstable, etc.), so maintainers should use `copy-tarballs.pl` to archive the patch if it needs to be available longer-term.

fetchsvn

Used with Subversion. Expects `url` to a Subversion directory, `rev`, and `hash`.

fetchgit

Used with Git. Expects `url` to a Git repo, `rev`, and `hash`. `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.

If only parts of the repository are needed, `sparseCheckout` can be used. This will prevent git from fetching unnecessary blobs from server, see [git sparse-checkout](#) for more information:

```
{ stdenv, fetchgit }:

stdenv.mkDerivation {
  name = "hello";
  src = fetchgit {
    url = "https://...";
    sparseCheckout = [
      "directory/to/be/included"
      "another/directory"
    ];
    hash = "sha256-AAAAAAAAAAAAAAA=";
  };
}
```

v: stable -

{

fetchfossil

Used with Fossil. Expects `url` to a Fossil archive, `rev`, and `hash`.

fetchcvs

Used with CVS. Expects `cvsRoot`, `tag`, and `hash`.

fetchhg

Used with Mercurial. Expects `url`, `rev`, and `hash`.

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.

fetchFromGitea

`fetchFromGitea` expects five arguments. `domain` is the gitea server name. `owner` is a string corresponding to the Gitea user or organization that controls this repository. `repo` corresponds to the name of the software repository. These are located at the top of every Gitea 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, `hash` corresponds to the hash of the extracted directory. Again, other hash algorithms are also available but `hash` is currently preferred.

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, `hash` corresponds to the hash of the extracted directory. Again, other hash algorithms are also available but `hash` is currently preferred.

v: stable -

of the extracted directory. Again, other hash algorithms are also available, but `hash` is currently preferred.

To use a different GitHub instance, use `githubBase` (defaults to "github.com").

`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.

fetchFromGitLab

This is used with GitLab repositories. It behaves similarly to `fetchFromGitHub`, and expects `owner`, `repo`, `rev`, and `hash`.

To use a specific GitLab instance, use `domain` (defaults to "gitlab.com").

fetchFromGitiles

This is used with Gitiles repositories. The arguments expected are similar to `fetchgit`.

fetchFromBitbucket

This is used with BitBucket repositories. The arguments expected are very similar to `fetchFromGitHub` above.

fetchFromSavannah

This is used with Savannah repositories. The arguments expected are very similar to `fetchFromGitHub` above.

fetchFromRepoOrCz

This is used with repo.or.cz repositories. The arguments expected are very similar to

v: stable -

fetchFromGitHub above.

fetchFromSourcehut

This is used with sourcehut repositories. Similar to `fetchFromGitHub` above, it expects `owner`, `repo`, `rev` and `hash`, but don't forget the tilde (~) in front of the username! Expected arguments also include `vc` ("git" (default) or "hg"), `domain` and `fetchSubmodules`.

If `fetchSubmodules` is `true`, `fetchFromSourcehut` uses `fetchgit` or `fetchhg` with `fetchSubmodules` or `fetchSubrepos` set to `true`, respectively. Otherwise, the fetcher uses `fetchzip`.

requireFile

`requireFile` allows requesting files that cannot be fetched automatically, but whose content is known. This is a useful last-resort workaround for license restrictions that prohibit redistribution, or for downloads that are only accessible after authenticating interactively in a browser. If the requested file is present in the Nix store, the resulting derivation will not be built, because its expected output is already available. Otherwise, the builder will run, but fail with a message explaining to the user how to provide the file. The following code, for example:

```
requireFile {  
    name = "jdk-${version}_linux-x64_bin.tar.gz";  
    url = "https://www.oracle.com/java/technologies/javase-jdk11-downloads.l  
    hash = "sha256-1L00+F7jjT71nlKJ7HRQuUQ7kkxVYlZh//5msD8sjeI=";  
}
```

results in this error message:

Unfortunately, we cannot download file jdk-11.0.10_linux-x64_bin.tar.gz at
Please go to <https://www.oracle.com/java/technologies/javase-jdk11-downloads.html>
using either
nix-store --add-fixed sha256 jdk-11.0.10_linux-x64_bin.tar.gz
or
v: stable -

```
nix-prefetch-url --type sha256 file:///path/to/jdk-11.0.10_linux-x64_bin
```

fetchtorrent

Parameters

`fetchtorrent` expects two arguments. `url` which can either be a Magnet URI (Magnet Link) such as `magnet:?xt=urn:btih:dd8255ecdc7ca55fb0bbf81323d87062db1f6d1c` or an HTTP URL pointing to a `.torrent` file. It can also take a `config` argument which will craft a `settings.json` configuration file and give it to `transmission`, the underlying program that is performing the fetch. The available config options for `transmission` can be found [here](#)

```
{ fetchtorrent }:

fetchtorrent {
  config = { peer-limit-global = 100; };
  url = "magnet:?xt=urn:btih:dd8255ecdc7ca55fb0bbf81323d87062db1f6d1c";
  sha256 = "";
}
```

Parameters

- `url`: Magnet URI (Magnet Link) such as `magnet:?xt=urn:btih:dd8255ecdc7ca55fb0bbf81323d87062db1f6d1c` or an HTTP URL pointing to a `.torrent` file.
- `backend`: Which bittorrent program to use. Default: "`transmission`". Valid values are "`rqbit`" or "`transmission`". These are the two most suitable torrent clients for fetching in a fixed-output derivation at the time of writing, as they can be easily exited after usage. `rqbit` is written in Rust and has a smaller closure size than `transmission`, and the performance and peer discovery properties differs between these clients, requiring experimentation to decide upon which is the best v: stable -

- **config**: When using `transmission` as the `backend`, a json configuration can be supplied to `transmission`. Refer to the [upstream documentation](#) for information on how to configure.

Trivial build helpers

Table of Contents

[runCommand](#)

[runCommandCC](#)

[runCommandLocal](#)

[writeTextFile, writeText, writeTextDir, writeScript, writeScriptBin](#)

[concatTextFile, concatText, concatScript](#)

[writeShellApplication](#)

[symlinkJoin](#)

[writeReferencesToFile](#)

[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.

runCommand

`runCommand :: String -> AttrSet -> String -> Derivation`

`runCommand name drvAttrs buildCommand` returns a derivation that is built by running the specified shell commands.

`name :: String`

v: stable -

The name that Nix will append to the store path in the same way that `stdenv.mkDerivation` uses its `name` attribute.

drvAttr :: AttrSet

Attributes to pass to the underlying call to [`stdenv.mkDerivation`](#).

buildCommand :: String

Shell commands to run in the derivation builder.

Note

You have to create a file or directory `$out` for Nix to be able to run the builder successfully.

Example 233. Invocation of `runCommand`

```
(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  
''
```

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.

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 round-trip 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`.

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`.

Here are a few examples:

```
# Writes my-file to /nix/store/<store path>
writeTextFile {
```

v: stable -

```
name = "my-file";
text = ''
  Contents of File
';
}

# See also the `writeText` helper function below.

# Writes executable my-file to /nix/store/<store path>/bin/my-file
writeTextFile {
  name = "my-file";
  text = ''
    Contents of File
';
  executable = true;
  destination = "/bin/my-file";
}

# Writes contents of file to /nix/store/<store path>
writeText "my-file"
'

  Contents of File
';

# Writes contents of file to /nix/store/<store path>/share/my-file
writeTextDir "share/my-file"
'

  Contents of File
';

# Writes my-file to /nix/store/<store path> and makes executable
writeScript "my-file"
'

  Contents of File
';

# Writes my-file to /nix/store/<store path>/bin/my-file and makes executable
writeScriptBin "my-file"
'

  Contents of File
';

# Writes my-file to /nix/store/<store path> and makes executable
writeShellScript "my-file"
```

v: stable -

```
''  
  Contents of File  
'';  
# Writes my-file to /nix/store/<store path>/bin/my-file and makes executable  
writeShellScriptBin "my-file"  
''  
  Contents of File  
'';
```

concatTextFile, concatText, concatScript

These functions concatenate `files` to the Nix store in a single file. This is useful for configuration files structured in lines of text. `concatTextFile` takes an attribute set and expects two arguments, `name` and `files`. `name` corresponds to the name used in the Nix store path. `files` will be the files to be concatenated. You can also set `executable` to true to make this file have the executable bit set. `concatText` and `concatScript` are simple wrappers over `concatTextFile`.

Here are a few examples:

```
# Writes my-file to /nix/store/<store path>  
concatTextFile {  
  name = "my-file";  
  files = [ drv1 "${drv2}/path/to/file" ];  
}  
# See also the `concatText` helper function below.  
  
# Writes executable my-file to /nix/store/<store path>/bin/my-file  
concatTextFile {  
  name = "my-file";  
  files = [ drv1 "${drv2}/path/to/file" ];  
  executable = true;  
  destination = "/bin/my-file";  
}  
# Writes contents of files to /nix/store/<store path>
```

v: stable -

```
concatText "my-file" [ file1 file2 ]  
  
# Writes contents of files to /nix/store/<store path>  
concatScript "my-file" [ file1 file2 ]
```

writeShellApplication

This can be used to easily produce a shell script that has some dependencies (`runtimeInputs`). It automatically sets the `PATH` of the script to contain all of the listed inputs, sets some sanity shellopts (`errexit`, `nounset`, `pipefail`), and checks the resulting script with [shellcheck](#).

For example, look at the following code:

```
writeShellApplication {  
  name = "show-nixos-org";  
  
  runtimeInputs = [ curl w3m ];  
  
  text = ''  
    curl -s 'https://nixos.org' | w3m -dump -T text/html  
  '';  
}
```

Unlike with normal `writeShellScriptBin`, there is no need to manually write out `curl/bin/curl`, setting the `PATH` was handled by `writeShellApplication`. Moreover, the script is being checked with `shellcheck` for more strict validation.

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. Here is an example:

v: stable -

```
# adds symlinks of hello and stack to current build and prints "links added"
symlinkJoin { name = "myexample"; paths = [ pkgs.hello pkgs.stack ]; postBuild =
```

This creates a derivation with a directory structure like the following:

```
/nix/store/sgrlsr5g079a5235hy29da3mq3hv8sjmm-myexample
|-- bin
|   |-- hello -> /nix/store/qy93dp4a3rqyn2mz63fbxjg228hffwyw-hello-2.10/bin/hello
|   `-- stack -> /nix/store/6lzdpxshx78281vy056lbk553ijsdr44-stack-2.1.3/bin/stack
`-- share
    |-- bash-completion
    |   '-- completions
    |       '-- stack -> /nix/store/6lzdpxshx78281vy056lbk553ijsdr44-stack/share/bash-completion/completions/stack
    |-- fish
    |   '-- vendor_completions.d
    |       '-- stack.fish -> /nix/store/6lzdpxshx78281vy056lbk553ijsdr44-stack/share/fish/vendor_completions.d/stack.fish
...
...
```

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`.

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.

Testers

Table of Contents

[hasPkgConfigModules](#)

[testVersion](#)

[testBuildFailure](#)

[testEqualContents](#)

v: stable -

[testEqualDerivation](#)[invalidateFetcherByDrvHash](#)[runNixOSTest](#)[nixosTest](#)

This chapter describes several testing builders which are available in the `testers` namespace.

hasPkgConfigModules

Checks whether a package exposes a given list of `pkg-config` modules. If the `moduleName`s argument is omitted, `hasPkgConfigModules` will use `meta.pkgConfigModules`.

Example:

```
passthru.tests.pkg-config = testers.hasPkgConfigModules {  
    package = finalAttrs.finalPackage;  
    moduleName = [ "libfoo" ];  
};
```

If the package in question has `meta.pkgConfigModules` set, it is even simpler:

```
passthru.tests.pkg-config = testers.hasPkgConfigModules {  
    package = finalAttrs.finalPackage;  
};  
  
meta.pkgConfigModules = [ "libfoo" ];
```

testVersion

Checks the command output contains the specified version

v: stable -

Although simplistic, this test assures that the main program can run. While there's no substitute for a real test case, it does catch dynamic linking errors and such. It also provides some protection against accidentally building the wrong version, for example when using an 'old' hash in a fixed-output derivation.

Examples:

```
passthru.tests.version = testers.testVersion { package = hello; };

passthru.tests.version = testers.testVersion {
    package = seaweedfs;
    command = "weed version";
};

passthru.tests.version = testers.testVersion {
    package = key;
    command = "KeY --help";
    # Wrong '2.5' version in the code. Drop on next version.
    version = "2.5";
};

passthru.tests.version = testers.testVersion {
    package = ghr;
    # The output needs to contain the 'version' string without any prefix or suffix.
    version = "v${version}";
};
```

testBuildFailure

Make sure that a build does not succeed. This is useful for testing testers.

This returns a derivation with an override on the builder, with the following effects:

- Fail the build when the original builder succeeds
- Move `$out` to `$out/result`, if it exists (assuming `out` is the default output)

v: stable -

- Save the build log to `$out/testBuildFailure.log` (same)

Example:

```
runCommand "example" {
  failed = testers.testBuildFailure (runCommand "fail" {} ''
    echo ok-ish >$out
    echo failing though
    exit 3
  '');
} ''
  grep -F 'ok-ish' $failed/result
  grep -F 'failing though' $failed/testBuildFailure.log
  [[ 3 = $(cat $failed/testBuildFailure.exit) ]]
  touch $out
'';
```

While `testBuildFailure` is designed to keep changes to the original builder's environment to a minimum, some small changes are inevitable.

- The file `$TMPDIR/testBuildFailure.log` is present. It should not be deleted.
- `stdout` and `stderr` are a pipe instead of a tty. This could be improved.
- One or two extra processes are present in the sandbox during the original builder's execution.
- The derivation and output hashes are different, but not unusual.
- The derivation includes a dependency on `buildPackages.bash` and `expect-failure.sh`, which is built to include a transitive dependency on `buildPackages.coreutils` and possibly more. These are not added to `PATH` or any other environment variable, so they should be hard to observe.

testEqualContents

Check that two paths have the same contents.

v: stable -

Example:

```
testers.testEqualContents {
    assertion = "sed -e performs replacement";
    expected = writeText "expected" ''
        foo baz baz
    '';
    actual = runCommand "actual" {
        # not really necessary for a package that's in stdenv
        nativeBuildInputs = [ gnused ];
        base = writeText "base" ''
            foo bar baz
        '';
    } ''
        sed -e 's/bar/baz/g' $base >$out
    '';
}
```

testEqualDerivation

Checks that two packages produce the exact same build instructions.

This can be used to make sure that a certain difference of configuration, such as the presence of an overlay does not cause a cache miss.

When the derivations are equal, the return value is an empty file. Otherwise, the build log explains the difference via `nix-diff`.

Example:

```
testers.testEqualDerivation
  "The hello package must stay the same when enabling checks."
  hello
  (hello.overrideAttrs(o: { doCheck = true; }))
```

v: stable -

invalidateFetcherByDrvHash

Use the derivation hash to invalidate the output via name, for testing.

Type: `(a@{ name, ... } -> Derivation) -> a -> Derivation`

Normally, fixed output derivations can and should be cached by their output hash only, but for testing we want to re-fetch everytime the fetcher changes.

Changes to the fetcher become apparent in the `drvPath`, which is a hash of how to fetch, rather than a fixed store path. By inserting this hash into the name, we can make sure to re-run the fetcher every time the fetcher changes.

This relies on the assumption that Nix isn't clever enough to reuse its database of local store contents to optimize fetching.

You might notice that the "salted" name derives from the normal invocation, not the final derivation. `invalidateFetcherByDrvHash` has to invoke the fetcher function twice: once to get a derivation hash, and again to produce the final fixed output derivation.

Example:

```
tests.fetchgit = testers.invalidateFetcherByDrvHash fetchgit {
  name = "nix-source";
  url = "https://github.com/NixOS/nix";
  rev = "9d9dbe6ed05854e03811c361a3380e09183f4f4a";
  hash = "sha256-7DszvbCNTjpzGRmpIVAWXk20P0/XTrWZ79KS0GLrUWY=";
};
```

runNixOSTest

A helper function that behaves exactly like the NixOS `runTest`, except it also assigns this Nixpkgs package set as the `pkgs` of the test and makes the `nixpkgs.*` options read-only.

If your test is part of the Nixpkgs repository, or if you need a more general entrypoint, see ["Calling a test" in the NixOS manual](#).

v: stable -

Example:

```
pkgs testers.runNixOSTest ({ lib, ... }: {
  name = "hello";
  nodes.machine = { pkgs, ... }: {
    environment.systemPackages = [ pkgs.hello ];
  };
  testScript = ''
    machine.succeed("hello")
  '';
})
```

nixosTest

Parameter

Result

Run a NixOS VM network test using this evaluation of Nixpkgs.

NOTE: This function is primarily for external use. NixOS itself uses `make-test-python.nix` directly. Packages defined in Nixpkgs [reuse NixOS tests via nixosTests, plural](#).

It is mostly equivalent to the function `import ./make-test-python.nix` from the [NixOS manual](#), except that the current application of Nixpkgs (`pkgs`) will be used, instead of letting NixOS invoke Nixpkgs anew.

If a test machine needs to set NixOS options under `nixpkgs`, it must set only the `nixpkgs.pkgs` option.

Parameter

A [NixOS VM test network](#), or path to it. Example:

v: stable -

```
{  
  name = "my-test";  
  nodes = {  
    machine1 = { lib, pkgs, nodes, ... }: {  
      environment.systemPackages = [ pkgs.hello ];  
      services.foo.enable = true;  
    };  
    # machine2 = ...;  
  };  
  testScript = ''  
    start_all()  
    machine1.wait_for_unit("foo.service")  
    machine1.succeed("hello | foo-send")  
  '';  
}
```

Result

A derivation that runs the VM test.

Notable attributes:

- `nodes`: the evaluated NixOS configurations. Useful for debugging and exploring the configuration.
- `driverInteractive`: a script that launches an interactive Python session in the context of the `testScript`.

Special build helpers

Table of Contents

[buildFHSEnv](#)

[pkgs.makeSetupHook](#)

[pkgs.mkShell](#)

vmTools

pkgs.checkpointBuildTools

This chapter describes several special build helpers.

buildFHSEnv

`buildFHSEnv` provides a way to build and run FHS-compatible lightweight sandboxes. It creates an isolated root filesystem with the host's `/nix/store`, so its footprint in terms of disk space is quite small. This allows you 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 for instance. It uses Linux' namespaces feature to create temporary lightweight environments which are destroyed after all child processes exit, without requiring elevated privileges. It works similar to containerisation technology such as Docker or FlatPak but provides no security-relevant separation from the host system.

Accepted arguments are:

- **name** The name of the environment and the wrapper executable.
- **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.
- **multiArch** Whether to install 32bit multiPkgs into the FHSEnv in 64bit environments
- **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.

v: stable -

- **extraInstallCommands** Additional commands to be executed for finalizing the derivation with runner script.
- **runScript** A shell command to be executed inside the sandbox. It defaults to **bash**. Command line arguments passed to the resulting wrapper are appended to this command by default. This command must be escaped; i.e. "foo app" --do-stuff --with "some file". See **lib.escapeShellArgs**.
- **profile** Optional script for `/etc/profile` within the sandbox.

You can create a simple environment using a `shell.nix` like this:

```
{ pkgs ? import <nixpkgs> {} }:

(pkgs.buildFHSEnv {
  name = "simple-x11-env";
  targetPkgs = pkgs: (with pkgs; [
    udev
    alsa-lib
  ]) ++ (with pkgs.xorg; [
    libX11
    libXcursor
    libXrandr
  ]);
  multiPkgs = pkgs: (with pkgs; [
    udev
    alsa-lib
  ]);
  runScript = "bash";
}).env
```

Running `nix-shell` on it would drop you into a shell inside an FHS env where those libraries and binaries are available in FHS-compliant paths. Applications that expect an FHS structure (i.e. proprietary binaries) can run inside this environment without modification. You can build a wrapper by running your binary in `runScript`, e.g. `./bin/start.sh`. Relative paths work as expected.

Additionally, the FHS builder links all relocated gsettings-schemas (the glib setup-hook moves them to the correct locations).

share/gsettings-schemas/\${name}/glib-2.0/schemas) to their standard FHS location. This means you don't need to wrap binaries with `wrapGAppsHook`.

pkgs.makeSetupHook

[Usage](#)

[Attributes](#)

`pkgs.makeSetupHook` is a build helper that produces hooks that go in to `nativeBuildInputs`

Usage

```
pkgs.makeSetupHook {  
    name = "something-hook";  
    propagatedBuildInputs = [ pkgs.commandsomething ];  
    depsTargetTargetPropagated = [ pkgs.libsthing ];  
} ./script.sh
```

setup hook that depends on the hello package and runs hello and @shell@ is substituted with path to bash

```
pkgs.makeSetupHook {  
    name = "run-hello-hook";  
    propagatedBuildInputs = [ pkgs.hello ];  
    substitutions = { shell = "${pkgs.bash}/bin/bash"; };  
    passthru.tests.greeting = callPackage ./test { };  
    meta.platforms = lib.platforms.linux;  
} (writeScript "run-hello-hook.sh" ''  
    '#!@shell@  
    hello  
    '')
```

Attributes

- **name** Set the name of the hook.
- **propagatedBuildInputs** Runtime dependencies (such as binaries) of the hook.
- **depsTargetTargetPropagated** Non-binary dependencies.
- **meta**
- **passthru**
- **substitutions** Variables for `substituteAll`

pkgs.mkShell

Usage

Attributes

Building the shell

`pkgs.mkShell` is a specialized `stdenv.mkDerivation` that removes some repetition when using it with `nix-shell` (or `nix develop`).

Usage

Here is a common usage example:

```
{ pkgs ? import <nixpkgs> {} }:  
pkgs.mkShell {  
    packages = [ pkgs.gnumake ];  
  
    inputsFrom = [ pkgs.hello pkgs.gnutar ];  
  
    shellHook = ''  
        export DEBUG=1
```

v: stable -

```
'';  
}
```

Attributes

- **name** (default: `nix-shell`). Set the name of the derivation.
- **packages** (default: `[]`). Add executable packages to the `nix-shell` environment.
- **inputsFrom** (default: `[]`). Add build dependencies of the listed derivations to the `nix-shell` environment.
- **shellHook** (default: `" "`). Bash statements that are executed by `nix-shell`.

... all the attributes of `stdenv.mkDerivation`.

Building the shell

This derivation output will contain a text file that contains a reference to all the build inputs. This is useful in CI where we want to make sure that every derivation, and its dependencies, build properly. Or when creating a GC root so that the build dependencies don't get garbage-collected.

vmTools

[vmTools.createEmptyImage](#)

[vmTools.runInLinuxVM](#)

[vmTools.extractFs](#)

[vmTools.extractMTDfs](#)

[vmTools.runInLinuxImage](#)

[vmTools.makeImageTestScript](#)

[vmTools.diskImageFuns](#)

[vmTools.diskImageExtraFuns](#)

v: stable -

vmTools.diskImages

A set of VM related utilities, that help in building some packages in more advanced scenarios.

vmTools.createEmptyImage

A bash script fragment that produces a disk image at `destination`.

Attributes

- `size`. The disk size, in MiB.
- `fullName`. Name that will be written to `${destination}/nix-support/full-name`.
- `destination` (optional, default `$out`). Where to write the image files.

vmTools.runInLinuxVM

Run a derivation in a Linux virtual machine (using Qemu/KVM). By default, there is no disk image; the root filesystem is a `tmpfs`, and the Nix store is shared with the host (via the [9P protocol](#)). Thus, any pure Nix derivation should run unmodified.

If the build fails and Nix is run with the `-K`/`--keep-failed` option, a script `run-vm` will be left behind in the temporary build directory that allows you to boot into the VM and debug it interactively.

Attributes

- `preVM` (optional). Shell command to be evaluated *before* the VM is started (i.e., on the host).
- `memSize` (optional, default 512). The memory size of the VM in MiB.
- `diskImage` (optional). A file system image to be attached to `/dev/sda`. Note that currently we expect the image to contain a filesystem, not a full disk image with a partition table etc.

Examples

v: stable -

Build the derivation hello inside a VM:

```
{ pkgs }: with pkgs; with vmTools;  
runInLinuxVM hello
```

Build inside a VM with extra memory:

```
{ pkgs }: with pkgs; with vmTools;  
runInLinuxVM (hello.overrideAttrs (_: { memSize = 1024; }))
```

Use VM with a disk image (implicitly sets `diskImage`, see [vmTools.createEmptyImage](#)):

```
{ pkgs }: with pkgs; with vmTools;  
runInLinuxVM (hello.overrideAttrs (_: {  
    preVM = createEmptyImage {  
        size = 1024;  
        fullName = "vm-image";  
    };  
}))
```

vmTools.extractFs

Takes a file, such as an ISO, and extracts its contents into the store.

Attributes

- **file**. Path to the file to be extracted. Note that currently we expect the image to contain a filesystem, not a full disk image with a partition table etc.
- **fs** (optional). Filesystem of the contents of the file.

Examples

Extract the contents of an ISO file:

v: stable -

```
{ pkgs }: with pkgs; with vmTools;  
extractFs { file = ./image.iso; }
```

vmTools.extractMTDfs

Like [the section called “`vmTools.extractFs`”](#), but it makes use of a [Memory Technology Device \(MTD\)](#).

vmTools.runInLinuxImage

Like [the section called “`vmTools.runInLinuxVM`”](#), but instead of using `stdenv` from the Nix store, run the build using the tools provided by `/bin`, `/usr/bin`, etc. from the specified filesystem image, which typically is a filesystem containing a [FHS](#)-based Linux distribution.

vmTools.makeImageTestScript

Generate a script that can be used to run an interactive session in the given image.

Examples

Create a script for running a Fedora 27 VM:

```
{ pkgs }: with pkgs; with vmTools;  
makeImageTestScript diskImages.fedora27x86_64
```

Create a script for running an Ubuntu 20.04 VM:

```
{ pkgs }: with pkgs; with vmTools;  
makeImageTestScript diskImages.ubuntu2004x86_64
```

vmTools.diskImageFuns

A set of functions that build a predefined set of minimal Linux distributions images.

v: stable -

Images

- Fedora
 - `fedora26x86_64`
 - `fedora27x86_64`
- CentOS
 - `centos6i386`
 - `centos6x86_64`
 - `centos7x86_64`
- Ubuntu
 - `ubuntu1404i386`
 - `ubuntu1404x86_64`
 - `ubuntu1604i386`
 - `ubuntu1604x86_64`
 - `ubuntu1804i386`
 - `ubuntu1804x86_64`
 - `ubuntu2004i386`
 - `ubuntu2004x86_64`
 - `ubuntu2204i386`
 - `ubuntu2204x86_64`
- Debian
 - `debian10i386`
 - `debian10x86_64`

v: stable -

- `debian11i386`
- `debian11x86_64`

Attributes

- `size` (optional, defaults to `4096`). The size of the image, in MiB.
- `extraPackages` (optional). A list names of additional packages from the distribution that should be included in the image.

Examples

8GiB image containing Firefox in addition to the default packages:

```
{ pkgs }: with pkgs; with vmTools;  
diskImageFuns.ubuntu2004x86_64 { extraPackages = [ "firefox" ]; size = 8192 }
```

vmTools.diskImageExtraFuns

Shorthand for `vmTools.diskImageFuns.<attr> { extraPackages = ... }.`

vmTools.diskImages

Shorthand for `vmTools.diskImageFuns.<attr> { }.`

pkgs.checkpointBuildTools

Example

`pkgs.checkpointBuildTools` provides a way to build derivations incrementally. It consists of two functions to make checkpoint builds using Nix possible.

For hermeticity, Nix derivations do not allow any state to be carried over between builds, m

v: stable -

transparent incremental build within a derivation impossible.

However, we can tell Nix explicitly what the previous build state was, by representing that previous state as a derivation output. This allows the passed build state to be used for an incremental build.

To change a normal derivation to a checkpoint based build, these steps must be taken:

- apply `prepareCheckpointBuild` on the desired derivation, e.g.

```
checkpointArtifacts = (pkgs.checkpointBuildTools.prepareCheckpointBuild pl
```

- change something you want in the sources of the package, e.g. use a source override:

```
changedVBox = pkgs.virtualbox.overrideAttrs (old: {  
    src = path/to/vbox/sources;  
});
```

- use `mkCheckpointBuild changedVBox checkpointArtifacts`
- enjoy shorter build times

Example

```
{ pkgs ? import <nixpkgs> {} }:  
let  
  inherit (pkgs.checkpointBuildTools)  
    prepareCheckpointBuild  
    mkCheckpointBuild  
    ;  
  helloCheckpoint = prepareCheckpointBuild pkgs.hello;  
  changedHello = pkgs.hello.overrideAttrs (_: {  
    doCheck = false;  
    patchPhase = ''  
      sed -i 's/Hello, world!/Hello, Nix!/g' src/hello.c  
    '';  
  });
```

v: stable -

```
in mkCheckpointBuild changedHello helloCheckpoint
```

Images

Table of Contents

[pkgs.appimageTools](#)

[pkgs.dockerTools](#)

[pkgs.ociTools](#)

[pkgs.snapTools](#)

[pkgs.portableService](#)

[`<nixpkgs/nixos/lib/make-disk-image.nix>`](#)

[pkgs.mkBinaryCache](#)

This chapter describes tools for creating various types of images.

pkgs.appimageTools

[AppImage formats](#)

[Wrapping](#)

`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.

v: stable -

in the future.

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 9660 CD-ROM filesystem, dynamically linked, buildtime timestamp 0x5f5e5f5e, entry 0x401000
spot sensor temperature 0.000000, unit celsius, color scheme 0, calibration 0, calibration timestamp 0x5f5e5f5e

$ file -k type2.AppImage
type2.AppImage: ELF 64-bit LSB executable, x86-64, version 1 (SYSV) (Leptokaryon)
```

Note how the type 1 AppImage is described as an `ISO 9660 CD-ROM filesystem`, and the type 2 AppImage is not.

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/Patchwork-v3.11.4.tar.gz";
    hash = "sha256-0qTitCeZ6xmWbqYTxp8sDrmVgTNjPZNW0hzUPW++mq4=";
  };
  extraPkgs = pkgs: with pkgs; [ ];
}
```

- `name` specifies the name of the resulting image.
- `src` specifies the ApplImage file to extract.
- `extraPkgs` allows you to pass a function to include additional packages inside the FHS environment your ApplImage is going to run in. There are a few ways to learn which dependencies an application needs:
 - Looking through the extracted ApplImage 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.

pkgs.dockerTools

[buildImage](#)

[buildLayeredImage](#)

[streamLayeredImage](#)

[pullImage](#)

[exportImage](#)

[Environment Helpers](#)

[fakeNss](#)

[buildNixShellImage](#)

`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.

buildImage

v: stable -

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";  
  
    copyToRoot = pkgs.buildEnv {  
        name = "image-root";  
        paths = [ pkgs.redis ];  
        pathsToLink = [ "/bin" ];  
    };  
  
    runAsRoot = ''  
        #!${pkgs.runtimeShell}  
        mkdir -p /data  
    '';  
  
    config = {  
        Cmd = [ "/bin/redis-server" ];  
        WorkingDir = "/data";  
        Volumes = { "/data" = { }; };  
    };  
  
    diskSize = 1024;  
    buildVMMemorySize = 512;  
}
```

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.

v: stable -

- **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.
- **copyToRoot** 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](#).
- **architecture** is *optional* and used to specify the image architecture, this is useful for multi-architecture builds that don't need cross compiling. If not specified it will default to **hostPlatform**.
- **diskSize** is used to specify the disk size of the VM used to build the image in megabytes. By default it's 1024 MiB.
- **buildVMMemorySize** is used to specify the memory size of the VM to build the image in megabytes. By default it's 512 MiB.

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";
  copyToRoot = pkgs.buildEnv {
    name = "image-root";
    paths = [ pkgs.hello ];
    pathsToLink = [ "/bin" ];
  };
};
```

v: stable -

```
config.Cmd = [ "/bin/hello" ];  
}
```

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.

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: `[]`

v: stable -

config optional

architecture is *optional* and used to specify the image architecture, this is useful for multi-architecture builds that don't need cross compiling. If not specified it will default to **hostPlatform**.

Run-time configuration of the container. A full list of the options available is 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.

enableFakechroot optional

Whether to run in **fakeRootCommands** in **fakechroot**, making programs behave as though / is the root of the image being created, while files in the Nix store are available as usual. This allows scripts that perform installation in / to work as expected. Considering that **fa** v: stable -

is implemented via the same mechanism as `fakeroot`, the same caveats apply.

Default: false

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/hello  
/share/info/hello.info -> /nix/store/h1zb1padqbbb7jicsvkmrym3r6snphxg-hello-2.10/share/info/hello.info  
/share/locale/bg/LC_MESSAGES/hello.mo -> /nix/store/h1zb1padqbbb7jicsvkmrym3r6snphxg-hello-2.10/share/locale/bg/LC_MESSAGES/hello.mo
```

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" ];  
}
```

Adjusting maxLayers

v: stable -

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, but 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.

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
```

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.

v: stable -

Its parameters are described in the example below:

```
pullImage {
  imageName = "nixos/nix";
  imageDigest =
    "sha256:473a2b52795866554806aea24d0131bacec46d23af09fef4598eeab331850";
  finalImageName = "nix";
  finalImageTag = "2.11.1";
  sha256 = "sha256-qvhj+Hlmviz+KEBVmsyPIzTB3QlVAFzwAY1zDPIBGxc=";
  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 -- --image-name mysql --image-tag 5
```

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

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
```

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
```

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`.

Environment Helpers

v: stable -

Some packages expect certain files to be available globally. When building an image from scratch (i.e. without `fromImage`), these files are missing. `pkgs.dockerTools` provides some helpers to set up an environment with the necessary files. You can include them in `copyToRoot` like this:

```
buildImage {  
    name = "environment-example";  
    copyToRoot = with pkgs.dockerTools; [  
        usrBinEnv  
        binSh  
        caCertificates  
        fakeNss  
    ];  
}
```

usrBinEnv

This provides the `env` utility at `/usr/bin/env`.

binSh

This provides `bashInteractive` at `/bin/sh`.

caCertificates

This sets up `/etc/ssl/certs/ca-certificates.crt`.

fakeNss

Provides `/etc/passwd` and `/etc/group` that contain root and nobody. Useful when packaging binaries that insist on using nss to look up username/groups (like nginx).

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:

v: stable -

```
buildImage {  
    name = "shadow-basic";  
  
    runAsRoot = ''  
        #!${pkgs.runtimeShell}  
        ${pkgs.dockerTools.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.

fakeNss

If your primary goal is providing a basic skeleton for user lookups to work, and/or a lesser privileged user, adding `pkgs.fakeNss` to the container image root might be the better choice than a custom script running `useradd` and friends.

It provides a `/etc/passwd` and `/etc/group`, containing `root` and `nobody` users and groups.

It also provides a `/etc/nsswitch.conf`, configuring NSS host resolution to first check `/etc/hosts`, before checking DNS, as the default in the absence of a config file (`dns [!UNAVAIL=return] files`) is quite unexpected.

You can pair it with `binSh`, which provides `bin/sh` as a symlink to `bashInteractive` (as `/bin/sh` is configured as a shell).

```
buildImage {  
    name = "shadow-basic";  
  
    copyToRoot = pkgs.buildEnv {  
        name = "image-root";
```

v: stable -

```
    paths = [ binSh pkgs.fakeNss ];
    pathsToLink = [ "/bin" "/etc" "/var" ];
}
}
```

buildNixShellImage

Create a Docker image that sets up an environment similar to that of running `nix-shell` on a derivation. When run in Docker, this environment somewhat resembles the Nix sandbox typically used by `nix-build`, with a major difference being that access to the internet is allowed. It additionally also behaves like an interactive `nix-shell`, running things like `shellHook` and setting an interactive prompt. If the derivation is fully buildable (i.e. `nix-build` can be used on it), running `buildDerivation` inside such a Docker image will build the derivation, with all its outputs being available in the correct `/nix/store` paths, pointed to by the respective environment variables like `$out`, etc.

Warning

The behavior doesn't match `nix-shell` or `nix-build` exactly and this function is known not to work correctly for e.g. fixed-output derivations, content-addressed derivations, impure derivations and other special types of derivations.

Arguments

`drv`

The derivation on which to base the Docker image.

Adding packages to the Docker image is possible by e.g. extending the list of `nativeBuildInputs` of this derivation like

```
buildNixShellImage {
  drv = someDrv.overrideAttrs (old: {
    nativeBuildInputs = old.nativeBuildInputs or [] ++ [
      somethingExtra
    ];
}
```

v: stable -

```
});  
# ...  
}
```

Similarly, you can extend the image initialization script by extending `shellHook`

name *optional*

The name of the resulting image.

Default: `drv.name + "-env"`

tag *optional*

Tag of the generated image.

Default: the resulting image derivation output path's hash

uid/gid *optional*

The user/group ID to run the container as. This is like a `nixbld` build user.

Default: 1000/1000

homeDirectory *optional*

The home directory of the user the container is running as

Default: `/build`

shell *optional*

The path to the `bash` binary to use as the shell. This shell is started when running the image.

Default: `pkgs.bashInteractive + "/bin/bash"`

command *optional*

Run this command in the environment of the derivation, in an interactive shell. See the `--command` option in the [nix-shell documentation](#).

Default: (none)

run *optional*

Same as `command`, but runs the command in a non-interactive shell instead. See the

v: stable -

option in the [nix-shell](#) documentation.

Default: (none)

Example

The following shows how to build the `pkgs.hello` package inside a Docker container built with `buildNixShellImage`.

```
with import <nixpkgs> {};
dockerTools.buildNixShellImage {
  drv = hello;
}
```

Build the derivation:

```
nix-build hello.nix
```

```
these 8 derivations will be built:
```

```
/nix/store/xmw3a5ln29rdalavcxk1w3m4zb2n7kk6-nix-shell-rc.drv
...
Creating layer 56 from paths: [/nix/store/crpnj8ssz0va2q0p5ibv9i6k6n52gc]
Creating layer 57 with customisation...
Adding manifests...
Done.
/nix/store/cpyn1lc897ghx0rhr2xy49jvyn52bazv-hello-2.12-env.tar.gz
```

Load the image:

```
docker load -i result
```

```
0d9f4c4cd109: Loading layer [=====
...
ab1d897c0697: Loading layer [===== v: stable - ::
```

```
Loaded image: hello-2.12-env:pgj9h98nal555415faa43vsydg161bdz
```

Run the container:

```
docker run -it hello-2.12-env:pgj9h98nal555415faa43vsydg161bdz
```

```
[nix-shell:/build]$
```

In the running container, run the build:

```
buildDerivation
```

```
unpacking sources
unpacking source archive /nix/store/8nqv6kshb3vs5q5bs2k600xpj5bkavkc-hello-2.12-env-build
...
patching script interpreter paths in /nix/store/z5wwy5nagzy15gag42vv61c2a9-hello-2.12-env-build/bin/hello
checking for references to /build/ in /nix/store/z5wwy5nagzy15gag42vv61c2a9-hello-2.12-env-build/bin/hello
```

Check the build result:

```
$out/bin/hello
```

```
Hello, world!
```

pkgs.ociTools

[buildContainer](#)

`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

v: stable -

use to run the created container.

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)

v: stable -

- `readonly` makes the container's rootfs read-only if it is set to true. The default value is false `false`.

pkgs.snapTools

[The makeSnap Function](#)

[Build a Hello World Snap](#)

[Build a Graphical Snap](#)

`pkgs.snapTools` is a set of functions for creating Snapcraft images. Snap and Snapcraft is not used to perform these operations.

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.

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";
```

v: stable -

```
};  
}
```

nix-build this expression and install it with snap install ./result --dangerous. hello will now be the Snapcraft version of the package.

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"  
      ];  
    };  
};
```

v: stable -

```
    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](#).

pkgs.portableService

`pkgs.portableService` is a function to create *portable service images*, as read-only, immutable, `squashfs` archives.

systemd supports a concept of [Portable Services](#). Portable Services are a delivery method for system services that uses two specific features of container management:

- Applications are bundled. I.e. multiple services, their binaries and all their dependencies are packaged in an image, and are run directly from it.
- Stricter default security policies, i.e. sandboxing of applications.

This allows using Nix to build images which can be run on many recent Linux distributions.

The primary tool for interacting with Portable Services is `portablectl`, and they are managed by the `systemd-portabled` system service.

Note

Portable services are supported starting with systemd 239 (released on 2018-06-22).

A very simple example of using `portableService` is described below:

```
pkgs.portableService {
  pname = "demo";
  version = "1.0";
  units = [ demo-service demo-socket ];
```

v: stable -

{

The above example will build an squashfs archive image in `result/$pname_${version}.raw`. The image will contain the file system structure as required by the portable service specification, and a subset of the Nix store with all the dependencies of the two derivations in the `units` list. `units` must be a list of derivations, and their names must be prefixed with the service name ("demo" in this case). Otherwise `systemd-portabled` will ignore them.

Note

The `.raw` file extension of the image is required by the portable services specification.

Some other options available are:

- `description`, `homepage`

Are added to the `/etc/os-release` in the image and are shown by the portable services tooling.
Default to empty values, not added to os-release.

- `symlinks`

A list of attribute sets {object, symlink}. Symlinks will be created in the root filesystem of the image to objects in the Nix store. Defaults to an empty list.

- `contents`

A list of additional derivations to be included in the image Nix store, as-is. Defaults to an empty list.

- `squashfsTools`

Defaults to `pkgs.squashfsTools`, allows you to override the package that provides `mksquashfs`.

- `squash-compression`, `squash-block-size`

Options to `mksquashfs`. Default to "xz -Xdict-size 100%" and "1M" respectively.

A typical usage of `symlinks` would be:

```
symlinks = [
```

v: stable -

```
{ object = "${pkgs.cacert}/etc/ssl"; symlink = "/etc/ssl"; }
{ object = "${pkgs.bash}/bin/bash"; symlink = "/bin/sh"; }
{ object = "${pkgs.php}/bin/php"; symlink = "/usr/bin/php"; }
];
```

to create these symlinks for legacy applications that assume them existing globally.

Once the image is created, and deployed on a host in `/var/lib/portables/`, you can attach the image and run the service. As root run:

```
portablectl attach demo_1.0.raw
systemctl enable --now demo.socket
systemctl enable --now demo.service
```

Note

See the [man page of `portablectl`](#) for more info on its usage.

<nixpkgs/nixos/lib/make-disk-image.nix>

[Features](#)

[Usage](#)

<nixpkgs/nixos/lib/make-disk-image.nix> is a function to create *disk images* in multiple formats: raw, QCOW2 (QEMU), QCOW2-Compressed (compressed version), VDI (VirtualBox), VPC (VirtualPC).

This function can create images in two ways:

- using `cptofs` without any virtual machine to create a Nix store disk image,
- using a virtual machine to create a full NixOS installation.

When testing early-boot or lifecycle parts of NixOS such as a bootloader or multiple genera v: stable -

necessary to opt for a full NixOS system installation. Whereas for many web servers, applications, it is possible to work with a Nix store only disk image and is faster to build.

NixOS tests also use this function when preparing the VM. The `cptofs` method is used when `virtualisation.useBootLoader` is false (the default). Otherwise the second method is used.

Features

For reference, read the function signature source code for documentation on arguments:

<https://github.com/NixOS/nixpkgs/blob/master/nixos/lib/make-disk-image.nix>. Features are separated in various sections depending on if you opt for a Nix-store only image or a full NixOS image.

Common

- arbitrary NixOS configuration
- automatic or bound disk size: `diskSize` parameter, `additionalSpace` can be set when `diskSize` is `auto` to add a constant of disk space
- multiple partition table layouts: EFI, legacy, legacy + GPT, hybrid, none through `partitionTableType` parameter
- OVMF or EFI firmwares and variables templates can be customized
- root filesystem `fsType` can be customized to whatever `mkfs.${fsType}` exist during operations
- root filesystem label can be customized, defaults to `nix-store` if it's a Nix store image, otherwise `nixpkgs/nixos`
- arbitrary code can be executed after disk image was produced with `postVM`
- the current nixpkgs can be realized as a channel in the disk image, which will change the hash of the image when the sources are updated
- additional store paths can be provided through `additionalPaths`

Full NixOS image

- arbitrary contents with permissions can be placed in the target filesystem using `content`

v: stable -

- a `/etc/nixpkgs/nixos/configuration.nix` can be provided through `configFile`
- bootloaders are supported
- EFI variables can be mutated during image production and the result is exposed in `$out`
- boot partition size when partition table is `efi` or `hybrid`

On bit-to-bit reproducibility

Images are **NOT** deterministic, please do not hesitate to try to fix this, source of determinisms are (not exhaustive) :

- bootloader installation have timestamps
- SQLite Nix store database contain registration times
- `/etc/shadow` is in a non-deterministic order

A `deterministic` flag is available for best efforts determinism.

Usage

To produce a Nix-store only image:

```
let
  pkgs = import <nixpkgs> {};
  lib = pkgs.lib;
  make-disk-image = import <nixpkgs/nixos/lib/make-disk-image.nix>;
in
  make-disk-image {
    inherit pkgs lib;
    config = {};
    additionalPaths = [ ];
    format = "qcow2";
    onlyNixStore = true;
    partitionTableType = "none";
    installBootLoader = false;
    touchEFIVars = false;
```

v: stable -

```
diskSize = "auto";
additionalSpace = "0M"; # Defaults to 512M.
copyChannel = false;
}
```

Some arguments can be left out, they are shown explicitly for the sake of the example.

Building this derivation will provide a QCOW2 disk image containing only the Nix store and its registration information.

To produce a NixOS installation image disk with UEFI and bootloader installed:

```
let
  pkgs = import <nixpkgs> {};
  lib = pkgs.lib;
  make-disk-image = import <nixpkgs/nixos/lib/make-disk-image.nix>;
  evalConfig = import <nixpkgs/nixos/lib/eval-config.nix>;
in
make-disk-image {
  inherit pkgs lib;
  config = evalConfig {
    modules = [
      {
        fileSystems."/" = { device = "/dev/vda"; fsType = "ext4"; autoFormat = true; boot.grub.device = "/dev/vda"; };
      }
    ];
  };
  format = "qcow2";
  onlyNixStore = false;
  partitionTableType = "legacy+gpt";
  installBootLoader = true;
  touchEFIVars = true;
  diskSize = "auto";
  additionalSpace = "0M"; # Defaults to 512M.
  copyChannel = false;
  memSize = 2048; # Qemu VM memory size in megabytes. Defaults to 1024M.
}
```

}

pkgs.mkBinaryCache

Example

`pkgs.mkBinaryCache` is a function for creating Nix flat-file binary caches. Such a cache exists as a directory on disk, and can be used as a Nix substituter by passing `--substituter file:///path/to/cache` to Nix commands.

Nix packages are most commonly shared between machines using [HTTP, SSH, or S3](#), but a flat-file binary cache can still be useful in some situations. For example, you can copy it directly to another machine, or make it available on a network file system. It can also be a convenient way to make some Nix packages available inside a container via bind-mounting.

Note that this function is meant for advanced use-cases. The more idiomatic way to work with flat-file binary caches is via the [nix-copy-closure](#) command. You may also want to consider [dockerTools](#) for your containerization needs.

Example

The following derivation will construct a flat-file binary cache containing the closure of `hello`.

```
mkBinaryCache {
  rootPaths = [hello];
}
```

- `rootPaths` specifies a list of root derivations. The transitive closure of these derivations' outputs will be copied into the cache.

Here's an example of building and using the cache.

Build the cache on one machine, `host1`:

v: stable -

```
nix-build -E 'with import <nixpkgs> {}; mkBinaryCache { rootPaths = [ hello ] }'
```

```
/nix/store/cc0562q828rnjqjyfj23d5q162gb424g-binary-cache
```

Copy the resulting directory to the other machine, `host2`:

```
scp result host2:/tmp/hello-cache
```

Substitute the derivation using the flat-file binary cache on the other machine, `host2`:

```
nix-build -A hello '<nixpkgs>' \
  --option require-sigs false \
  --option trusted-substituters file:///tmp/hello-cache \
  --option substituters file:///tmp/hello-cache
```

```
/nix/store/gl5a41azbpsadfkmfbilh9yk40dh5dl0-hello-2.12.1
```

Hooks reference

Table of Contents

[Autoconf](#)

[Automake](#)

[autoPatchelfHook](#)

[bmake](#)

[breakpointHook](#)

[cmake](#)

[gdk-pixbuf](#)

v: stable -

[GHC](#)[GNOME platform](#)[installShellFiles](#)[libiconv, libintl](#)[libxml2](#)[Meson](#)[mpiCheckPhaseHook](#)[ninja](#)[patchRcPath hooks](#)[Perl](#)[pkg-config](#)[postgresqlTestHook](#)[Python](#)[scons](#)[teTeX / TeX Live](#)[unzip](#)[validatePkgConfig](#)[wafHook](#)[zig.hook](#)[xcbuildHook](#)

Nixpkgs has several hook packages that augment the stdenv phases.

The stdenv built-in hooks are documented in [the section called “Package setup hooks”](#).

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.

Automake

Adds the `share/aclocal` subdirectory of each build input to the `ACLOCAL_PATH` environment variable.

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.
`autoPatchelfIgnoreMissingDeps` can be set to a list like `autoPatchelfIgnoreMissingDeps = ["libcuda.so.1" "libcudart.so.1"]`; or to `["*"]` to ignore all missing

v: stable -

dependencies.

The `autoPatchelf` command also recognizes a `--no-recurse` command line flag, which prevents it from recursing into subdirectories.

bmake

[bmake](#) is the portable variant of NetBSD make utility.

In Nixpkgs, `bmake` comes with a hook that overrides the default build, check, install and dist phases.

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

Caution with remote builds

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`.

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.

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](#).

GHC

Creates a temporary package database and registers every Haskell build input in it (TODO: how?).

GNOME platform

Hooks related to GNOME platform and related libraries like GLib, GTK and GStreamer are described in the section called "GNOME".

installShellFiles

v: stable -

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)
'';
```

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`.

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.

Meson

Variables controlling Meson

[Meson](#) is an open source meta build system meant to be fast and user-friendly.

In Nixpkgs, meson comes with a setup hook that overrides the configure, check, and install phases.

Being a meta build system, meson needs an accompanying backend. In the context of Nixpkgs, the typical companion backend is [Ninja](#), that provides a setup hook registering ninja-based build and install phases.

Variables controlling Meson

Meson Exclusive Variables

mesonFlags

Controls the flags passed to `meson setup` during configure phase.

mesonWrapMode

Which value is passed as `-Dwrap_mode=` to. In Nixpkgs the default value is `nodownload` v: stable -

subproject will be downloaded (since network access is already disabled during deployment in Nixpkgs).

Note: Meson allows pre-population of subprojects that would otherwise be downloaded.

mesonBuildType

Which value is passed as `--buildtype` to `meson setup` during configure phase. In Nixpkgs the default value is `plain`.

mesonAutoFeatures

Which value is passed as `-Dauto_features=` to `meson setup` during configure phase. In Nixpkgs the default value is `enabled`, meaning that every feature declared as “auto” by the meson scripts will be enabled.

mesonCheckFlags

Controls the flags passed to `meson test` during check phase.

mesonInstallFlags

Controls the flags passed to `meson install` during install phase.

mesonInstallTags

A list of installation tags passed to Meson’s commandline option `--tags` during install phase.

Note: `mesonInstallTags` should be a list of strings, that will be converted to a comma-separated string that is recognized to `--tags`. Example: `mesonInstallTags = ["emulator" "assembler"]`; will be converted to `--tags emulator,assembler`.

dontUseMesonConfigure

v: stable -

When set to true, don't use the predefined `mesonConfigurePhase`.

dontUseMesonCheck

When set to true, don't use the predefined `mesonCheckPhase`.

dontUseMesonInstall

When set to true, don't use the predefined `mesonInstallPhase`.

Honored variables

The following variables commonly used by `stdenv.mkDerivation` are honored by Meson setup hook.

- `prefixKey`
- `enableParallelBuilding`

mpiCheckPhaseHook

This hook can be used to setup a check phase that requires running a MPI application. It detects the used present MPI implementation type and exports the necessary environment variables to use `mpirun` and `mpiexec` in a Nix sandbox.

Example:

```
{ mpiCheckPhaseHook, mpi, ... }:

...
nativeCheckInputs = [
  openssh
  mpiCheckPhaseHook
];
```

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.

Note that if the [Meson setup hook](#) is also active, Ninja's install and check phases will be disabled in favor of Meson's.

patchRcPath hooks

These hooks provide shell-specific utilities (with the same name as the hook) to patch shell scripts meant to be sourced by software users.

The typical usage is to patch initialisation or [rc](#) scripts inside `$out/bin` or `$out/etc`. Such scripts, when being sourced, would insert the binary locations of certain commands into PATH, modify other environment variables or run a series of start-up commands. When shipped from the upstream, they sometimes use commands that might not be available in the environment they are getting sourced in.

The compatible shells for each hook are:

- `patchRcPathBash`: [Bash](#), [ksh](#), [zsh](#) and other shells supporting the Bash-like parameter expansions.
- `patchRcPathCsh`: Csh scripts, such as those targeting [tcsh](#).
- `patchRcPathFish`: [Fish](#) scripts.
- `patchRcPathPosix`: POSIX-conformant shells supporting the limited parameter expansions specified by the POSIX standard. Current implementation uses the parameter expansion `${foo-}` only.

For each supported shell, it modifies the script with a PATH prefix that is later removed when the script ends. It allows nested patching, which guarantees that a patched script may source another patched script.

Syntax to apply the utility to a script:

```
patchRcPath<shell> <file> <PATH-prefix>
```

Example usage:

Given a package `foo` containing an init script `this-foo.fish` that depends on `coreutils`, `man` and `which`, patch the init script for users to source without having the above dependencies in their PATH:

```
{ lib, stdenv, patchRcPathFish }:
stdenv.mkDerivation {

  # ...

  nativeBuildInputs = [
    patchRcPathFish
  ];

  postFixup = ''
    patchRcPathFish $out/bin/this-foo.fish ${lib.makeBinPath [ coreutils man which ]}
  '';
}
```

Note

`patchRcPathCsh` and `patchRcPathPosix` implementation depends on `sed` to do the string processing. The others are in vanilla shell and have no third-party dependencies.

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.

pkg-config

Adds the `lib/pkgconfig` and `share/pkgconfig` subdirectories of each build input to the `PKG_CONFIG_PATH` environment variable.

PKG_CONFIG_PATH environment variable.

postgresqlTestHook

[Variables](#)

[Hooks](#)

[TCP and the Nix sandbox](#)

This hook starts a PostgreSQL server during the `checkPhase`. Example:

```
{ stdenv, postgresql, postgresqlTestHook }:
stdenv.mkDerivation {

  # ...

  nativeCheckInputs = [
    postgresql
    postgresqlTestHook
  ];
}
```

If you use a custom `checkPhase`, remember to add the `runHook` calls:

```
checkPhase ''
  runHook preCheck

  # ... your tests

  runHook postCheck
..
```

Variables

v: stable -

The hook logic will read a number of variables and set them to a default value if unset or empty.

Exported variables:

- **PGDATA**: location of server files.
- **PGHOST**: location of UNIX domain socket directory; the default `host` in a connection string.
- **PGUSER**: user to create / log in with, default: `test_user`.
- **PGDATABASE**: database name, default: `test_db`.

Bash-only variables:

- **postgresqlTestUserOptions**: SQL options to use when creating the `$PGUSER` role, default: "LOGIN". Example: "LOGIN SUPERUSER"
- **postgresqlTestSetupSQL**: SQL commands to run as database administrator after startup, default: statements that create `$PGUSER` and `$PGDATABASE`.
- **postgresqlTestSetupCommands**: bash commands to run after database start, defaults to running `$postgresqlTestSetupSQL` as database administrator.
- **postgresqlEnableTCP**: set to 1 to enable TCP listening. Flaky; not recommended.
- **postgresqlStartCommands**: defaults to `pg_ctl start`.

Hooks

A number of additional hooks are ran in `postgresqlTestHook`

- **postgresqlTestSetupPost**: ran after `postgresql` has been set up.

TCP and the Nix sandbox

`postgresqlEnableTCP` relies on network sandboxing, which is not available on macOS and some custom Nix installations, resulting in flaky tests. For this reason, it is disabled by default.

The preferred solution is to make the test suite use a UNIX domain socket connection. This is the default behavior when no `host` connection parameter is provided. Some test suites hardcode `v: stable -`

for `host` though, so a patch may be required. If you can upstream the patch, you can make `host` default to the `PGHOST` environment variable when set. Otherwise, you can patch it locally to omit the `host` connection string parameter altogether.

Note

The error `libpq: failed (could not receive data from server: Connection refused` is generally an indication that the test suite is trying to connect through TCP.

Python

Adds the `lib/${python.libPrefix}/site-packages` subdirectory of each build input to the `PYTHONPATH` environment variable.

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.

teTeX / TeX Live

Adds the `share/texmf-nix` subdirectory of each build input to the `TEXINPUTS` environment variable.

unzip

This setup hook will allow you to unzip .zip files specified in `$src`. There are many similar packages like `unrar`, `undmg`, etc.

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.

wafHook

v: stable -

Variables controlling wafHook

[Waf](#) is a Python-based software building system.

In Nixpkgs, `wafHook` overrides the default configure, build, and install phases.

Variables controlling wafHook

wafHook Exclusive Variables

The variables below are exclusive of `wafHook`.

wafPath

Location of the `waf` tool. It defaults to `./waf`, to honor software projects that include it directly inside their source trees.

If `wafPath` doesn't exist, then `wafHook` will copy the `waf` provided from Nixpkgs to it.

wafflags

Controls the flags passed to `waf` tool during build and install phases. For settings specific to build or install phases, use `wafBuildFlags` or `wafInstallFlags` respectively.

dontAddWafCrossFlags

When set to `true`, don't add cross compilation flags during configure phase.

dontUseWafConfigure

When set to `true`, don't use the predefined `wafConfigurePhase`.

dontUseWafBuild

v: stable -

When set to true, don't use the predefined `wafBuildPhase`.

don'tUseWafInstall

When set to true, don't use the predefined `wafInstallPhase`.

Similar variables

The following variables are similar to their `stdenv.mkDerivation` counterparts.

wafHook Variable	stdenv.mkDerivation Counterpart
<code>wafConfigureFlags</code>	<code>configureFlags</code>
<code>wafConfigureTargets</code>	<code>configureTargets</code>
<code>wafBuildFlags</code>	<code>buildFlags</code>
<code>wafBuildTargets</code>	<code>buildTargets</code>
<code>wafInstallFlags</code>	<code>installFlags</code>
<code>wafInstallTargets</code>	<code>installTargets</code>

Honored variables

The following variables commonly used by `stdenv.mkDerivation` are honored by `wafHook`.

- `prefixKey`
- `enableParallelBuilding`
- `enableParallelInstalling`

zig.hook

[Example code snippet](#)

v: stable -

Variables controlling zig.hook

[Zig](#) is a general-purpose programming language and toolchain for maintaining robust, optimal and reusable software.

In Nixpkgs, `zig.hook` overrides the default build, check and install phases.

Example code snippet

```
{ lib
, stdenv
, zig_0_11
}:

stdenv.mkDerivation {
  # . . .

  nativeBuildInputs = [
    zig_0_11.hook
  ];

  zigBuildFlags = [ "-Dman-pages=true" ];

  dontUseZigCheck = true;

  # . . .
}
```

Variables controlling zig.hook

zig.hook Exclusive Variables

The variables below are exclusive to `zig.hook`.

v: stable -

don'tUseZigBuild

Disables using `zigBuildPhase`.

don'tUseZigCheck

Disables using `zigCheckPhase`.

don'tUseZigInstall

Disables using `zigInstallPhase`.

Similar variables

The following variables are similar to their `stdenv.mkDerivation` counterparts.

zig.hook Variable	stdenv.mkDerivation Counterpart
<code>zigBuildFlags</code>	<code>buildFlags</code>
<code>zigCheckFlags</code>	<code>checkFlags</code>
<code>zigInstallFlags</code>	<code>installFlags</code>

Variables honored by zig.hook

The following variables commonly used by `stdenv.mkDerivation` are honored by `zig.hook`.

- `prefixKey`
- `dontAddPrefix`

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 `xcbuildHook = false`.

buildPhase and configurePhase to a custom value. xcbuildFlags controls flags passed only to xcbuild.

Languages and frameworks

Table of Contents

[Agda](#)

[Android](#)

[BEAM Languages \(Erlang, Elixir & LFE\)](#)

[Bower](#)

[CHICKEN](#)

[Coq and coq packages](#)

[Crystal](#)

[CUDA](#)

[Cue \(Cuelang\)](#)

[Dart](#)

[Dhall](#)

[Dotnet](#)

[Emscripten](#)

[GNOME](#)

[Go](#)

[Haskell](#)

[Hy](#)

[Idris](#)

v: stable -

[iOS](#)[Java](#)[Javascript](#)[lisp-modules](#)[User's Guide to Lua Infrastructure](#)[Maven](#)[Nim](#)[OCaml](#)[Octave](#)[Perl](#)[PHP](#)[pkg-config](#)[Python](#)[Qt](#)[R](#)[Ruby](#)[Rust](#)[Swift](#)[TeX Live](#)[Titanium](#)[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 accommodated 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.

Agda

[How to use Agda](#)

[Compiling Agda](#)

[Writing Agda packages](#)

[Maintaining the Agda package set on Nixpkgs](#)

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:

v: stable -

```
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";
      hash = "sha256-nEyxYGSWIDNJqBfGpRDLiOAnlHJKEKAOMnIaqfVZzJk=";
    };
  })
])
```

If you want to use a library not added to Nixpkgs, you can add a dependency to a local library by calling `agdaPackages.mkDerivation`.

v: stable -

```
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 = "...";  
      hash = "...";  
    };  
  })  
])
```

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
```

v: stable -

```
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](#).

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;
}
```

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
    ];
  }
}
```

Building Agda packages

The default build phase for `agdaPackages.mkDerivation` 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.

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.

Maintaining the Agda package set on Nixpkgs

We are aiming at providing all common Agda libraries as packages on `nixpkgs`, and keeping them up to date. Contributions and maintenance help is always appreciated, but the maintenance effort is typically low since the Agda ecosystem is quite small.

The `nixpkgs` Agda package set tries to take up a role similar to that of [Stackage](#) in the Haskell world
v: stable -

It is a curated set of libraries that:

1. Always work together.
2. Are as up-to-date as possible.

While the Haskell ecosystem is huge, and Stackage is highly automated, the Agda package set is small and can (still) be maintained by hand.

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  
    '';  
};
```

v: stable -

{

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>.

In the pull request adding this library, you can test whether it builds correctly by writing in a comment:

```
@ofborg build agdaPackages.iowa-stdlib
```

Maintaining Agda packages

As mentioned before, the aim is to have a compatible, and up-to-date package set. These two conditions sometimes exclude each other: For example, if we update `agdaPackages.standard-library` because there was an upstream release, this will typically break many reverse dependencies, i.e. downstream Agda libraries that depend on the standard library. In `nixpkgs` we are typically among the first to notice this, since we have build tests in place to check this.

In a pull request updating e.g. the standard library, you should write the following comment:

```
@ofborg build agdaPackages.standard-library.passthru.tests
```

This will build all reverse dependencies of the standard library, for example `agdaPackages.agda-categories`, or `agdaPackages.generic`.

In some cases it is useful to build *all* Agda packages. This can be done with the following Github comment:

v: stable -

```
@ofborg build agda.passthru.tests.allPackages
```

Sometimes, the builds of the reverse dependencies fail because they have not yet been updated and released. You should drop the maintainers a quick issue notifying them of the breakage, citing the build error (which you can get from the ofborg logs). If you are motivated, you might even send a pull request that fixes it. Usually, the maintainers will answer within a week or two with a new release. Bumping the version of that reverse dependency should be a further commit on your PR.

In the rare case that a new release is not to be expected within an acceptable time, mark the broken package as broken by setting `meta.broken = true;`. This will exclude it from the build test. It can be added later when it is fixed, and does not hinder the advancement of the whole package set in the meantime.

Android

[Deploying an Android SDK installation with plugins](#)

[Using predefined Android package compositions](#)

[Building an Android application](#)

[Spawning emulator instances](#)

[Notes on environment variables in Android projects](#)

[Notes on improving build.gradle compatibility](#)

[Querying the available versions of each plugin](#)

[Updating the generated expressions](#)

The Android build environment provides three major features and a number of supporting features.

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.

v: stable -

```
with import <nixpkgs> {};

let
  androidComposition = androidenv.composeAndroidPackages {
    cmdLineToolsVersion = "8.0";
    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:

- `cmdLineToolsVersion`, specifies the version of the `cmdline-tools` package to use
- `toolsVersion`, specifies the version of the `tools` package. Notice `tools` is obsolete, and currently only `26.1.1` is available, so there's not a lot of options here, however, you can `-v: stable -`

`null` if you don't want it.

- `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:

v: stable -

- `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
```

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
```

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  
    # build requires  
    platformVersions = [ "24" ];  
  
    # When we include the NDK, then ndk-build is invoked before Ant gets invoked  
    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 cont

v: stable -

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.

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";  
  app = ./MyApp.apk;  
  package = "MyApp";  
  activity = "MainActivity";  
}
```

v: stable -

In addition to prebuilt APKs, you can also bind the APK parameter to a `buildApp {}` function invocation shown in the previous example.

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=${ANDRO
}
```

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";
```

v: stable -

```
# 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/${cmakeVersion}.*/bin"):$
  '';
}
}
```

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.

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"
```

v: stable -

```
    }  
}  
}
```

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.

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
```

BEAM Languages (Erlang, Elixir & LFE)

Introduction

Available versions and deprecations schedule

Structure

Build Tools

How to Install BEAM Packages

v: stable -

Packaging BEAM Applications

How to Develop

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.

Available versions and deprecations schedule

Elixir

nixpkgs follows the [official elixir deprecation schedule](#) and keeps the last 5 released versions of Elixir available.

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.erlang_22`, 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

v: stable -

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.erlang_22_odbc_javac`, which corresponds to `beam.interpreters.erlang_22` and `beam.interpreters.erlang_22_nox`, which corresponds to `beam.interpreters.erlang_22`.

Build Tools

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.

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.

How to Install BEAM Packages

BEAM builders are not registered at the top level, because they are not relevant to the vast majority of users. Instead, they are registered under the `beam` group, which is part of the `otp` group.

Nix users. To use any of those builders into your environment, refer to them by their attribute path under `beamPackages`, e.g. `beamPackages.rebar3`:

Example 234. Ephemeral shell

```
$ nix-shell -p beamPackages.rebar3
```

Example 235. Declarative shell

```
let
  pkgs = import <nixpkgs> { config = {}; overlays = []; };
in
pkgs.mkShell {
  packages = [ pkgs.beamPackages.rebar3 ];
}
```

Packaging BEAM Applications

Erlang Applications

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.

Erlang.mk Packages

Erlang.mk functions similarly to Rebar3, except we use `buildErlangMk` instead of `buildRebar3`.

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.

mixRelease - Elixir Phoenix example

there are 3 steps, frontend dependencies (javascript), backend dependencies (elixir) and the final derivation that puts both of those together

mixRelease - Frontend dependencies (javascript)

For phoenix projects, inside of nixpkgs you can either use `yarn2nix` (`mkYarnModule`) or `node2nix`. An example with `yarn2nix` can be found [here](#). An example with `node2nix` will follow. To package something outside of nixpkgs, you have alternatives like [npmlock2nix](#) or [nix-npm-buildpackage](#)

mixRelease - backend dependencies (mix)

There are 2 ways to package backend dependencies. With `mix2nix` and with a fixed-output-derivation (FOD).

mix2nix

`mix2nix` is a cli tool available in nixpkgs. it will generate a nix expression from a `mix.lock` file. It is quite standard in the 2nix tool series.

Note that currently `mix2nix` can't handle git dependencies inside the `mix.lock` file. If you have git dependencies, you can either add them manually (see [example](#)) or use the FOD method.

The advantage of using `mix2nix` is that nix will know your whole dependency graph. On a dependency update, this won't trigger a full rebuild and download of all the dependencies, where FOD will do so.

Practical steps:

- run `mix2nix > mix_deps.nix` in the upstream repo.
- pass `mixNixDeps = with pkgs; import ./mix_deps.nix { inherit lib beamPackages; };` as an argument to `mixRelease`.

If there are git dependencies.

v: stable -

- You'll need to fix the version artificially in mix.exs and regenerate the mix.lock with fixed version (on upstream). This will enable you to run `mix2nix > mix_deps.nix`.
- From the mix_deps.nix file, remove the dependencies that had git versions and pass them as an override to the import function.

```
mixNixDeps = import ./mix.nix {  
    inherit beamPackages lib;  
    overrides = (final: prev: {  
        # mix2nix does not support git dependencies yet,  
        # so we need to add them manually  
        prometheus_ex = beamPackages.buildMix rec {  
            name = "prometheus_ex";  
            version = "3.0.5";  
  
            # Change the argument src with the git src that you actually need  
            src = fetchFromGitLab {  
                domain = "git.pleroma.social";  
                group = "pleroma";  
                owner = "elixir-libraries";  
                repo = "prometheus.ex";  
                rev = "a4e9beb3c1c479d14b352fd9d6dd7b1f6d7deee5";  
                hash = "sha256-U17LlN6aGUKUFnT4XyYXppRN+TvUBIBRHEUsfeIiG0w=";  
            };  
            # you can re-use the same beamDeps argument as generated  
            beamDeps = with final; [ prometheus ];  
        };  
    });  
};
```

You will need to run the build process once to fix the hash to correspond to your new git src.

FOD

A fixed output derivation will download mix dependencies from the internet. To ensure reproducibility, a hash will be supplied. Note that mix is relatively reproducible. An FOD generating a different hash on each run hasn't been observed (as opposed to npm where the chances are relatively high). v: stable -

for a usage example of FOD.

Practical steps

- start with the following argument to mixRelease

```
mixFodDeps = fetchMixDeps {  
  pname = "mix-deps-${pname}";  
  inherit src version;  
  hash = lib.fakeHash;  
};
```

The first build will complain about the hash value, you can replace with the suggested value after that.

Note that if after you've replaced the value, nix suggests another hash, then mix is not fetching the dependencies reproducibly. An FOD will not work in that case and you will have to use mix2nix.

mixRelease - example

Here is how your `default.nix` file would look for a phoenix project.

```
with import <nixpkgs> { };  
  
let  
  # beam.interpreters.erlang_26 is available if you need a particular version  
  packages = beam.packagesWith beam.interpreters.erlang;  
  
  pname = "your_project";  
  version = "0.0.1";  
  
  src = builtins.fetchgit {  
    url = "ssh://git@github.com/your_id/your_repo";  
    rev = "replace_with_your_commit";  
  };  
  
  # if using mix2nix you can use the mixNixDeps attribute  
  mixFodDeps = packages.fetchMixDeps {  
    pname = "mix-deps-${pname}";
```

v: stable -

```
inherit src version;
# nix will complain and tell you the right value to replace this with
hash = lib.fakeHash;
mixEnv = ""; # default is "prod", when empty includes all dependencies
# if you have build time environment variables add them here
MY_ENV_VAR="my_value";
};

nodeDependencies = (pkgs.callPackage ./assets/default.nix { }).shell.nodeDependencies;

in packages.mixRelease {
  inherit src pname version mixFodDeps;
  # if you have build time environment variables add them here
  MY_ENV_VAR="my_value";

  postBuild = ''
    ln -sf ${nodeDependencies}/lib/node_modules assets/node_modules
    npm run deploy --prefix ./assets

    # for external task you need a workaround for the no deps check flag
    # https://github.com/phoenixframework/phoenix/issues/2690
    mix do deps.loadpaths --no-deps-check, phx.digest
    mix phx.digest --no-deps-check
  '';
}
}
```

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 .`

v: stable -

- 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.

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" ];
    # note that if you are connecting to a postgres instance on a different
    # postgresql.service should not be included in the requires.
    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;
    };
  };
}
```

v: stable -

```
# 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 ];
};

# in case you have migration scripts or you want to use a remote shell
environment.systemPackages = [ release ];
}
```

How to Develop

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
```

v: stable -

```
elixir = beam.packages.erlang_24.elixir_1_12;  
in  
mkShell {  
  buildInputs = [ elixir ];  
}
```

Using an overlay

If you need to use an overlay to change some attributes of a derivation, e.g. if you need a bugfix from a version that is not yet available in nixpkgs, you can override attributes such as `version` (and the corresponding `hash`) and then use this overlay in your development environment:

shell.nix

```
let  
  elixir_1_13_1_overlay = (self: super: {  
    elixir_1_13 = super.elixir_1_13.override {  
      version = "1.13.1";  
      sha256 = "sha256-t0ic1LcC7EV3avWGdR7VbyX7pGDpnJSW1ZvwvQUPC3w=";  
    };  
  });  
  pkgs = import <nixpkgs> { overlays = [ elixir_1_13_1_overlay ]; };  
in  
with pkgs;  
mkShell {  
  buildInputs = [  
    elixir_1_13  
  ];  
}
```

Elixir - Phoenix project

Here is an example `shell.nix`.

```
with import <nixpkgs> { };
```

v: stable -

```
let
  # define packages to install
  basePackages = [
    git
    # replace with beam.packages.erlang.elixir_1_13 if you need
    beam.packages.erlang.elixir
    nodejs
    postgresql_14
    # 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
    # make hex from Nixpkgs available
    # `mix local.hex` will install hex into MIX_HOME and should take precedence
    export MIX_PATH="${beam.packages.erlang.hex}/lib/erlang/lib/hex/ebin"
    export PATH=$MIX_HOME/bin:$HEX_HOME/bin:$PATH
    export LANG=C.UTF-8
    # keep your shell history in iex
    export ERL_AFLAGS="-kernel shell_history enabled"

    # postges related
    # keep all your db data in a folder inside the project
    export PGDATA="$PWD/db"
```

v: stable -

```
# 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`

Bower

[bower2nix usage](#)

[buildBowerComponents function](#)

[Troubleshooting](#)

[Bower](#) is a package manager for web site front-end components. Bower packages (compr v: stable -

artifacts 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`.

bower2nix usage

Suppose you have a `bower.json` with the following contents:

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" "1749xb0firxdra4rzadm4q9x90v6pzkl"
  (fetchbower "bootstrap" "3.3.6" "~3.3.6" "1vvqlpbfcy0k5pncfjaiskj3y6scw"
  (fetchbower "jquery" "2.2.2" "1.9.1 - 2" "10sp5h98sqwk90y4k6hbdviwqzvzw"
];
```

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` v: stable -

useful.

buildBowerComponents function

The function is implemented in [pkgs/development/bower-modules/generic/default.nix](#).

Example buildBowerComponents

```
bowerComponents = buildBowerComponents {  
    name = "my-web-app";  
    generated = ./bower-packages.nix; # note 1  
    src = myWebApp; # note 2  
};
```

In “[buildBowerComponents](#)” example the following arguments are of special significance to the function:

1. `generated` specifies the file which was created by **bower2nix**.
2. `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.

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");
```

v: stable -

```
gulp
  .src(["./bower_components/**/*"])
  .pipe(gulp.dest("./gulpdist"));
});
```

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 { # note 1
    name = "my-web-app";
    generated = ./bower-packages.nix;
    src = myWebApp;
  };  
  
  buildPhase = ''
    cp --reflink=auto --no-preserve=mode -R $bowerComponents/bower_components
    export HOME=$PWD # note 3
    ${pkgs.nodePackages.gulp}/bin/gulp build # note 4
  '';  
  
  installPhase = "mv gulpdist $out";
}
```

A few notes about [Full example – default.nix](#):

1. The result of `buildBowerComponents` is an input to the frontend build.

2. Whether to symlink or copy the `bower_components` directory depends on the build

v: stable -

In this case a copy is used to avoid **gulp** silliness with permissions.

3. **gulp** requires **HOME** to refer to a writeable directory.

4. The actual build command in this example is **gulp**. Other tools could be used instead.

Troubleshooting

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**.

CHICKEN

[Using Eggs](#)

[Updating Eggs](#)

[Adding Eggs](#)

[Override Scope](#)

[CHICKEN](#) is a [R⁵RS](#)-compliant Scheme compiler. It includes an interactive mode and a custom package format, "eggs".

Using Eggs

Eggs described in nixpkgs are available inside the **chickenPackages.chickenEggs** attrset.

Including an egg as a build input is done in the typical Nix fashion. For example, to include support for [SRFI 189](#) in a derivation, one might write:

v: stable -

```
buildInputs = [
  chicken
  chickenPackages.chickenEggs.srfi-189
];
```

Both `chicken` and its eggs have a setup hook which configures the environment variables `CHICKEN_INCLUDE_PATH` and `CHICKEN_REPOSITORY_PATH`.

Updating Eggs

nixpkgs only knows about a subset of all published eggs. It uses [egg2nix](#) to generate a package set from a list of eggs to include.

The package set is regenerated by running the following shell commands:

```
$ nix-shell -p chickenPackages.egg2nix
$ cd pkgs/development/compilers/chicken/5/
$ egg2nix eggs.scm > eggs.nix
```

Adding Eggs

When we run `egg2nix`, we obtain one collection of eggs with mutually-compatible versions. This means that when we add new eggs, we may need to update existing eggs. To keep those separate, follow the procedure for updating eggs before including more eggs.

To include more eggs, edit `pkgs/development/compilers/chicken/5/eggs.scm`. The first section of this file lists eggs which are required by `egg2nix` itself; all other eggs go into the second section. After editing, follow the procedure for updating eggs.

Override Scope

The chicken package and its eggs, respectively, reside in a scope. This means, the scope can be overridden to effect other packages in it.

This example shows how to use a local copy of `srfi-180` and have it affect all the other eggs in the scope.

```
let
  myChickenPackages = pkgs.chickenPackages.overrideScope' (self: super: {
    # The chicken package itself can be overridden to effect the whole e
    # chicken = super.chicken.overrideAttrs {
    #   src = ...
    # };
  });

  chickenEggs = super.chickenEggs.overrideScope' (eggself: eggsuper: {
    srfi-180 = eggsuper.srfi-180.overrideAttrs {
      # path to a local copy of srfi-180
      src = ...
    };
  });
};

in
# Here, `myChickenPackages.chickenEggs.json-rpc`, which depends on `srfi-180` depends on `srfi-180`.
# the local copy of `srfi-180`.
# ...
```

Coq and coq packages

[Coq derivation: coq](#)

[Coq packages attribute sets: coqPackages](#)

[Three ways of overriding Coq packages](#)

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 `v: stable -> v: experimental` section.

coqPackages section below,

- **customOCamlPackages** (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"; }.`

The associated package set can be obtained using `mkCoqPackages coq`, where `coq` is the derivation to use.

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: p` v: stable -

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 **hash** 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,

v: stable -

- **namePrefix** (optional, defaults to ["coq"]), provides a way to alter the computation of `name` from `pname`, by explaining which dependencies must occur in `name`,
- **nativeBuildInputs** (optional), is a list of executables that are required to build the current derivation, in addition to the default ones (namely `which`, `dune` and `ocaml` depending on whether `useDune`, `useDuneifVersion` and `mlPlugin` are set).
- **extraNativeBuildInputs** (optional, deprecated), an additional list of derivation to add to `nativeBuildInputs`,
- **overrideNativeBuildInputs** (optional) replaces the default list of derivation to which `nativeBuildInputs` and `extraNativeBuildInputs` adds extra elements,
- **buildInputs** (optional), is a list of libraries and dependencies that are required to build and run the current derivation, in addition to the default one [coq],
- **extraBuildInputs** (optional, deprecated), an additional list of derivation to add to `buildInputs`,
- **overrideBuildInputs** (optional) replaces the default list of derivation to which `buildInputs` and `extraBuildInputs` adds extras elements,
- **propagatedBuildInputs** (optional) is passed as is to `mkDerivation`, we recommend to use this for Coq libraries and Coq plugin dependencies, as this makes sure the paths of the compiled libraries and plugins will always be added to the build environments of subsequent derivation, which is necessary for Coq packages to work correctly,
- **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 `nativeBuildInputs`, `buildInputs`, and `propagatedBuildInputs` to depend on the same package set Coq was built against.
- **useDuneifVersion** (optional, default to (x: false)) uses Dune to build the package if the provided predicate evaluates to true on the version, e.g. `useDuneifVersion = versions.isGe "1.1"` will use dune if the version of the package is greater or equal to "1.1",
- **useDune** (optional, defaults to `false`) uses Dune to build the package if set to true, the presence of this attribute overrides the behavior of the previous one.

v: stable -

- **opam-name** (optional, defaults to concatenating with a dash separator the components of **namePrefix** and **pname**), name of the Dune package to build.
- **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-multi */ v: stable - .
```

```
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 = "1.11.0" }
  { cases = [ (range "8.7" "8.11") (range "1.8" "1.10") ]; out = "1.10" }
  { cases = [ (range "8.7" "8.10") (range "1.8" "1.10") ]; out = "1.10" }
  { cases = [ "8.6" (range "1.6" "1.7") ]; out = "1.6" }
] null;
release = {
  "1.5.2".sha256 = "15aspf3jfykp1xgsxf8knqkxv8aav2p39c2fyirw7pwsfbsv2c4";
  "1.5.1".sha256 = "13nlfm2wqrripaq671gakz5mn4r0xwm0646araxv0nh455p9ndjs";
  "1.5.0".sha256 = "064rv0x5g7y1a0nip6ic91vzmq52alf6in2bc2dmss6dmzv90hw";
  "1.5.0".rev = "1.5";
  "1.4".sha256 = "0vnkirs8iqsv8s59yx1fvg1nkwnzydl42z3scya1xp1b48qkgn0j";
  "1.3".sha256 = "0l3vi5n094nx3qmy66hsv867fnqm196r8v605kpk24gl0aa57wh";
  "1.2".sha256 = "1mh1w339dslgv4f810xr1b8v2w7rpx6fgk9pz96q0fyq49fw2xc";
  "1.1".sha256 = "1q8alsm89wkc0lhcvxlyn0pd8rbl2nnxg81zyrabpz610qqjqc3";
  "1.0".sha256 = "1qmbxp1h81cy3imh627pznmng0kvv37k4hrwi2faa101s6bcx55r
};

propagatedBuildInputs =
[ mathcomp.ssreflect mathcomp.algebra mathcomp-finmap mathcomp-bigeno
];

meta = {
  description = "A Coq/SSReflect Library for Monoidal Rings and Multinomial
  license = licenses.cecill-c;
};
}
```

Three ways of overriding Coq packages

There are three distinct ways of changing a Coq package by overriding one of its values: `.override`, `overrideCoqDerivation`, and `.overrideAttrs`. This section explains what sort of values can be overridden with each of these methods.

v: stable -

.override

.override lets you change arguments to a Coq derivation. In the case of the multinomials package above, .override would let you override arguments like mkCoqDerivation, version, coq, mathcomp, mathcom-finmap, etc.

For example, assuming you have a special mathcomp dependency you want to use, here is how you could override the mathcomp dependency:

```
multinomials.override {
    mathcomp = my-special-mathcomp;
}
```

In Nixpkgs, all Coq derivations take a version argument. This can be overridden in order to easily use a different version:

```
coqPackages.multinomials.override {
    version = "1.5.1";
}
```

Refer to [the section called “Coq packages attribute sets: coqPackages”](#) for all the different formats that you can potentially pass to version, as well as the restrictions.

overrideCoqDerivation

The overrideCoqDerivation function lets you easily change arguments to mkCoqDerivation. These arguments are described in [the section called “Coq packages attribute sets: coqPackages”](#).

For example, here is how you could locally add a new release of the multinomials library, and set the defaultVersion to use this release:

```
coqPackages.lib.overrideCoqDerivation
{
    defaultVersion = "2.0";
    release."2.0".sha256 = "1lq8x86vd3vqqh2yq6hvyaagpnhfq5wmk5pg2" v: stable -
```

```
}
```

```
coqPackages.multinomials
```

.overrideAttrs

.overrideAttrs lets you override arguments to the underlying stdenv.mkDerivation call. Internally, mkCoqDerivation uses stdenv.mkDerivation to create derivations for Coq libraries. You can override arguments to stdenv.mkDerivation with .overrideAttrs.

For instance, here is how you could add some code to be performed in the derivation after installation is complete:

```
coqPackages.multinomials.overrideAttrs (oldAttrs: {
    postInstall = oldAttrs.postInstall or "" + ''
        echo "you can do anything you want here"
    '';
})
```

Crystal

[Building a Crystal package](#)

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. Executable projects should usually commit the `shard.lock` file, but sometimes that's not the case, which means you need to generate it yourself. With an existing `shard.lock` file, `crystal2nix` can be run.

```
$ git clone https://github.com/mint-lang/mint
```

v: stable -

```
$ cd mint
$ git checkout 0.5.0
$ if [ ! -f shard.lock ]; then nix-shell -p shards --run "shards lock"; fi
$ 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;
    hash = "sha256-dFN9l5fgrM/TtOPqlQvUYgixE4KPr629aBmkwdDoq28=";
  };

  # 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;  
        hash = "sha256-dFN9l5fgrM/TtOPqlQvUYgixE4KPr629aBmkwdDoq28=";  
    };  
  
    shardsFile = ./shards.nix;  
    crystalBinaries.mint.src = "src/mint.cr";  
  
    buildInputs = [ openssl ];  
}
```

CUDA

Adding a new CUDA release

CUDA-only packages are stored in the `cudaPackages` packages set. This set includes the `cudatoolkit`, portions of the toolkit in separate derivations, `cudnn`, `cutensor` and `nccl`.

A package set is available for each CUDA version, so for example `cudaPackages_11_6`. Within each set is a matching version of the above listed packages. Additionally, other versions of the packages that are packaged and compatible are available as well. For example, there can be a `cudaPackages.cudnn_8_3` package.

To use one or more CUDA packages in an expression, give the expression a `cudaPackages` parameter, and in case CUDA is optional

```
{ config
, cudaSupport ? config.cudaSupport
, cudaPackages ? {}
, ...
}:
```

When using `callPackage`, you can choose to pass in a different variant, e.g. when a different version of the toolkit suffices

```
mypkg = callPackage { cudaPackages = cudaPackages_11_5; }
```

If another version of say `cudnn` or `cutensor` is needed, you can override the package set to make it the default. This guarantees you get a consistent package set.

```
mypkg = let
  cudaPackages = cudaPackages_11_5.overrideScope (final: prev: {
    cudnn = prev.cudnn_8_3;
  });
in callPackage { inherit cudaPackages; };
```

The CUDA NVCC compiler requires flags to determine which hardware you want to target for in terms of SASS (real hardware) or PTX (JIT kernels).

Nixpkgs tries to target support real architecture defaults based on the CUDA toolkit version v: stable -

support for future hardware. Experienced users may optimize this configuration for a variety of reasons such as reducing binary size and compile time, supporting legacy hardware, or optimizing for specific hardware.

You may provide capabilities to add support or reduce binary size through `config` using `cudaCapabilities = ["6.0" "7.0"]`; and `cudaForwardCompat = true`; if you want PTX support for future hardware.

Please consult [GPUs supported](#) for your specific card(s).

Library maintainers should consult [NVCC Docs](#) and release notes for their software package.

Adding a new CUDA release

WARNING

This section of the docs is still very much in progress. Feedback is welcome in GitHub Issues tagging @NixOS/cuda-maintainers or on [Matrix](#).

The CUDA Toolkit is a suite of CUDA libraries and software meant to provide a development environment for CUDA-accelerated applications. Until the release of CUDA 11.4, NVIDIA had only made the CUDA Toolkit available as a multi-gigabyte runfile installer, which we provide through the [`cudaPackages.cudaToolkit`](#) attribute. From CUDA 11.4 and onwards, NVIDIA has also provided CUDA redistributables (“CUDA-redist”): individually packaged CUDA Toolkit components meant to facilitate redistribution and inclusion in downstream projects. These packages are available in the [`cudaPackages`](#) package set.

All new projects should use the CUDA redistributables available in [`cudaPackages`](#) in place of [`cudaPackages.cudaToolkit`](#), as they are much easier to maintain and update.

Updating CUDA redistributables

1. Go to NVIDIA’s index of CUDA redistributables: <https://developer.download.nvidia.com/compute/cuda/redist/>
2. Copy the `redistrib_*.json` corresponding to the release to `pkgs/development/compilers/cudatoolkit/redist/manifests`.

v: stable -

3. Generate the `redistrib_features_*.json` file by running:

```
nix run github:ConnorBaker/cuda-redist-find-features -- <path to mani
```

That command will generate the `redistrib_features_*.json` file in the same directory as the manifest.

4. Include the path to the new manifest in `pkgs/development/compilers/cudatoolkit/redist/extension.nix`.

Updating the CUDA Toolkit runfile installer

WARNING

While the CUDA Toolkit runfile installer is still available in Nixpkgs as the `cudaPackages.cudatoolkit` attribute, its use is not recommended and should it be considered deprecated. Please migrate to the CUDA redistributables provided by the `cudaPackages` package set.

To ensure packages relying on the CUDA Toolkit runfile installer continue to build, it will continue to be updated until a migration path is available.

1. Go to NVIDIA's CUDA Toolkit runfile installer download page: <https://developer.nvidia.com/cuda-downloads>
2. Select the appropriate OS, architecture, distribution, and version, and installer type.
 - For example: Linux, x86_64, Ubuntu, 22.04, runfile (local)
 - NOTE: Typically, we use the Ubuntu runfile. It is unclear if the runfile for other distributions will work.
3. Take the link provided by the installer instructions on the webpage after selecting the installer type and get its hash by running:

```
nix store prefetch-file --hash-type sha256 <link>
```

v: stable -

4. Update `pkgs/development/compilers/cudatoolkit/versions.toml` to include the release.

Updating the CUDA package set

1. Include a new `cudaPackages_<major>_<minor>` package set in `pkgs/top-level/all-packages.nix`.
 - NOTE: Changing the default CUDA package set should occur in a separate PR, allowing time for additional testing.
2. Successfully build the closure of the new package set, updating `pkgs/development/compilers/cudatoolkit/redist/overrides.nix` as needed. Below are some common failures:

Unable to ...	During ...	Reason	Solution	Note
Find headers	<code>configurePhase</code> or <code>buildPhase</code>	Missing dependency on a <code>dev</code> output	Add the missing dependency	The <code>dev</code> output typically contain the headers
Find libraries	<code>configurePhase</code>	Missing dependency on a <code>dev</code> output	Add the missing dependency	The <code>dev</code> output typically contain CMake configuration files
Find libraries	<code>buildPhase</code> or <code>patchelf</code>	Missing dependency on a <code>lib</code> or <code>static</code> output	Add the missing dependency	The <code>lib</code> or <code>static</code> output typically contain the libraries

In the scenario you are unable to run the resulting binary: this is arguably the most complicated as it could be any combination of the previous reasons. This type of failure typically occurs when a library attempts to load or open a library it depends on that it does not declare in its `DT_NEEDED` section. As a first step, ensure that dependencies are patched with `cudaPackages.autoAddOpenGLF` v: stable -

Failing that, try running the application with [nixGL](#) or a similar wrapper tool. If that works, it likely means that the application is attempting to load a library that is not in the **RPATH** or **RUNPATH** of the binary.

Cue (Cuelang)

[Cuelang schema quick start](#)

[writeCueValidator](#)

[Cuelang](#) is a language to:

- describe schemas and validate backward-compatibility
- generate code and schemas in various formats (e.g. JSON Schema, OpenAPI)
- do configuration akin to [Dhall Lang](#)
- perform data validation

Cuelang schema quick start

Cuelang schemas are similar to JSON, here is a quick cheatsheet:

- Default types includes: `null`, `string`, `bool`, `bytes`, `number`, `int`, `float`, lists as `[...T]` where T is a type.
- All structures, defined by: `myStructName: { <fields> }` are **open** – they accept fields which are not specified.
- Closed structures can be built by doing `myStructName: close({ <fields> })` – they are strict in what they accept.
- `#X` are **definitions**, referenced definitions are **recursively closed**, i.e. all its children structures are **closed**.
- `&` operator is the [unification operator](#) (similar to a type-level merging operator), `|` is the [v: stable -](#)

[operator](#) (similar to a type-level union operator).

- Values **are** types, i.e. `myStruct: { a: 3 }` is a valid type definition that only allows 3 as value.
- Read <https://cuelang.org/docs/concepts/logic/> to learn more about the semantics.
- Read <https://cuelang.org/docs/references/spec/> to learn about the language specification.

writeCueValidator

Nixpkgs provides a `pkgs.writeCueValidator` helper, which will write a validation script based on the provided Cuelang schema.

Here is an example:

```
pkgs.writeCueValidator
  (pkgs.writeText "schema.cue" ''
    #Def1: {
      field1: string
    }
  )
{ document = "#Def1"; }
```

- The first parameter is the Cue schema file.
- The second parameter is an options parameter, currently, only: `document` can be passed.

`document` : match your input data against this fragment of structure or definition, e.g. you may use the same schema file but different documents based on the data you are validating.

Another example, given the following `validator.nix`:

```
{ pkgs ? import <nixpkgs> {} }:
let
  genericValidator = version:
  pkgs.writeCueValidator
    (pkgs.writeText "schema.cue" ''
```

v: stable -

```
#Version1: {
    field1: string
}
#Version2: #Version1 & {
    field1: "unused"
} ''
)
{ document = "#Version${toString version}"; };

in
{
  validateV1 = genericValidator 1;
  validateV2 = genericValidator 2;
}
```

The result is a script that will validate the file you pass as the first argument against the schema you provided `writeCueValidator`.

It can be any format that `cue vet` supports, i.e. YAML or JSON for example.

Here is an example, named `example.json`, given the following JSON:

```
{ "field1": "abc" }
```

You can run the result script (named `validate`) as the following:

```
$ nix-build validator.nix
$ ./result example.json
$ ./result-2 example.json
field1: conflicting values "unused" and "abc":
  ./example.json:1:13
  ../../../../../../nix/store/v64dzx3vr3glpk0cq4hzmh450lrwh6sg-schema.ci
$ sed -i 's/"abc"/3/' example.json
$ ./result example.json
field1: conflicting values 3 and string (mismatched types int and string)
  ./example.json:1:13
  ../../../../../../nix/store/v64dzx3vr3glpk0cq4hzmh450lrwh6sg - - - - - v: stable -
```

Known limitations

- The script will enforce **concrete** values and will not accept lossy transformations (strictness). You can add these options if you need them.

Dart

[Dart applications](#)

[Flutter applications](#)

Dart applications

The function `buildDartApplication` builds Dart applications managed with pub.

It fetches its Dart dependencies automatically through `fetchDartDeps`, and (through a series of hooks) builds and installs the executables specified in the pubspec file. The hooks can be used in other derivations, if needed. The phases can also be overridden to do something different from installing binaries.

If you are packaging a Flutter desktop application, use [`buildFlutterApplication`](#) instead.

`vendorHash`: is the hash of the output of the dependency fetcher derivation. To obtain it, set it to `lib.fakeHash` (or omit it) and run the build ([more details here](#)).

If the upstream source is missing a `pubspec.lock` file, you'll have to vendor one and specify it using `pubspecLockFile`. If it is needed, one will be generated for you and printed when attempting to build the derivation.

The `depsListFile` must always be provided when packaging in Nixpkgs. It will be generated and printed if the derivation is attempted to be built without one. Alternatively, `autoDepsList` may be set to `true` only when outside of Nixpkgs, as it relies on import-from-derivation.

The `dart` commands run can be overridden through `pubGetScript` and `dartCompileCommand`, you can also add flags using `dartCompileFlags` or `dartJitFlags`.

Dart supports multiple [`outputs types`](#), you can choose between them using `dartOutputT` v: stable -

(defaults to `exe`). If you want to override the binaries path or the source path they come from, you can use `dartEntryPoints`. Outputs that require a runtime will automatically be wrapped with the relevant runtime (`dartaotruntime` for `aot-snapshot`, `dart run` for `jit-snapshot` and `kernel`, `node` for `js`), this can be overridden through `dartRuntimeCommand`.

```
{ buildDartApplication, fetchFromGitHub }:

buildDartApplication rec {
  pname = "dart-sass";
  version = "1.62.1";

  src = fetchFromGitHub {
    owner = "sass";
    repo = pname;
    rev = version;
    hash = "sha256-U6enz8yJcc4Wf8m54eYIAvVg/jsGi247Wy8lp1r1wg4=";
  };

  pubspecLockFile = ./pubspec.lock;
  depsListFile = ./deps.json;
  vendorHash = "sha256-Atm7zfnDambN/BmmUf4BG0yUz/y6xWzf0reDw3Ad41s=";
}
```

Flutter applications

The function `buildFlutterApplication` builds Flutter applications.

See the [Dart documentation](#) for more details on required files and arguments.

```
{ flutter, fetchFromGitHub }:

flutter.buildFlutterApplication {
  pname = "firmware-updater";
  version = "unstable-2023-04-30";

  src = fetchFromGitHub {
```

v: stable -

```
owner = "canonical";
repo = "firmware-updater";
rev = "6e7dbdb64e344633ea62874b54ff3990bd3b8440";
sha256 = "sha256-s5mwtr5MSPqLMN+k851+pFIFFPa0N1hqz97ys050tFA=";
fetchSubmodules = true;
};

pubspecLockFile = ./pubspec.lock;
depsListFile = ./deps.json;
vendorHash = "sha256-cdM0+tr6kYiN5xKXa+uTMAcFf2C75F3wVPrn21G4QPQ=";
}
```

Dhall

[Remote imports](#)

[Packaging a Dhall expression from scratch](#)

[Contents of a Dhall package](#)

[Packaging functions](#)

[dhall-to-nixpkgs](#)

[Overriding dependency versions](#)

[Overrides](#)

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](#)

Remote imports

Nixpkgs bypasses Dhall's support for remote imports using Dhall's semantic integrity check.

v: stable -

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.

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:

v: stable -

```
$ dhall freeze --inplace ./true.dhall
```

... which gives us:

```
-- ./true.dhall

let Prelude =
    https://prelude.dhall-lang.org/v20.1.0/package.dhall
    sha256:26b0ef498663d269e4dc6a82b0ee289ec565d683ef4c00d0ebdd25333a!

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/94b2848550b12~0~d1~";
    hash     = "sha256-B4Q3c6IvTLg3Q92qYa8y+i4uTaphtFdjp+Ir3QQjdN0=" v: stable -
```

```
};

dhallOverlay = self: super: {
  true = self.callPackage ./true.nix { };
};

overlay = self: super: {
  dhallPackages = super.dhallPackages.override (old: {
    overrides =
      self.lib.composeExtensions (old.overrides or (_: _: {})) dhallOverlay
  });
};

pkgs = import nixpkgs { config = {}; overlays = [ overlay ]; };

in
pkgs
```

... which we can then build using this command:

```
$ nix build --file ./example.nix dhallPackages.true
```

Contents of a Dhall package

The above package produces the following directory tree:

```
$ tree -a ./result
result
├── .cache
│   └── dhall
│       └── 122027abdeddf8503496adeb623466caa47da5f63abd2bc6fa19f6cfcb73e
└── binary.dhall
└── source.dhall
```

... where:

v: stable -

- `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/122027abdeddf8503496adeb623466ca
True
```

- `binary.dhall` contains a Dhall expression which handles fetching and decoding the same cache product:

```
$ cat ./result/binary.dhall
missing sha256:27abdeddf8503496adeb623466caa47da5f63abd2bc6fa19f6cfcb73
$ 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
        └── 122026b0ef498663d269e4dc6a82b0ee289ec565d683ef4c00d0
```

```
└── 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
│       └── 122026b0ef498663d269e4dc6a82b0ee289ec565d683ef4c00d0ebdd25333a
└── 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) ->
```

v: stable -

```
List/fold Bool _ Bool (\(_ : Bool) -> \(_ : Bool) -> _@1 == _) True  
, fold =  
  \(_ : Bool) ->  
...  
...
```

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`

v: stable -

- **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 `hash` or `fetchSubmodules`.

dhall-to-nixpkgs

You can use the `dhall-to-nixpkgs` command-line utility to automate packaging Dhall code. For example:

```
$ nix-shell -p haskellPackages.dhall-nixpkgs nix-prefetch-git  
[nix-shell]$ dhall-to-nixpkgs github https://github.com/Gabriell v: stable -
```

```
{ buildDhallGitHubPackage, Prelude }:
buildDhallGitHubPackage {
  name = "dhall-semver";
  githubBase = "github.com";
  owner = "Gabriella439";
  repo = "dhall-semver";
  rev = "2d44ae605302ce5dc6c657a1216887fbb96392a4";
  fetchSubmodules = false;
  hash = "sha256-n0nQtswVapWi/x7or003MEYmAkt/a1uvl0tnje6GGnk=";
  directory = "";
  file = "package.dhall";
  source = false;
  document = false;
  dependencies = [ (Prelude.overridePackage { file = "package.dhall"; } ) ]
```

Note

`nix-prefetch-git` is added to the `nix-shell -p` invocation above, because it has to be in `$PATH` for `dhall-to-nixpkgs` to work.

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 = ~/proj/dhall-semver;
  file = "package.dhall";
  source = false;
  document = false;
  dependencies = [ (Prelude.overridePackage { file = "package.dhall"; } ) ]
```

Remote imports as fixed-output derivations

`dhall-to-nixpkgs` has the ability to fetch and build remote imports as fixed-output derivations by using their Dhall integrity check. This is sometimes easier than manually packaging all remote imports.

This can be used like the following:

```
$ dhall-to-nixpkgs directory --fixed-output-derivations ~/proj/dhall-semver
{ buildDhallDirectoryPackage, buildDhallUrl }:
buildDhallDirectoryPackage {
  name = "proj";
  src = ~/proj/dhall-semver;
  file = "package.dhall";
  source = false;
  document = false;
  dependencies = [
    (buildDhallUrl {
      url = "https://prelude.dhall-lang.org/v17.0.0/package.dhall";
      hash = "sha256-ENs8kZwl6QRoM9+Jeo/+JwHcOQ+git2VjDQwUkvlpD4=";
      dhallHash = "sha256:10db3c919c25e9046833df897a8ffe2701dc390fa0893c";
    })
  ];
}
```

Here, `dhall-semver`'s `Prelude` dependency is fetched and built with the `buildDhallUrl` helper function, instead of being passed in as a function argument.

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:eb693342eb769f782174157eba9b5924cf8ac6793897fc36a  v: stable -
```

```
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
```

Dhall was compiled without the 'with-http' flag.

The requested URL was: <https://prelude.dhall-lang.org/v19.0.0/package.dhall>

```
4|     https://prelude.dhall-lang.org/v19.0.0/package.dhall  
5|     sha256:eb693342eb769f782174157eba9b5924cf8ac6793897fc36a31cc1  
  
/nix/store/rsab4y99h14912h4zplqx2iizr5n4rc2-true.dhall:4:7  
[1 built (1 failed), 0.0 MiB DL]  
error: build of '/nix/store/0f1hla7ff1wiaqyk1r2ky4wnhnw114fi-true.drv' fa
```

... because the default Prelude selected by Nixpkgs revision

94b2848559b12a8ed1fe433084686b2a81123c99 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 global variable or by specifying the package's name in the `prelude` field of the `g` key in the `pkgs` section of the `configuration.nix` file:

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 { };
  };
};
```

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; };

  ...
};
```

Dotnet

[Local Development Workflow](#)

[dotnet-sdk vs dotnetCorePackages.sdk](#)

v: stable -

[dotnetCorePackages.sdk vs dotnetCorePackages.runtime vs dotnetCorePackages.aspnetcore](#)

[Packaging a Dotnet Application](#)

[Dotnet global tools](#)

Local Development Workflow

For local development, it's recommended to use nix-shell to create a dotnet environment:

```
# shell.nix
with import <nixpkgs> {};

mkShell {
  name = "dotnet-env";
  packages = [
    dotnet-sdk
  ];
}
```

Using many sdks in a workflow

It's very likely that more than one sdk will be needed on a given project. Dotnet provides several different frameworks (E.g dotnetcore, aspnetcore, etc.) as well as many versions for a given framework. Normally, dotnet is able to fetch a framework and install it relative to the executable. However, this would mean writing to the nix store in nixpkgs, which is read-only. To support the many-sdk use case, one can compose an environment using `dotnetCorePackages.combinePackages`:

```
with import <nixpkgs> {};

mkShell {
  name = "dotnet-env";
  packages = [
    (with dotnetCorePackages; combinePackages [
      sdk_6_0
    ])
  ];
}
```

v: stable -

```
sdk_7_0  
])  
];  
}
```

This will produce a dotnet installation that has the dotnet 6.0 7.0 sdk. The first sdk listed will have it's cli utility present in the resulting environment. Example info output:

```
$ dotnet --info  
.NET SDK:  
Version: 7.0.202  
Commit: 6c74320bc3  
  
Środowisko uruchomieniowe:  
OS Name: nixos  
OS Version: 23.05  
OS Platform: Linux  
RID: linux-x64  
Base Path: /nix/store/n2pm44xq20hz7ybsasgmd7p3yh31gnh4-dotnet-sdk-7.0.:  
  
Host:  
Version: 7.0.4  
Architecture: x64  
Commit: 0a396acafe  
  
.NET SDKs installed:  
6.0.407 [/nix/store/3b19303vwrhv0xxz1hg355c7f2hgxxgd-dotnet-core-combine  
7.0.202 [/nix/store/3b19303vwrhv0xxz1hg355c7f2hgxxgd-dotnet-core-combine  
  
.NET runtimes installed:  
Microsoft.AspNetCore.App 6.0.15 [/nix/store/3b19303vwrhv0xxz1hg355c7f2hgxxgd-dotnet-core-combine  
Microsoft.AspNetCore.App 7.0.4 [/nix/store/3b19303vwrhv0xxz1hg355c7f2hgxxgd-dotnet-core-combine  
Microsoft.NETCore.App 6.0.15 [/nix/store/3b19303vwrhv0xxz1hg355c7f2hgxxgd-dotnet-core-combine  
Microsoft.NETCore.App 7.0.4 [/nix/store/3b19303vwrhv0xxz1hg355c7f2hgxxgd-dotnet-core-combine  
  
Other architectures found:  
None v: stable -
```

Environment variables:

Not set

global.json file:

Not found

Learn more:

<https://aka.ms/dotnet/info>

Download .NET:

<https://aka.ms/dotnet/download>

dotnet-sdk vs dotnetCorePackages.sdk

The `dotnetCorePackages.sdk_X_Y` is preferred over the old `dotnet-sdk` as both major and minor version are very important for a dotnet environment. If a given minor version isn't present (or was changed), then this will likely break your ability to build a project.

dotnetCorePackages.sdk vs dotnetCorePackages.runtime vs dotnetCorePackages.aspnetcore

The `dotnetCorePackages.sdk` contains both a runtime and the full sdk of a given version. The `runtime` and `aspnetcore` packages are meant to serve as minimal runtimes to deploy alongside already built applications.

Packaging a Dotnet Application

To package Dotnet applications, you can use `buildDotnetModule`. This has similar arguments to `stdenv.mkDerivation`, with the following additions:

- `projectFile` is used for specifying the dotnet project file, relative to the source root. These have `.sln` (entire solution) or `.csproj` (single project) file extensions. This can be a list of multiple projects as well. When omitted, will attempt to find and build the solution (`.sln`). If running into problems, make sure to set it to a file (or a list of files) with the `.csproj` extension - building applications as entire solutions is not fully supported by the .NET CLI.

v: stable -

- **nugetDeps** takes either a path to a `deps.nix` file, or a derivation. The `deps.nix` file can be generated using the script attached to `passthru.fetch-deps`. This file can also be generated manually using `nuget-to-nix` tool, which is available in nixpkgs. If the argument is a derivation, it will be used directly and assume it has the same output as `mkNugetDeps`.
- **packNupkg** is used to pack project as a `nupkg`, and installs it to `$out/share`. If set to `true`, the derivation can be used as a dependency for another dotnet project by adding it to `projectReferences`.
- **projectReferences** can be used to resolve `ProjectReference` project items. Referenced projects can be packed with `buildDotnetModule` by setting the `packNupkg = true` attribute and passing a list of derivations to `projectReferences`. Since we are sharing referenced projects as NuGets they must be added to csproj/fsproj files as `PackageReference` as well. For example, your project has a local dependency:

```
<ProjectReference Include="..../foo/bar.fsproj" />
```

To enable discovery through `projectReferences` you would need to add:

```
<ProjectReference Include="..../foo/bar.fsproj" />
<PackageReference Include="bar" Version="*" Condition=" '$(ContinuousIntegration)' == 'true' "/>
```

- **executables** is used to specify which executables get wrapped to `$out/bin`, relative to `$out/lib/$pname`. If this is unset, all executables generated will get installed. If you do not want to install any, set this to `[]`. This gets done in the `prefixup` phase.
- **runtimeDeps** is used to wrap libraries into `LD_LIBRARY_PATH`. This is how dotnet usually handles runtime dependencies.
- **buildType** is used to change the type of build. Possible values are `Release`, `Debug`, etc. By default, this is set to `Release`.
- **selfContainedBuild** allows to enable the [self-contained](#) build flag. By default, it is set to false and generated applications have a dependency on the selected dotnet runtime. If enabled, the dotnet runtime is bundled into the executable and the built app has no dependency on .NET.

v: stable -

- `useAppHost` will enable creation of a binary executable that runs the .NET application using the specified root. More info in [Microsoft docs](#). Enabled by default.
- `useDotnetFromEnv` will change the binary wrapper so that it uses the .NET from the environment. The runtime specified by `dotnet-runtime` is given as a fallback in case no .NET is installed in the user's environment. This is most useful for .NET global tools and LSP servers, which often extend the .NET CLI and their runtime should match the users' .NET runtime.
- `dotnet-sdk` is useful in cases where you need to change what dotnet SDK is being used. You can also set this to the result of `dotnetSdkPackages.combinePackages`, if the project uses multiple SDKs to build.
- `dotnet-runtime` is useful in cases where you need to change what dotnet runtime is being used. This can be either a regular dotnet runtime, or an aspnetcore.
- `dotnet-test-sdk` is useful in cases where unit tests expect a different dotnet SDK. By default, this is set to the `dotnet-sdk` attribute.
- `testProjectFile` is useful in cases where the regular project file does not contain the unit tests. It gets restored and build, but not installed. You may need to regenerate your nuget lockfile after setting this. Note that if set, only tests from this project are executed.
- `disabledTests` is used to disable running specific unit tests. This gets passed as: `dotnet test --filter "FullyQualifiedName != {}"`, to ensure compatibility with all unit test frameworks.
- `dotnetRestoreFlags` can be used to pass flags to `dotnet restore`.
- `dotnetBuildFlags` can be used to pass flags to `dotnet build`.
- `dotnetTestFlags` can be used to pass flags to `dotnet test`. Used only if `doCheck` is set to `true`.
- `dotnetInstallFlags` can be used to pass flags to `dotnet install`.
- `dotnetPackFlags` can be used to pass flags to `dotnet pack`. Used only if `packNupkg` is set to `true`.
- `dotnetFlags` can be used to pass flags to all of the above phases.

v: stable -

When packaging a new application, you need to fetch its dependencies. Create an empty `deps.nix`, set `nugetDeps = ./deps.nix`, then run `nix-build -A package.fetch-deps` to generate a script that will build the lockfile for you.

Here is an example `default.nix`, using some of the previously discussed arguments:

```
{ lib, buildDotnetModule, dotnetCorePackages, ffmpeg }:

let
  referencedProject = import ../../bar { ... };
in buildDotnetModule rec {
  pname = "someDotnetApplication";
  version = "0.1";

  src = ./.;

  configFile = "src/project.sln";
  # File generated with `nix-build -A package.passthru.fetch-deps`.
  # To run fetch-deps when this file does not yet exist, set nugetDeps to
  nugetDeps = ./deps.nix;

  projectReferences = [ referencedProject ]; # `referencedProject` must co

  dotnet-sdk = dotnetCorePackages.sdk_6_0;
  dotnet-runtime = dotnetCorePackages.runtime_6_0;

  executables = [ "foo" ]; # This wraps "$out/lib/$pname/foo" to `'$out/bin
  executables = []; # Don't install any executables.

  packNupkg = true; # This packs the project as "foo-0.1.nupkg" at `'$out/`

  runtimeDeps = [ ffmpeg ]; # This will wrap ffmpeg's library path into `'$out/
}


```

Dotnet global tools

v: stable -

[.NET Global tools](#) are a mechanism provided by the dotnet CLI to install .NET binaries from NuGet packages.

They can be installed either as a global tool for the entire system, or as a local tool specific to project.

The local installation is the easiest and works on NixOS in the same way as on other Linux distributions.

[See dotnet documentation](#) to learn more.

[The global installation method](#) should also work most of the time. You have to remember to update the **PATH** value to the location the tools are installed to (the CLI will inform you about it during installation) and also set the **DOTNET_ROOT** value, so that the tool can find the .NET SDK package. You can find the path to the SDK by running `nix eval --raw nixpkgs#dotnet-sdk` (substitute the `dotnet-sdk` package for another if a different SDK version is needed).

This method is not recommended on NixOS, since it's not declarative and involves installing binaries not made for NixOS, which will not always work.

The third, and preferred way, is packaging the tool into a Nix derivation.

Packaging Dotnet global tools

Dotnet global tools are standard .NET binaries, just made available through a special NuGet package. Therefore, they can be built and packaged like every .NET application, using **buildDotnetModule**.

If however the source is not available or difficult to build, the **buildDotnetGlobalTool** helper can be used, which will package the tool straight from its NuGet package.

This helper has the same arguments as **buildDotnetModule**, with a few differences:

- **pname** and **version** are required, and will be used to find the NuGet package of the tool
- **nugetVersion** can be used to override the NuGet package name that will be downloaded, if it's different from **pname**
- **nugetSha256** is the hash of the fetched NuGet package. Set this to `lib.fakeHash256` for the first build, and it will error out, giving you the proper hash. Also remember to update it during version updates (it will not error out if you just change the version while having a fetched package in `/nix/store`)

v: stable -

- `dotnet-runtime` is set to `dotnet-sdk` by default. When changing this, remember that .NET tools fetched from NuGet require an SDK.

Here is an example of packaging `pbm`, an unfree binary without source available:

```
{ buildDotnetGlobalTool, lib }:

buildDotnetGlobalTool {
  pname = "pbm";
  version = "1.3.1";

  nugetSha256 = "sha256-ZG2HFyKYhVNvYd2kRlkbAjZJq880ADe3yjxmLuxXDUo=";

  meta = with lib;
    homepage = "https://cmd.petabridge.com/index.html";
    changelog = "https://cmd.petabridge.com/articles/RELEASE_NOTES.html";
    license = licenses.unfree;
    platforms = platforms.linux;
  };
}
```

When packaging a new .NET application in nixpkgs, you can tag the [@NixOS/dotnet](#) team for help and code review.

Emscripten

[Examples](#)

[Debugging](#)

[Emscripten](#): An LLVM-to-JavaScript Compiler

If you want to work with `emcc`, `emconfigure` and `emmake` as you are used to from Ubuntu and similar distributions,

v: stable -

```
nix-shell -p emscripten
```

A few things to note:

- `export EMCC_DEBUG=2` is nice for debugging
- The build artifact cache in `~/.emscripten` sometimes creates issues and needs to be removed from time to time

Examples

Let's see two different examples from `pkgs/top-level/emscripten-packages.nix`:

- `pkgs.zlib.override`
- `pkgs.buildEmscriptenPackage`

A special requirement of the `pkgs.buildEmscriptenPackage` is the `doCheck = true`. This means each Emscripten package requires that a [checkPhase](#) is implemented.

- Use `export EMCC_DEBUG=2` from within a phase to get more detailed debug output what is going wrong.
- The cache at `~/.emscripten` requires to set `HOME=$TMPDIR` in individual phases. This makes compilation slower but also more deterministic.

Example 236. Using `pkgs.zlib.override {}`

This example uses `zlib` from Nixpkgs, but instead of compiling **C** to **ELF** it compiles **C** to **JavaScript** since we were using `pkgs.zlib.override` and changed `stdenv` to `pkgs.emscriptenStdenv`.

A few adaptions and hacks were put in place to make it work. One advantage is that when `pkgs.zlib` is updated, it will automatically update this package as well.

```
(pkgs.zlib.override {  
    stdenv = pkgs.emscriptenStdenv;  
}).overrideAttrs
```

v: stable -

```
(old: rec {  
    buildInputs = old.buildInputs ++ [ pkg-config ];  
    # we need to reset this setting!  
    env = (old.env or { }) // { 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  
v: stable -
```

```
        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"'
';
})
```

Example 237. Using `pkgs.buildEmscriptenPackage {}`

This `xmlmirror` example features an Emscripten package that is defined completely from this context and no `pkgs.zlib.override` is used.

```
pkgs.buildEmscriptenPackage rec {
  name = "xmlmirror";

  buildInputs = [ pkg-config autoconf automake libtool gnumake libxml2 nodejs];
  nativeBuildInputs = [ pkg-config zlib ];

  src = pkgs.fetchgit {
    url = "https://gitlab.com/odfplugfest/xmlmirror.git";
    rev = "4fd7e86f7c9526b8f4c1733e5c8b45175860a8fd";
    hash = "sha256-i+QgY+5PYVg5pwhzcDnkfXAznBg3e8sWH2jZtixuWsk=";
  };

  configurePhase = ''
    rm -f fastXmlLint.js*
    # a fix for ERROR:root:For asm.js, TOTAL_MEMORY must be a multiple of
    # 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
  '';
```

v: stable -

```
sed -e "s/\$(JSONC_LDFLAGS) \$(ZLIB_LDFLAGS) \$(LIBINTL20_LDFLAGS)/\$(`  
# https://gitlab.com/odfplugfest/xmlmirror/issues/11  
sed -e "s/-o fastXmlLint.js/-s EXTRA_EXPORTED_RUNTIME_METHODS='[\"cca"  
'';  
  
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 = ''  
'';  
}
```

Debugging

v: stable -

Use `nix-shell -I nixpkgs=/some/dir/nixpkgs -A emscriptenPackages.libz` and from there you can go through 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...

GNOME

[Packaging GNOME applications](#)

[Onto wrapGAppsHook](#)

[Updating GNOME packages](#)

[Frequently encountered issues](#)

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

applications, environment variables needed to make the modules load, and finally a script that will do the work for us.

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.

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 transitively through `v: stable -`)

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).

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.

Icons

When an application uses icons, an icon theme should be available in `XdgDataDirs` 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 add their datadirs to `XdgIconDirs` environment variable (this is Nixpkgs specific, not actually a XDG standard variable). 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.

v: stable -

In the rare case you need to use icons from dependencies (e.g. when an app forces an icon theme), you can use the following to pick them up:

```
buildInputs = [
  pantheon.elementary-icon-theme
];
prefixup = ''
  gappsWrapperArgs+=(
    # The icon theme is hardcoded.
    --prefix XDG_DATA_DIRS : "$XDG_ICON_DIRS"
  )
'';
```

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.

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 the v: stable -

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.

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`.

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`.

Various plug-ins

If your application uses [GStreamer](#) or [Grilo](#), you should set `GST_PLUGIN_SYSTEM_PATH_1_0` and `GRL_PLUGIN_PATH`, respectively.

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 : "${gsettings-desktop-schemas}/share/gsettings" \
        --prefix XDG_DATA_DIRS : "${hicolor-icon-theme}/share" \
        --prefix GI_TYPELIB_PATH : "${lib.makeSearchPath "lib/girepository-"
done
'';
```

v: stable -

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`.
- `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 `wrapG/` v: stable -

- Setup hooks of `gst_all_1.gststreamer` 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"
)
'';
```

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.

Frequently encountered issues

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.

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.gschem.xml` and let the hooks handle the wrapping as [above](#).

When using wrapGAppsHook with special deriviers you can end up with double wrapped binaries.

This is because deriviers 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:

v: stable -

```
mkDerivation {  
  pname = "calibre";  
  version = "3.47.0";  
  
  nativeBuildInputs = [  
    wrapGAppsHook  
    qmake  
    ...  
  ];  
  
  dontWrapGApps = true;  
  
  # Arguments to be passed to `makeWrapper`, only used by qt5's mkDerivation  
  preFixup = ''  
    qtWrapperArgs+=("''${gappsWrapperArgs[@]}")  
  '';  
}
```

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 hardcoded 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, v: stable -

`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.

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.

Go

[Go modules](#)

[buildGoPackage \(legacy\)](#)

[Attributes used by the builders](#)

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.

Example for buildGoModule

In the following is an example expression using `buildGoModule`, the following arguments are of special significance to the function:

- `vendorHash`: is the hash of the output of the intermediate fetcher derivation.

v: stable -

`vendorHash` can also be set to `null`. In that case, rather than fetching the dependencies and vendoring them, the dependencies vendored in the source repo will be used.

To avoid updating this field when dependencies change, run `go mod vendor` in your source repo and set `vendorHash = null;`

To obtain the actual hash, set `vendorHash = lib.fakeHash;` and run the build ([more details here](#)).

- **proxyVendor**: Fetches (go mod download) and proxies 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 or if any dependency has case-insensitive conflicts which will produce platform-dependent `vendorHash` checksums.
- **modPostBuild**: Shell commands to run after the build of the goModules executes `go mod vendor`, and before calculating fixed output derivation's `vendorHash`. Note that if you change this attribute, you need to update `vendorHash` attribute.

```
pet = buildGoModule rec {
  pname = "pet";
  version = "0.3.4";

  src = fetchFromGitHub {
    owner = "knqyf263";
    repo = "pet";
    rev = "v${version}";
    hash = "sha256-Gjw1dRrgM8D3G7v6WIM2+50r4HmTXvx0Xxme2fH9TlQ=";
  };

  vendorHash = "sha256-ciBIR+a1oaYH+H1PcC8cD8ncfJczk1IiJ8iYNM+R6aA=";

  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 ];
  };
};
```

v: stable -

}

buildGoPackage (legacy)

The function `buildGoPackage` builds legacy Go programs, not supporting Go modules.

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}";
    hash = "sha256-XCPD4LNwtAd8uz7zyCLRfT8rzxycIUmTACjU03GnaeM=";
  };

  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`:

v: stable -

```
# 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 `hash` defined
    type = "git";
    url = "https://gopkg.in/yaml.v2";
    rev = "a83829b6f1293c91addabc89d0571c246397bbf4";
    hash = "sha256-EMrdy0M0tNu0cITaTAmT5/dPSKPXwHDKCXFpkGbVjdQ=";
  };
}
{
  goPackagePath = "github.com/docopt/docopt-go";
  fetch = {
    type = "git";
    url = "https://github.com/docopt/docopt-go";
    rev = "784ddc588536785e7299f7272f39101f7faccc3f";
    hash = "sha256-Uo89zjE+v3R7zz0q/gbQOHj3SMYt2W1nDHS7RCUi3M=";
  };
}
]
]
```

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
```

Attributes used by the builders

Many attributes [controlling the build phase](#) are respected by both `buildGoModule` and `buildGoPackage`. Note that `buildGoModule` reads the following attributes also when building the `vendor/` goModules fixed output derivation as well:

- [sourceRoot](#)
- [prePatch](#)
- [patches](#)
- [patchFlags](#)
- [postPatch](#)
- [preBuild](#)

In addition to the above attributes, and the many more variables respected also by `stdenv.mkDerivation`, both `buildGoModule` and `buildGoPackage` respect Go-specific attributes that tweak them to behave slightly differently:

ldflags

Arguments to pass to the Go linker tool via the `-ldflags` argument of `go build`. The most common use case for this argument is to make the resulting executable aware of its own version. For example:

```
ldflags = [
    "-X main.Version=${version}"
    "-X main.Commit=${version}"
];
```

tags

Arguments to pass to the Go via the `-tags` argument of `go build`. For example:

```
tags = [
```

v: stable -

```
"production"  
"sqlite"  
];
```

```
tags = [ "production" ] ++ lib.optionals withSqlite [ "sqlite" ];
```

deleteVendor

Removes the pre-existing vendor directory. This should only be used if the dependencies included in the vendor folder are broken or incomplete.

subPackages

Specified as a string or list of strings. Limits the builder from building child packages that have not been listed. If `subPackages` is not specified, all child packages will be built.

excludedPackages

Specified as a string or list of strings. Causes the builder to skip building child packages that match any of the provided values. If `excludedPackages` is not specified, all child packages will be built.

Haskell

[Available packages](#)

[haskellPackages.mkDerivation](#)

[Development environments](#)

[Overriding Haskell packages](#)

[F.A.Q.](#)

The Haskell infrastructure in Nixpkgs has two main purposes: The primary purpose is to pr v: stable -

Haskell compiler and build tools as well as infrastructure for packaging Haskell-based packages.

The secondary purpose is to provide support for Haskell development environments including prebuilt Haskell libraries. However, in this area sacrifices have been made due to self-imposed restrictions in Nixpkgs, to lessen the maintenance effort and to improve performance. (More details in the subsection [Limitations](#).)

Available packages

The compiler and most build tools are exposed at the top level:

- `ghc` is the default version of GHC
- Language specific tools: `cabal-install`, `stack`, `hpack`, ...

Many “normal” user facing packages written in Haskell, like `niv` or `cachix`, are also exposed at the top level, and there is nothing Haskell specific to installing and using them.

All of these packages are originally defined in the `haskellPackages` package set and are re-exposed with a reduced dependency closure for convenience. (see `justStaticExecutables` or `separateBinOutput` below)

The `haskellPackages` set includes at least one version of every package from Hackage as well as some manually injected packages. This amounts to a lot of packages, so it is hidden from `nix-env -qa` by default for performance reasons. You can still list all packages in the set like this:

```
$ nix-env -f '<nixpkgs>' -qaP -A haskellPackages
haskellPackages.a50
haskellPackages.AAI
haskellPackages.aasam
haskellPackages.abacate
haskellPackages.abc-puzzle
...
...
```

Also, the `haskellPackages` set is included on search.nixos.org.

The attribute names in `haskellPackages` always correspond with their name on Hackage.

v: stable -

Hackage allows names that are not valid Nix without escaping, you need to take care when handling attribute names like `3dmodels`.

For packages that are part of [Stackage](#) (a curated set of known to be compatible packages), we use the version prescribed by a Stackage snapshot (usually the current LTS one) as the default version. For all other packages we use the latest version from [Hackage](#) (the repository of basically all open source Haskell packages). See [below](#haskell-available- versions) for a few more details on this.

Roughly half of the 16K packages contained in `haskellPackages` don't actually build and are [marked as broken semi-automatically](#). Most of those packages are deprecated or unmaintained, but sometimes packages that should build, do not build. Very often fixing them is not a lot of work.

`haskellPackages` is built with our default compiler, but we also provide other releases of GHC and package sets built with them. You can list all available compilers like this:

```
$ nix-env -f '<nixpkgs>' -qaP -A haskell.compiler
haskell.compiler.ghc810          ghc-8.10.7
haskell.compiler.ghc88            ghc-8.8.4
haskell.compiler.ghc90            ghc-9.0.2
haskell.compiler.ghc924           ghc-9.2.4
haskell.compiler.ghc925           ghc-9.2.5
haskell.compiler.ghc926           ghc-9.2.6
haskell.compiler.ghc92           ghc-9.2.7
haskell.compiler.ghc942           ghc-9.4.2
haskell.compiler.ghc943           ghc-9.4.3
haskell.compiler.ghc94           ghc-9.4.4
haskell.compiler.ghcHEAD          ghc-9.7.20221224
haskell.compiler.ghc8102Binary    ghc-binary-8.10.2
haskell.compiler.ghc8102BinaryMinimal ghc-binary-8.10.2
haskell.compiler.ghc8107BinaryMinimal ghc-binary-8.10.7
haskell.compiler.ghc8107Binary    ghc-binary-8.10.7
haskell.compiler.ghc865Binary     ghc-binary-8.6.5
haskell.compiler.ghc924Binary     ghc-binary-9.2.4
haskell.compiler.ghc924BinaryMinimal ghc-binary-9.2.4
haskell.compiler.integer-simple.ghc810 ghc-integer-simple-8.10.7
haskell.compiler.integer-simple.ghc8107 ghc-integer-simple-8.10.7
haskell.compiler.integer-simple.ghc88 ghc-integer-simple-8.8. v: stable -
```

haskell.compiler.integer-simple.ghc884	ghc-integer-simple-8.8.4
haskell.compiler.native-bignum.ghc90	ghc-native-bignum-9.0.2
haskell.compiler.native-bignum.ghc902	ghc-native-bignum-9.0.2
haskell.compiler.native-bignum.ghc924	ghc-native-bignum-9.2.4
haskell.compiler.native-bignum.ghc925	ghc-native-bignum-9.2.5
haskell.compiler.native-bignum.ghc926	ghc-native-bignum-9.2.6
haskell.compiler.native-bignum.ghc92	ghc-native-bignum-9.2.7
haskell.compiler.native-bignum.ghc927	ghc-native-bignum-9.2.7
haskell.compiler.native-bignum.ghc942	ghc-native-bignum-9.4.2
haskell.compiler.native-bignum.ghc943	ghc-native-bignum-9.4.3
haskell.compiler.native-bignum.ghc94	ghc-native-bignum-9.4.4
haskell.compiler.native-bignum.ghc944	ghc-native-bignum-9.4.4
haskell.compiler.native-bignum.ghcHEAD	ghc-native-bignum-9.7.20221224
haskell.compiler.ghcjs	ghcjs-8.10.7

Each of those compiler versions has a corresponding attribute set built using it. However, the non-standard package sets are not tested regularly and, as a result, contain fewer working packages. The corresponding package set for GHC 9.4.5 is `haskell.packages.ghc945`. In fact `haskellPackages` is just an alias for `haskell.packages.ghc927`:

```
$ nix-env -f '<nixpkgs>' -qaP -A haskell.packages.ghc927
haskell.packages.ghc927.a50
haskell.packages.ghc927.AAI
haskell.packages.ghc927.aasam
haskell.packages.ghc927.abacate
haskell.packages.ghc927.abc-puzzle
...
...
```

Every package set also re-exposes the GHC used to build its packages as `haskell.packages.*.ghc`.

Available package versions

We aim for a “blessed” package set which only contains one version of each package, like [Stackage](#), which is a curated set of known to be compatible packages. We use the version information from Stackage snapshots and extend it with more packages. Normally in Nixpkgs the number of v: stable -

Haskell packages is roughly two to three times the size of Stackage. For choosing the version to use for a certain package we use the following rules:

1. By default, for `haskellPackages.foo` is the newest version of the package `foo` found on [Hackage](#), which is the central registry of all open source Haskell packages. Nixpkgs contains a reference to a pinned Hackage snapshot, thus we use the state of Hackage as of the last time we updated this pin.
2. If the [Stackage](#) snapshot that we use (usually the newest LTS snapshot) contains a package, [we use instead the version in the Stackage snapshot as default version for that package.](#)
3. For some packages, which are not on Stackage, we have if necessary [manual overrides to set the default version to a version older than the newest on Hackage.](#)
4. For all packages, for which the newest Hackage version is not the default version, there will also be a `haskellPackages.foo_x_y_z` package with the newest version. The `x_y_z` part encodes the version with dots replaced by underscores. When the newest version changes by a new release to Hackage the old package will disappear under that name and be replaced by a newer one under the name with the new version. The package name including the version will also disappear when the default version e.g. from Stackage catches up with the newest version from Hackage. E.g. if `haskellPackages.foo` gets updated from 1.0.0 to 1.1.0 the package `haskellPackages.foo_1_1_0` becomes obsolete and gets dropped.
5. For some packages, we also [manually add other `haskellPackages.foo_x_y_z` versions](#), if they are required for a certain build.

Relying on `haskellPackages.foo_x_y_z` attributes in derivations outside nixpkgs is discouraged because they may change or disappear with every package set update.

All `haskell.packages.*` package sets use the same package descriptions and the same sets of versions by default. There are however GHC version specific override `.nix` files to loosen this a bit.

Dependency resolution

Normally when you build Haskell packages with `cabal-install`, `cabal-install` does dependency resolution. It will look at all Haskell package versions known on Hackage and tries to pick for every (transitive) dependency of your build exactly one version. Those versions need to satisfy al v: stable -

constraints given in the `.cabal` file of your package and all its dependencies.

The [Haskell builder in nixpkgs](#) does no such thing. It will take as input packages with names off the desired dependencies and just check whether they fulfill the version bounds and fail if they don't (by default, see `jailbreak` to circumvent this).

The `haskellPackages.callPackage` function does the package resolution. It will, e.g., use `haskellPackages.aeson` which has the default version as described above for a package input of name `aeson`. (More general: `<packages>.callPackage f` will call `f` with named inputs provided from the package set `<packages>`.) While this is the default behavior, it is possible to override the dependencies for a specific package, see [override](#) and [overrideScope](#).

Limitations

Our main objective with `haskellPackages` is to package Haskell software in nixpkgs. This entails some limitations, partially due to self-imposed restrictions of nixpkgs, partially in the name of maintainability:

- Only the packages built with the default compiler see extensive testing of the whole package set. For other GHC versions only a few essential packages are tested and cached.
- As described above we only build one version of most packages.

The experience using an older or newer packaged compiler or using different versions may be worse, because builds will not be cached on `cache.nixos.org` or may fail.

Thus, to get the best experience, make sure that your project can be compiled using the default compiler of nixpkgs and recent versions of its dependencies.

A result of this setup is, that getting a valid build plan for a given package can sometimes be quite painful, and in fact this is where most of the maintenance work for `haskellPackages` is required. Besides that, it is not possible to get the dependencies of a legacy project from nixpkgs or to use a specific stack solver for compiling a project.

Even though we couldn't use them directly in nixpkgs, it would be desirable to have tooling to generate working Nix package sets from build plans generated by `cabal-install` or a specific Stackage snapshot via import-from-derivation. Sadly we currently don't have tooling for this. For this

v: stable -

be interested in the alternative [haskell.nix](#) framework, which, be warned, is completely incompatible with packages from `haskellPackages`.

haskellPackages.mkDerivation

Every haskell package set has its own haskell-aware `mkDerivation` which is used to build its packages. Generally you won't have to interact with this builder since [cabal2nix](#) can generate packages using it for an arbitrary cabal package definition. Still it is useful to know the parameters it takes when you need to [override](#) a generated Nix expression.

`haskellPackages.mkDerivation` is a wrapper around `stdenv.mkDerivation` which re-defines the default phases to be haskell aware and handles dependency specification, test suites, benchmarks etc. by compiling and invoking the package's `Setup.hs`. It does *not* use or invoke the `cabal-install` binary, but uses the underlying `Cabal` library instead.

General arguments

pname

Package name, assumed to be the same as on Hackage (if applicable)

version

Packaged version, assumed to be the same as on Hackage (if applicable)

src

Source of the package. If omitted, fetch package corresponding to `pname` and `version` from Hackage.

sha256

Hash to use for the default case of `src`.

revision

Revision number of the updated cabal file to fetch from Hackage. If `null` (which is the default value), the one included in `src` is used.

editedCabalFile

`sha256` hash of the cabal file identified by `revision` or `null`.

v: stable -

configureFlags

Extra flags passed when executing the `configure` command of `Setup.hs`.

buildFlags

Extra flags passed when executing the `build` command of `Setup.hs`.

haddockFlags

Extra flags passed to `Setup.hs` `haddock` when building the documentation.

doCheck

Whether to execute the package's test suite if it has one. Defaults to `true` unless cross-compiling.

doBenchmark

Whether to execute the package's benchmark if it has one. Defaults to `false`.

doHoogle

Whether to generate an index file for [hoogle](#) as part of `haddockPhase` by passing the [--hoogle option](#). Defaults to `true`.

doHaddockQuickjump

Whether to generate an index for interactive navigation of the HTML documentation. Defaults to `true` if supported.

doInstallIntermediates

Whether to install intermediate build products (files written to `dist/build` by GHC during the build process). With `enableSeparateIntermediatesOutput`, these files are instead installed to [a separate intermediates output](#). The output can then be passed into a future build of the same package with the `previousIntermediates` argument to support incremental builds. See ["Incremental builds"](#) for more information. Defaults to `false`.

enableLibraryProfiling

Whether to enable [profiling](#) for libraries contained in the package. Enabled by default if supported.

enableExecutableProfiling

Whether to enable [profiling](#) for executables contained in the package. Disabled by default.

profilingDetail

v: stable -

[Profiling detail level](#) to set. Defaults to **exported-functions**.

enableSharedExecutables

Whether to link executables dynamically. By default, executables are linked statically.

enableSharedLibraries

Whether to build shared Haskell libraries. This is enabled by default unless we are using `pkgsStatic` or shared libraries have been disabled in GHC.

enableStaticLibraries

Whether to build static libraries. Enabled by default if supported.

enableDeadCodeElimination

Whether to enable linker based dead code elimination in GHC. Enabled by default if supported.

enableHsc2hsViaAsm

Whether to pass `--via-asm` to `hsc2hs`. Enabled by default only on Windows.

hyperlinkSource

Whether to render the source as well as part of the haddock documentation by passing the [--hyperlinked-source flag](#). Defaults to `true`.

isExecutable

Whether the package contains an executable.

isLibrary

Whether the package contains a library.

jailbreak

Whether to execute [jailbreak-cabal](#) before `configurePhase` to lift any version constraints in the cabal file. Note that this can't lift version bounds if they are conditional, i.e. if a dependency is hidden behind a flag.

enableParallelBuilding

Whether to use the `-j` flag to make GHC/Cabal start multiple jobs in parallel.

maxBuildCores

Upper limit of jobs to use in parallel for compilation regardless of `$NIX_BUILD_CORES` v: stable -

to 16 as Haskell compilation with GHC currently sees a [performance regression](#) if too many parallel jobs are used.

doCoverage

Whether to generate and install files needed for [HPC](#). Defaults to `false`.

doHaddock

Whether to build (HTML) documentation using [haddock](#). Defaults to `true` if supported.

testTarget

Name of the test suite to build and run. If unset, all test suites will be executed.

preCompileBuildDriver

Shell code to run before compiling `Setup.hs`.

postCompileBuildDriver

Shell code to run after compiling `Setup.hs`.

preHaddock

Shell code to run before building documentation using `haddock`.

postHaddock

Shell code to run after building documentation using `haddock`.

coreSetup

Whether to only allow core libraries to be used while building `Setup.hs`. Defaults to `false`.

useCphs

Whether to enable the [cphs](#) preprocessor. Defaults to `false`.

enableSeparateBinOutput

Whether to install executables to a separate `bin` output. Defaults to `false`.

enableSeparateDataOutput

Whether to install data files shipped with the package to a separate `data` output. Defaults to `false`.

enableSeparateDocOutput

v: stable -

Whether to install documentation to a separate `doc` output. Is automatically enabled if `doHaddock` is `true`.

`enableSeparateIntermediatesOutput`

When `doInstallIntermediates` is true, whether to install intermediate build products to a separate `intermediates` output. See “[Incremental builds](#)” for more information. Defaults to `false`.

`allowInconsistentDependencies`

If enabled, allow multiple versions of the same Haskell package in the dependency tree at configure time. Often in such a situation compilation would later fail because of type mismatches. Defaults to `false`.

`enableLibraryForGhci`

Build and install a special object file for GHCi. This improves performance when loading the library in the REPL, but requires extra build time and disk space. Defaults to `false`.

`previousIntermediates`

If non-null, intermediate build artifacts are copied from this input to `dist/build` before performing compiling. See “[Incremental builds](#)” for more information. Defaults to `null`.

`buildTarget`

Name of the executable or library to build and install. If unset, all available targets are built and installed.

Specifying dependencies

Since `haskellPackages.mkDerivation` is intended to be generated from cabal files, it reflects cabal’s way of specifying dependencies. For one, dependencies are grouped by what part of the package they belong to. This helps to reduce the dependency closure of a derivation, for example benchmark dependencies are not included if `doBenchmark == false`.

`setup*Depends`

dependencies necessary to compile `Setup.hs`

`library*Depends`

dependencies of a library contained in the package

v: stable -

executable*Depends

dependencies of an executable contained in the package

test*Depends

dependencies of a test suite contained in the package

benchmark*Depends

dependencies of a benchmark contained in the package

The other categorization relates to the way the package depends on the dependency:

***ToolDepends**

Tools we need to run as part of the build process. They are added to the derivation's `nativeBuildInputs`.

***HaskellDepends**

Haskell libraries the package depends on. They are added to `propagatedBuildInputs`.

***SystemDepends**

Non-Haskell libraries the package depends on. They are added to `buildInputs`

***PkgconfigDepends**

`*SystemDepends` which are discovered using `pkg-config`. They are added to `buildInputs` and it is additionally ensured that `pkg-config` is available at build time.

***FrameworkDepends**

Apple SDK Framework which the package depends on when compiling it on Darwin.

Using these two distinctions, you should be able to categorize most of the dependency specifications that are available: `benchmarkFrameworkDepends`, `benchmarkHaskellDepends`, `benchmarkPkgconfigDepends`, `benchmarkSystemDepends`, `benchmarkToolDepends`, `executableFrameworkDepends`, `executableHaskellDepends`, `executablePkgconfigDepends`, `executableSystemDepends`, `executableToolDepends`, `libraryFrameworkDepends`, `libraryHaskellDepends`, `libraryPkgconfigDepends`, `librarySystemDepends`, `libraryToolDepends`, `setupHaskellDepends`, `testFrameworkDepends`, `testHaskellDepends`, `testPkgconfigDepends`, `testSystemDepends` and `testToolDepends`.

v: stable -

That only leaves the following extra ways for specifying dependencies:

`buildDepends`

Allows specifying Haskell dependencies which are added to `propagatedBuildInputs` unconditionally.

`buildTools`

Like `*ToolDepends`, but are added to `nativeBuildInputs` unconditionally.

`extraLibraries`

Like `*SystemDepends`, but are added to `buildInputs` unconditionally.

`pkg-configDepends`

Like `*PkgconfigDepends`, but are added to `buildInputs` unconditionally.

`testDepends`

Deprecated, use either `testHaskellDepends` or `testSystemDepends`.

`benchmarkDepends`

Deprecated, use either `benchmarkHaskellDepends` or `benchmarkSystemDepends`.

The dependency specification methods in this list which are unconditional are especially useful when writing `overrides` when you want to make sure that they are definitely included. However, it is recommended to use the more accurate ones listed above when possible.

Meta attributes

`haskellPackages.mkDerivation` accepts the following attributes as direct arguments which are transparently set in `meta` of the resulting derivation. See the [Meta-attributes section](#) for their documentation.

- These attributes are populated with a default value if omitted:
 - `homepage`: defaults to the Hackage page for `pname`.
 - `platforms`: defaults to `lib.platforms.all` (since GHC can cross-compile)
- These attributes are only set if given:

v: stable -

- **description**
- **license**
- **changelog**
- **maintainers**
- **broken**
- **hydraPlatforms**

Incremental builds

`haskellPackages.mkDerivation` supports incremental builds for GHC 9.4 and newer with the `doInstallIntermediates`, `enableSeparateIntermediatesOutput`, and `previousIntermediates` arguments.

The basic idea is to first perform a full build of the package in question, save its intermediate build products for later, and then copy those build products into the build directory of an incremental build performed later. Then, GHC will use those build artifacts to avoid recompiling unchanged modules.

For more detail on how to store and use incremental build products, see [Gabriella Gonzalez' blog post "Nixpkgs support for incremental Haskell builds"](#). motivation behind this feature.

An incremental build for [the `turtle` package](#) can be performed like so:

```
let
  pkgs = import <nixpkgs> {};
  inherit (pkgs) haskell;
  inherit (haskell.lib.compose) overrideCabal;

  # Incremental builds work with GHC >=9.4.
  turtle = haskell.packages.ghc944.turtle;

  # This will do a full build of `turtle`, while writing the intermediate
  # (compiled modules, etc.) to the `intermediates` output.
  turtle-full-build-with-incremental-output = overrideCabal (drv
    v: stable -
```

```
doInstallIntermediates = true;
enableSeparateIntermediatesOutput = true;
}) turtle;

# This will do an incremental build of `turtle` by copying the previous
# compiled modules and intermediate build products into the source tree
# before running the build.
#
# GHC will then naturally pick up and reuse these products, making this
# complete much more quickly than the previous one.
turtle-incremental-build = overrideCabal (drv: {
  previousIntermediates = turtle-full-build-with-incremental-output.inter;
}) turtle;
in
  turtle-incremental-build
```

Development environments

In addition to building and installing Haskell software, nixpkgs can also provide development environments for Haskell projects. This has the obvious advantage that you benefit from [cache.nixos.org](#) and no longer need to compile all project dependencies yourself. While it is often very useful, this is not the primary use case of our package set. Have a look at the section [available package versions](#) to learn which versions of packages we provide and the section [limitations](#), to judge whether a `haskellPackages` based development environment for your project is feasible.

By default, every derivation built using [`haskellPackages.mkDerivation`](#) exposes an environment suitable for building it interactively as the `env` attribute. For example, if you have a local checkout of `random`, you can enter a development environment for it like this (if the dependencies in the development and packaged version match):

```
$ cd ~/src/random
$ nix-shell -A haskellPackages.random.env '<nixpkgs>'
[nix-shell:~/src/random]$ ghc-pkg list
/nix/store/a8hh154xlzfizrhcf03c1l3f6l9l8qwv-ghc-9.2.4-with-packages/lib/glib
  Cabal-3.6.3.0
  array-0.5.4.0
                                         v: stable -
```

```
base-4.16.3.0
binary-0.8.9.0
...
ghc-9.2.4
...
```

As you can see, the environment contains a GHC which is set up so it finds all dependencies of `random`. Note that this environment does not mirror the environment used to build the package, but is intended as a convenient tool for development and simple debugging. `env` relies on the `ghcWithPackages` wrapper which automatically injects a pre-populated package-db into every GHC invocation. In contrast, using `nix-shell -A haskellPackages.random` will not result in an environment in which the dependencies are in GHCs package database. Instead, the Haskell builder will pass in all dependencies explicitly via configure flags.

`env` mirrors the normal derivation environment in one aspect: It does not include familiar development tools like `cabal-install`, since we rely on plain `Setup.hs` to build all packages. However, `cabal-install` will work as expected if in PATH (e.g. when installed globally and using a `nix-shell` without `--pure`). A declarative and pure way of adding arbitrary development tools is provided via [shellFor](#).

When using `cabal-install` for dependency resolution you need to be a bit careful to achieve build purity. `cabal-install` will find and use all dependencies installed from the packages `env` via Nix, but it will also consult Hackage to potentially download and compile dependencies if it can't find a valid build plan locally. To prevent this you can either never run `cabal update`, remove the cabal database from your `~/.cabal` folder or run `cabal` with `--offline`. Note though, that for some usecases `cabal2nix` needs the local Hackage db.

Often you won't work on a package that is already part of `haskellPackages` or Hackage, so we first need to write a Nix expression to obtain the development environment from. Luckily, we can generate one very easily from an already existing cabal file using `cabal2nix`:

```
$ ls
my-project.cabal src ...
$ cabal2nix ./ > my-project.nix
```

The generated Nix expression evaluates to a function ready to be `callPackage`-ed. For nix v: stable -

add a minimal `default.nix` which does just that:

```
# Retrieve nixpkgs impurely from NIX_PATH for now, you can pin it instead
{ pkgs ? import <nixpkgs> {} }:

# use the nixpkgs default haskell package set
pkgs.haskellPackages.callPackage ./my-project.nix { }
```

Using `nix-build default.nix` we can now build our project, but we can also enter a shell with all the package's dependencies available using `nix-shell -A env default.nix`. If you have `cabal-install` installed globally, it'll work inside the shell as expected.

shellFor

Having to install tools globally is obviously not great, especially if you want to provide a batteries-included `shell.nix` with your project. Luckily there's a proper tool for making development environments out of packages' build environments: `shellFor`, a function exposed by every haskell package set. It takes the following arguments and returns a derivation which is suitable as a development environment inside `nix-shell`:

packages

This argument is used to select the packages for which to build the development environment.

This should be a function which takes a haskell package set and returns a list of packages.

`shellFor` will pass the used package set to this function and include all dependencies of the returned package in the build environment. This means you can reuse Nix expressions of packages included in nixpkgs, but also use local Nix expressions like this: `hpkgs: [(hpkgs.callPackage ./my-project.nix { })]`.

nativeBuildInputs

Expects a list of derivations to add as build tools to the build environment. This is the place to add packages like `cabal-install`, `doctest` or `hlint`. Defaults to `[]`.

buildInputs

Expects a list of derivations to add as library dependencies, like `openssl`. This is rarely necessary as the haskell package expressions usually track system dependencies as well. Default v: stable

(see also [derivation dependencies](#))

withHoogle

If this is true, `hoogle` will be added to `nativeBuildInputs`. Additionally, its database will be populated with all included dependencies, so you'll be able search through the documentation of your dependencies. Defaults to `false`.

genericBuilderArgsModifier

This argument accepts a function allowing you to modify the arguments passed to `mkDerivation` in order to create the development environment. For example, `args: { doCheck = false; }` would cause the environment to not include any test dependencies. Defaults to `lib.id`.

doBenchmark

This is a shortcut for enabling `doBenchmark` via `genericBuilderArgsModifier`. Setting it to `true` will cause the development environment to include all benchmark dependencies which would be excluded by default. Defaults to `false`.

One neat property of `shellFor` is that it allows you to work on multiple packages using the same environment in conjunction with [cabal.project files](#). Say our example above depends on `distribution-nixpkgs` and we have a project file set up for both, we can add the following `shell.nix` expression:

```
{ pkgs ? import <nixpkgs> {} }:

pkgs.haskellPackages.shellFor {
  packages = hpkgs: [
    # reuse the nixpkgs for this package
    hpkgs.distribution-nixpkgs
    # call our generated Nix expression manually
    (hpkgs.callPackage ./my-project/my-project.nix { })
  ];

  # development tools we use
  nativeBuildInputs = [
    pkgs.cabal-install
```

v: stable -

```
pkgs.haskellPackages.doctest  
pkgs.cabal2nix  
];  
  
# Extra arguments are added to mkDerivation's arguments as-is.  
# Since it adds all passed arguments to the shell environment,  
# we can use this to set the environment variable the `Paths_`  
# module of distribution-nixpkgs uses to search for bundled  
# files.  
# See also: https://cabal.readthedocs.io/en/latest/cabal-package.html#distribution\_nixpkgs\_datadir = toString ./distribution-nixpkgs;  
}
```

haskell-language-server

To use HLS in short: Install `pkgs.haskell-language-server` e.g. in `nativeBuildInputs` in `shellFor` and use the `haskell-language-server-wrapper` command to run it. See the [HLS user guide](#) on how to configure your text editor to use HLS and how to test your setup.

HLS needs to be compiled with the GHC version of the project you use it on.

`pkgs.haskell-language-server` provides `haskell-language-server-wrapper`, `haskell-language-server` and `haskell-language-server-x.x.x` binaries, where `x.x.x` is the GHC version for which it is compiled. By default, it only includes binaries for the current GHC version, to reduce closure size. The closure size is large, because HLS needs to be dynamically linked to work reliably. You can override the list of supported GHC versions with e.g.

```
pkgs.haskell-language-server.override { supportedGhcVersions = [ "90" "94" ] }
```

Where all strings `version` are allowed such that `haskell.packages.ghc${version}` is an existing package set.

When you run `haskell-language-server-wrapper` it will detect the GHC version used by the project you are working on (by asking e.g. cabal or stack) and pick the appropriate versioned binary from your path.

v: stable -

Be careful when installing HLS globally and using a pinned nixpkgs for a Haskell project in a `nix-shell`. If the nixpkgs versions deviate too much (e.g., use different `glibc` versions) the `haskell-language-server-?.?.?` executable will try to detect these situations and refuse to start. It is recommended to obtain HLS via `nix-shell` from the nixpkgs version pinned in there instead.

The top level `pkgs.haskell-language-server` attribute is just a convenience wrapper to make it possible to install HLS for multiple GHC versions at the same time. If you know, that you only use one GHC version, e.g., in a project specific `nix-shell` you can use `pkgs.haskellPackages.haskell-language-server` or `pkgs.haskell.packages.*.haskell-language-server` from the package set you use.

If you use `nix-shell` for your development environments remember to start your editor in that environment. You may want to use something like `direnv` and/or an editor plugin to achieve this.

Overriding Haskell packages

Overriding a single package

Like many language specific subsystems in nixpkgs, the Haskell infrastructure also has its own quirks when it comes to overriding. Overriding of the *inputs* to a package at least follows the standard procedure. For example, imagine you need to build `nix-tree` with a more recent version of `brick` than the default one provided by `haskellPackages`:

```
haskellPackages.nix-tree.override {  
    brick = haskellPackages.brick_0_67;  
}
```

The custom interface comes into play when you want to override the arguments passed to `haskellPackages.mkDerivation`. For this, the function `overrideCabal` from `haskell.lib.compose` is used. E.g., if you want to install a man page that is distributed with the package, you can do something like this:

```
haskell.lib.compose.overrideCabal (drv: {  
    postInstall = ''  
        ${drv.postInstall or ""}  
})
```

v: stable -

```
install -Dm644 man/pnbackup.1 -t $out/share/man/man1
';
}) haskellPackages.pnbackup
```

`overrideCabal` takes two arguments:

1. A function which receives all arguments passed to `haskellPackages.mkDerivation` before and returns a set of arguments to replace (or add) with a new value.
2. The Haskell derivation to override.

The arguments are ordered so that you can easily create helper functions by making use of currying:

```
let
  installManPage = haskell.lib.compose.overrideCabal (drv: {
    postInstall = ''
      ${drv.postInstall or ""}
      install -Dm644 man/${drv.pname}.1 -t "$out/share/man/man1"
    '';
  });
in
installManPage haskellPackages.pnbackup
```

In fact, `haskell.lib.compose` already provides lots of useful helpers for common tasks, detailed in the next section. They are also structured in such a way that they can be combined using `lib.pipe`:

```
lib.pipe my-haskell-package [
  # lift version bounds on dependencies
  haskell.lib.compose.doJailbreak
  # disable building the haddock documentation
  haskell.lib.compose.dontHaddock
  # pass extra package flag to Cabal's configure step
  (haskell.lib.compose.enableCabalFlag "myflag")
]
```

v: stable -

haskell.lib.compose

The base interface for all overriding is the following function:

`overrideCabal f drv`

Takes the arguments passed to obtain `drv` to `f` and uses the resulting attribute set to update the argument set. Then a recomputed version of `drv` using the new argument set is returned.

All other helper functions are implemented in terms of `overrideCabal` and make common overrides shorter and more complicate ones trivial. The simple overrides which only change a single argument are only described very briefly in the following overview. Refer to the [documentation of `haskellPackages.mkDerivation`](#) for a more detailed description of the effects of the respective arguments.

Packaging Helpers

`overrideSrc { src, version } drv`

Replace the source used for building `drv` with the path or derivation given as `src`. The `version` attribute is optional. Prefer this function over overriding `src` via `overrideCabal`, since it also automatically takes care of removing any Hackage revisions.

`justStaticExecutables drv`

Only build and install the executables produced by `drv`, removing everything that may refer to other Haskell packages' store paths (like libraries and documentation). This dramatically reduces the closure size of the resulting derivation. Note that the executables are only statically linked against their Haskell dependencies, but will still link dynamically against libc, GMP and other system library dependencies. If dependencies use their Cabal-generated `Paths_*` module, this may not work as well if GHC's dead code elimination is unable to remove the references to the dependency's store path that module contains.

`enableSeparateBinOutput drv`

Install executables produced by `drv` to a separate `bin` output. This has a similar effect as `justStaticExecutables`, but preserves the libraries and documentation in the `out` output alongside the `bin` output with a much smaller closure size.

`markBroken drv`

Sets the `broken` flag to `true` for `drv`.

v: stable -

markUnbroken `drv`, **unmarkBroken** `drv`

Set the `broken` flag to `false` for `drv`.

doDistribute `drv`

Updates `hydraPlatforms` so that Hydra will build `drv`. This is sometimes necessary when working with versioned packages in `haskellPackages` which are not built by default.

dontDistribute `drv`

Sets `hydraPlatforms` to `[]`, causing Hydra to skip this package altogether. Useful if it fails to evaluate cleanly and is causing noise in the evaluation errors tab on Hydra.

Development Helpers**sdistTarball** `drv`

Create a source distribution tarball like those found on Hackage instead of building the package `drv`.

documentationTarball `drv`

Create a documentation tarball suitable for uploading to Hackage instead of building the package `drv`.

buildFromSdist `drv`

Uses `sdistTarball` `drv` as the source to compile `drv`. This helps to catch packaging bugs when building from a local directory, e.g. when required files are missing from `extra-source-files`.

failOnAllWarnings `drv`

Enables all warnings GHC supports and makes it fail the build if any of them are emitted.

enableDWARFDebugging `drv`

Compiles the package with additional debug symbols enabled, useful for debugging with e.g. `gdb`.

doStrip `drv`

Sets `doStrip` to `true` for `drv`.

dontStrip `drv`

Sets `doStrip` to `false` for `drv`.

v: stable -

Trivial Helpers

`doJailbreak` `drv`

Sets the `jailbreak` argument to `true` for `drv`.

`dontJailbreak` `drv`

Sets the `jailbreak` argument to `false` for `drv`.

`doHaddock` `drv`

Sets `doHaddock` to `true` for `drv`.

`dontHaddock` `drv`

Sets `doHaddock` to `false` for `drv`. Useful if the build of a package is failing because of e.g. a syntax error in the Haddock documentation.

`doHyperlinkSource` `drv`

Sets `hyperlinkSource` to `true` for `drv`.

`dontHyperlinkSource` `drv`

Sets `hyperlinkSource` to `false` for `drv`.

`doCheck` `drv`

Sets `doCheck` to `true` for `drv`.

`dontCheck` `drv`

Sets `doCheck` to `false` for `drv`. Useful if a package has a broken, flaky or otherwise problematic test suite breaking the build.

`appendConfigureFlags` `list` `drv`

Adds the strings in `list` to the `configureFlags` argument for `drv`.

`enableCabalFlag` `flag` `drv`

Makes sure that the Cabal flag `flag` is enabled in Cabal's configure step.

`disableCabalFlag` `flag` `drv`

Makes sure that the Cabal flag `flag` is disabled in Cabal's configure step.

`appendBuildFlags` `list` `drv`

Adds the strings in `list` to the `buildFlags` argument for `drv`.

v: stable -

appendPatches *list* *drv*

Adds the *list* of derivations or paths to the `patches` argument for *drv*.

addBuildTools *list* *drv*

Adds the *list* of derivations to the `buildTools` argument for *drv*.

addExtraLibraries *list* *drv*

Adds the *list* of derivations to the `extraLibraries` argument for *drv*.

addBuildDepends *list* *drv*

Adds the *list* of derivations to the `buildDepends` argument for *drv*.

addTestToolDepends *list* *drv*

Adds the *list* of derivations to the `testToolDepends` argument for *drv*.

addPkgconfigDepends *list* *drv*

Adds the *list* of derivations to the `pkg-configDepends` argument for *drv*.

addSetupDepends *list* *drv*

Adds the *list* of derivations to the `setupHaskellDepends` argument for *drv*.

doBenchmark *drv*

Set `doBenchmark` to `true` for *drv*. Useful if your development environment is missing the dependencies necessary for compiling the benchmark component.

dontBenchmark *drv*

Set `doBenchmark` to `false` for *drv*.

setBuildTargets *drv* *list*

Sets the `buildTarget` argument for *drv* so that the targets specified in *list* are built.

doCoverage *drv*

Sets the `doCoverage` argument to `true` for *drv*.

dontCoverage *drv*

Sets the `doCoverage` argument to `false` for *drv*.

enableExecutableProfiling *drv*

v: stable -

Sets the `enableExecutableProfiling` argument to `true` for `drv`.

`disableExecutableProfiling drv`

Sets the `enableExecutableProfiling` argument to `false` for `drv`.

`enableLibraryProfiling drv`

Sets the `enableLibraryProfiling` argument to `true` for `drv`.

`disableLibraryProfiling drv`

Sets the `enableLibraryProfiling` argument to `false` for `drv`.

Library functions in the Haskell package sets

Some library functions depend on packages from the Haskell package sets. Thus they are exposed from those instead of from `haskell.lib.compose` which can only access what is passed directly to it. When using the functions below, make sure that you are obtaining them from the same package set (`haskellPackages`, `haskell.packages.ghc944` etc.) as the packages you are working with or – even better – from the `self/final` fix point of your overlay to `haskellPackages`.

Note: Some functions like `shellFor` that are not intended for overriding per se, are omitted in this section.

`cabalSdist { src, name ? ... }`

Generates the Cabal sdist tarball for `src`, suitable for uploading to Hackage. Contrary to `haskell.lib.compose.sdistTarball`, it uses `cabal-install` over `Setup.hs`, so it is usually faster: No build dependencies need to be downloaded, and we can skip compiling `Setup.hs`.

`buildFromCabalSdist drv`

Build `drv`, but run its `src` attribute through `cabalSdist` first. Useful for catching files necessary for compilation that are missing from the sdist.

`generateOptparseApplicativeCompletions list drv`

Generate and install shell completion files for the installed executables whose names are given via `list`. The executables need to be using `optparse-applicative` for [this to work](#). Note that this feature is automatically disabled when cross-compiling, since it requires executing the binaries in question.

v: stable -

F.A.Q.

Why is topic X not covered in this section? Why is section Y missing?

We have been working on [moving the nixpkgs Haskell documentation back into the nixpkgs manual](#). Since this process has not been completed yet, you may find some topics missing here covered in the old [haskell4nix docs](#).

If you feel any important topic is not documented at all, feel free to comment on the issue linked above.

How to enable or disable profiling builds globally?

By default, Nixpkgs builds a profiling version of each Haskell library. The exception to this rule are some platforms where it is disabled due to concerns over output size. You may want to...

- ...enable profiling globally so that you can build a project you are working on with profiling ability giving you insight in the time spent across your code and code you depend on using [GHC's profiling feature](#).
- ...disable profiling (globally) to reduce the time spent building the profiling versions of libraries which a significant amount of build time is spent on (although they are not as expensive as the "normal" build of a Haskell library).

Note

The method described below affects the build of all libraries in the respective Haskell package set as well as GHC. If your choices differ from Nixpkgs' default for your (host) platform, you will lose the ability to substitute from the official binary cache.

If you are concerned about build times and thus want to disable profiling, it probably makes sense to use `haskell.lib.compose.disableLibraryProfiling` (see [the section called "Trivial Helpers"](#)) on the packages you are building locally while continuing to substitute their dependencies and GHC.

Since we need to change the profiling settings for the desired Haskell package set *and* GHC (as the core libraries like `base`, `filepath` etc. are bundled with GHC), it is recommended to use `overlays` for Nixpkgs to change them. Since the interrelated parts, i.e. the package set and GHC, are cor v: stable -

the Nixpkgs fixpoint, we need to modify them both in a way that preserves their connection (or else we'd have to wire it up again manually). This is achieved by changing GHC and the package set in separate overlays to prevent the package set from pulling in GHC from `prev`.

The result is two overlays like the ones shown below. Adjustable parts are annotated with comments, as are any optional or alternative ways to achieve the desired profiling settings without causing too many rebuilds.

```
let
```

```
# Name of the compiler and package set you want to change. If you are us  
# the default package set `haskellPackages`, you need to look up what ve  
# of GHC it currently uses (note that this is subject to change).  
ghcName = "ghc92";  
# Desired new setting  
enableProfiling = true;
```

```
in
```

```
[
```

```
# The first overlay modifies the GHC derivation so that it does or does  
# build profiling versions of the core libraries bundled with it. It is  
# recommended to only use such an overlay if you are enabling profiling  
# platform that doesn't by default, because compiling GHC from scratch :  
# quite expensive.
```

```
(final: prev:
```

```
let
```

```
  inherit (final) lib;
```

```
in
```

```
{
```

```
  haskell = lib.recursiveUpdate prev.haskell {  
    compiler.${ghcName} = prev.haskell.compiler.${ghcName}.override {  
      # Unfortunately, the GHC setting is named differently for historical  
      enableProfiledLibs = enableProfiling;  
    };  
  };  
})
```

v: stable -

```
(final: prev:  
let  
  inherit (final) lib;  
  haskellLib = final.haskell.lib.compose;  
in  
  
{  
  haskell = lib.recursiveUpdate prev.haskell {  
    packages.${ghcName} = prev.haskell.packages.${ghcName}.override {  
      overrides = hfinal: hprev: {  
        mkDerivation = args: hprev.mkDerivation (args // {  
          # Since we are forcing our ideas upon mkDerivation, this changes  
          # affect every package in the package set.  
          enableLibraryProfiling = enableProfiling;  
  
          # To actually use profiling on an executable, executable profiling  
          # needs to be enabled for the executable you want to profile.  
          # can either do this globally or...  
          enableExecutableProfiling = enableProfiling;  
        });  
  
        # ...only for the package that contains an executable you want to  
        # That saves on unnecessary rebuilds for packages that you only  
        # on for their library, but also contain executables (e.g. pandoc)  
        my-executable = haskellLib.enableExecutableProfiling hprev.my-executable;  
  
        # If you are disabling profiling to save on build time, but want  
        # retain the ability to substitute from the binary cache. Drop the  
        # override for mkDerivation above and instead have an override  
        # this for the specific packages you are building locally and want  
        # to make cheaper to build.  
        my-library = haskellLib.disableLibraryProfiling hprev.my-library;  
      };  
    };  
  };  
})  
]
```

v: stable -

Hy

[Installation](#)

Installation

Installation without packages

You can install `hy` via `nix-env` or by adding it to `configuration.nix` by referring to it as a `hy` attribute. This kind of installation adds `hy` to your environment and it successfully works with `python3`.

Caution

Packages that are installed with your `python` derivation, are not accessible by `hy` this way.

Installation with packages

Creating `hy` derivation with custom `python` packages is really simple and similar to the way that `python` does it. Attribute `hy` provides function `withPackages` that creates custom `hy` derivation with specified packages.

For example if you want to create shell with `matplotlib` and `numpy`, you can do it like so:

```
$ nix-shell -p "hy.withPackages (ps: with ps; [ numpy matplotlib ])"
```

Or if you want to extend your `configuration.nix`:

```
{ # ...
environment.systemPackages = with pkgs; [
  (hy.withPackages (py-packages: with py-packages; [ numpy matplotlib ])
];
}
```

v: stable -

Idris

[Installing Idris](#)

[Starting Idris with library support](#)

[Building an Idris project with Nix](#)

[Passing options to `idris` commands](#)

Installing Idris

The easiest way to get a working idris version is to install the `idris` attribute:

```
$ nix-env -f "<nixpkgs>" -iA 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
```

v: stable -

```
$ 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
```

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
```

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
```

v: stable -

```
, 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 name.
  # 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";
    hash = "sha256-h28F9EEPuval6zrfeE+0k1XGQJGwINnsJEG8yjWl7w=";
  };

  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 { })'
```

v: stable -

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
```

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.

iOS

[Deploying a proxy component wrapper exposing Xcode](#)

v: stable -

[Building an iOS application](#)

[Spawning simulator instances](#)

[Troubleshooting](#)

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.

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 {
```

v: stable -

```
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/Xc
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/X
lrwxr-xr-x 1 sander staff 14 1 jan 1970 xcrun -> /usr/bin/xcrun
```

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 bui
  scheme = null; # -scheme will correspond to the app name by de
  sdk = null; # null will set it to 'iphonesimulator` for simula
```

```
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 is also possible to adjust the `xcodewrapper` parameters. This is only needed in rare circumstances. In most cases the default values should suffice:

- Specifies which `xcodewrapper` 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 `xcodewrapper`. By default, it corresponds to the app name.

v: stable -

- 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 that 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 `xcodeBaseD: v: stable - r`

can be overridden to refer to a different Xcode version.

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
```

v: stable -

```
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.

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
```

Java

Ant-based Java packages are typically built from source as follows:

```
stdenv.mkDerivation {
  name = "...";
  src = fetchurl { ... };
```

v: stable -

```
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/irv: stable -

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
  ...
```

You can also specify what JDK your JRE should be based on, for example selecting a ‘headless’ build to avoid including a link to GTK+:

```
my_jre = pkgs.jre_minimal.override {
  jdk = jdk11_headless;
};
```

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.

v: stable -

Javascript

[Introduction](#)

[Getting unstuck / finding code examples](#)

[Tools overview](#)

[General principles](#)

[Javascript packages inside nixpkgs](#)

[Tool specific instructions](#)

[Outside Nixpkgs](#)

Introduction

This contains instructions on how to package javascript applications.

The various tools available will be listed in the [tools-overview](#). Some general principles for packaging will follow. Finally some tool specific instructions will be given.

Getting unstuck / finding code examples

If you find you are lacking inspiration for packing javascript applications, the links below might prove useful. Searching online for prior art can be helpful if you are running into solved problems.

Github

- Searching Nix files for `mkYarnPackage`: <https://github.com/search?q=mkYarnPackage+language%3ANix&type=code>
- Searching just `flake.nix` files for `mkYarnPackage`: https://github.com/search?q=mkYarnPackage+path%3A**%2Fflake.nix&type=code

Gitlab

v: stable -

- Searching Nix files for `mkYarnPackage`: <https://gitlab.com/search?scope=blobs&search=mkYarnPackage+extension%3Anix>
- Searching just `flake.nix` files for `mkYarnPackage`: <https://gitlab.com/search?scope=blobs&search=mkYarnPackage+filename%3Aflake.nix>

Tools overview

General principles

The following principles are given in order of importance with potential exceptions.

Try to use the same node version used upstream

It is often not documented which node version is used upstream, but if it is, try to use the same version when packaging.

This can be a problem if upstream is using the latest and greatest and you are trying to use an earlier version of node. Some cryptic errors regarding V8 may appear.

Try to respect the package manager originally used by upstream (and use the upstream lock file)

A lock file (`package-lock.json`, `yarn.lock`...) is supposed to make reproducible installations of `node_modules` for each tool.

Guidelines of package managers, recommend to commit those lock files to the repos. If a particular lock file is present, it is a strong indication of which package manager is used upstream.

It's better to try to use a Nix tool that understand the lock file. Using a different tool might give you hard to understand error because different packages have been installed. An example of problems that could arise can be found [here](#). Upstream use NPM, but this is an attempt to package it with `yarn2nix` (that uses `yarn.lock`).

Using a different tool forces to commit a lock file to the repository. Those files are fairly large, so when packaging for nixpkgs, this approach does not scale well.

Exceptions to this rule are:

- When you encounter one of the bugs from a Nix tool. In each of the tool specific instructions, known problems will be detailed. If you have a problem with a particular tool, then it's best to try another tool, even if this means you will have to recreate a lock file and commit it to nixpkgs. In general **yarn2nix** has less known problems and so a simple search in nixpkgs will reveal many `yarn.lock` files committed.
- Some lock files contain particular version of a package that has been pulled off NPM for some reason. In that case, you can recreate upstream lock (by removing the original and `npm install`, `yarn`, ...) and commit this to nixpkgs.
- The only tool that supports workspaces (a feature of NPM that helps manage sub-directories with different `package.json` from a single top level `package.json`) is **yarn2nix**. If upstream has workspaces you should try **yarn2nix**.

Try to use upstream `package.json`

Exceptions to this rule are:

- Sometimes the upstream repo assumes some dependencies be installed globally. In that case you can add them manually to the upstream `package.json` (`yarn add xxx` or `npm install xxx`, ...). Dependencies that are installed locally can be executed with `npx` for CLI tools. (e.g. `npx postcss ...`, this is how you can call those dependencies in the phases).
- Sometimes there is a version conflict between some dependency requirements. In that case you can fix a version by removing the `^`.
- Sometimes the script defined in the `package.json` does not work as is. Some scripts for example use CLI tools that might not be available, or cd in directory with a different `package.json` (for workspaces notably). In that case, it's perfectly fine to look at what the particular script is doing and break this down in the phases. In the build script you can see `build:*` calling in turns several other build scripts like `build:ui` or `build:server`. If one of those fails, you can try to separate those into,

```
yarn build:ui  
yarn build:server  
# OR
```

v: stable -

```
npm run build:ui  
npm run build:server
```

when you need to override a package.json. It's nice to use the one from the upstream source and do some explicit override. Here is an example:

```
patchedPackageJSON = final.runCommand "package.json" { } ''  
  ${jq}/bin/jq '.version = "0.4.0" |  
    .devDependencies["@jsdoc/cli"] = "^0.2.5"  
  ${sonar-src}/package.json > $out  
'';
```

You will still need to commit the modified version of the lock files, but at least the overrides are explicit for everyone to see.

Using node_modules directly

Each tool has an abstraction to just build the node_modules (dependencies) directory. You can always use the `stdenv.mkDerivation` with the node_modules to build the package (symlink the node_modules directory and then use the package build command). The node_modules abstraction can be also used to build some web framework frontends. For an example of this see how [plausible](#) is built. `mkYarnModules` to make the derivation containing node_modules. Then when building the frontend you can just symlink the node_modules directory.

Javascript packages inside nixpkgs

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.

v: stable -

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 [`pkgs/development/node-packages/overrides.nix`](#):

```
dat = prev.dat.override (oldAttrs: {
  buildInputs = [ final.node-gyp-build pkgs.libtool pkgs.autoconf pkgs.gcc ];
  meta = oldAttrs.meta // { broken = since "12"; };
});
```

Adding and Updating Javascript packages in nixpkgs

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:

```
./pkgs/development/node-packages/generate.sh
```

3. Build your new package to test your changes:

```
nix-build -A nodePackages.<new-or-updated-package>
```

v: stable -

To build against the latest stable Current Node.js version (e.g. 18.x):

```
nix-build -A nodePackages_latest.<new-or-updated-package>
```

If the package doesn't build, you may need to add an override as explained above.

4. If the package's name doesn't match any of the executables it provides, add an entry in [pkgs/development/node-packages/main-programs.nix](#). This will be the case for all scoped packages, e.g., `@angular/cli`.
5. Add and commit all modified and generated files.

For more information about the generation process, consult the [README.md](#) file of the `node2nix` tool.

To update NPM packages in nixpkgs, run the same `generate.sh` script:

```
./pkgs/development/node-packages/generate.sh
```

Git protocol error

Some packages may have Git dependencies from GitHub specified with `git://`. GitHub has [disabled unencrypted Git connections](#), so you may see the following error when running the generate script:

```
The unauthenticated git protocol on port 9418 is no longer supported
```

Use the following Git configuration to resolve the issue:

```
git config --global url."https://github.com/".insteadOf git://github.com/
```

Tool specific instructions

buildNpmPackage

`buildNpmPackage` allows you to package npm-based projects in Nixpkgs without the use of `npm`. Instead, you can use `node` to run the build scripts. This is particularly useful for projects that have complex build systems or dependencies that are difficult to handle with `npm`.

generated dependencies file (as used in [node2nix](#)). It works by utilizing npm's cache functionality – creating a reproducible cache that contains the dependencies of a project, and pointing npm to it.

Here's an example:

```
{ lib, buildNpmPackage, fetchFromGitHub }:

buildNpmPackage rec {
  pname = "flood";
  version = "4.7.0";

  src = fetchFromGitHub {
    owner = "jesec";
    repo = pname;
    rev = "v${version}";
    hash = "sha256-BR+ZGkBBfd0dSQqAvujsbgsEPFYw/ThrylxUb0ksYxM=";
  };

  npmDepsHash = "sha256-tuEfyePwlOy2/m0PdXbqJsk06IowvAP4DWg8xSZwbJw=";

  # The prepack script runs the build script, which we'd rather do in the
  npmPackFlags = [ "--ignore-scripts" ];

  NODE_OPTIONS = "--openssl-legacy-provider";

  meta = with lib;
  {
    description = "A modern web UI for various torrent clients with a Node";
    homepage = "https://flood.js.org";
    license = licenses/gpl3Only;
    maintainers = with maintainers; [ winter ];
  };
}
```

In the default `installPhase` set by `buildNpmPackage`, it uses `npm pack --json --dry-run` to decide what files to install in `$out/lib/node_modules/$name/`, where `$name` is the `name` string defined in the package's `package.json`. Additionally, the `bin` and `man` keys in the source's `meta` object are copied to the `nodeModuleMeta` object.

v: stable -

`package.json` are used to decide what binaries and manpages are supposed to be installed. If these are not defined, `npm pack` may miss some files, and no binaries will be produced.

Arguments

- `npmDepsHash`: The output hash of the dependencies for this project. Can be calculated in advance with [`prefetch-npm-deps`](#).
- `makeCacheWritable`: Whether to make the cache writable prior to installing dependencies. Don't set this unless npm tries to write to the cache directory, as it can slow down the build.
- `npmBuildScript`: The script to run to build the project. Defaults to "`build`".
- `npmWorkspace`: The workspace directory within the project to build and install.
- `dontNpmBuild`: Option to disable running the build script. Set to `true` if the package does not have a build script. Defaults to `false`. Alternatively, setting `buildPhase` explicitly also disables this.
- `dontNpmInstall`: Option to disable running `npm install`. Defaults to `false`. Alternatively, setting `installPhase` explicitly also disables this.
- `npmFlags`: Flags to pass to all npm commands.
- `npmInstallFlags`: Flags to pass to `npm ci`.
- `npmBuildFlags`: Flags to pass to `npm run ${npmBuildScript}`.
- `npmPackFlags`: Flags to pass to `npm pack`.
- `npmPruneFlags`: Flags to pass to `npm prune`. Defaults to the value of `npmInstallFlags`.
- `makeWrapperArgs`: Flags to pass to `makeWrapper`, added to executable calling the generated `.js` with `node` as an interpreter. These scripts are defined in `package.json`.
- `nodejs`: The `nodejs` package to build against, using the corresponding `npm` shipped with that version of `node`. Defaults to `pkgs.nodejs`.
- `npmDeps`: The dependencies used to build the npm package. Especially useful to not have to recompute workspace dependencies.

prefetch-npm-deps

`prefetch-npm-deps` is a Nixpkgs package that calculates the hash of the dependencies of an npm project ahead of time.

```
$ ls  
package.json package-lock.json index.js  
$ prefetch-npm-deps package-lock.json  
...  
sha256-AAAAAAAAAAAAAAAAAAAAAAA=
```

fetchNpmDeps

`fetchNpmDeps` is a Nix function that requires the following mandatory arguments:

- `src`: A directory / tarball with `package-lock.json` file
- `hash`: The output hash of the node dependencies defined in `package-lock.json`.

It returns a derivation with all `package-lock.json` dependencies downloaded into `$out/`, usable as an npm cache.

corepack

This package puts the corepack wrappers for pnpm and yarn in your PATH, and they will honor the `packageManager` setting in the `package.json`.

node2nix

Preparation

You will need to generate a Nix expression for the dependencies. Don't forget the `-l package-lock.json` if there is a lock file. Most probably you will need the `--development` to include the `devDependencies`

So the command will most likely be:

v: stable -

```
node2nix --development -l package-lock.json
```

See [node2nix docs](#) for more info.

Pitfalls

- If upstream package.json does not have a “version” attribute, `node2nix` will crash. You will need to add it like shown in [the package.json section](#).
- `node2nix` has some [bugs](#) related to working with lock files from NPM distributed with `nodejs_16`.
- `node2nix` does not like missing packages from NPM. If you see something like `Cannot resolve version: vue-loader-v16@undefined` then you might want to try another tool. The package might have been pulled off of NPM.

yarn2nix

Preparation

You will need at least a `yarn.lock` file. If upstream does not have one you need to generate it and reference it in your package definition.

If the downloaded files contain the `package.json` and `yarn.lock` files they can be used like this:

```
offlineCache = fetchYarnDeps {  
  yarnLock = src + "/yarn.lock";  
  hash = "....";  
};
```

mkYarnPackage

`mkYarnPackage` will by default try to generate a binary. For package only generating static assets (Svelte, Vue, React, WebPack, ...), you will need to explicitly override the build step with your instructions.

It's important to use the `--offline` flag. For example if your script is `"build": "some command"` in `v: stable -`

package.json use:

```
buildPhase = ''  
  export HOME=$(mktemp -d)  
  yarn --offline build  
'';
```

The dist phase is also trying to build a binary, the only way to override it is with:

```
distPhase = "true";
```

The configure phase can sometimes fail because it makes many assumptions which may not always apply. One common override is:

```
configurePhase = ''  
  ln -s $node_modules node_modules  
'';
```

or if you need a writeable node_modules directory:

```
configurePhase = ''  
  cp -r $node_modules node_modules  
  chmod +w node_modules  
'';
```

mkYarnModules

This will generate a derivation including the `node_modules` directory. If you have to build a derivation for an integrated web framework (rails, phoenix...), this is probably the easiest way.

Overriding dependency behavior

In the `mkYarnPackage` record the property `pkgConfig` can be used to override packages when you encounter problems building.

v: stable -

For instance, say your package is throwing errors when trying to invoke node-sass:

```
ENOENT: no such file or directory, scandir '/build/source/node_modules/nod
```

To fix this we will specify different versions of build inputs to use, as well as some post install steps to get the software built the way we want:

```
mkYarnPackage rec {
  pkgConfig = {
    node-sass = {
      buildInputs = with final; [ python libsass pkg-config ];
      postInstall = ''
        LIBSASS_EXT=auto yarn --offline run build
        rm build/config.gypi
      '';
    };
  };
}
```

Pitfalls

- If version is missing from upstream package.json, yarn will silently install nothing. In that case, you will need to override package.json as shown in the [package.json section](#)
- Having trouble with node-gyp? Try adding these lines to the `yarnPreBuild` steps:

```
yarnPreBuild = ''
  mkdir -p $HOME/.node-gyp/${nodejs.version}
  echo 9 > $HOME/.node-gyp/${nodejs.version}/installVersion
  ln -sfv ${nodejs}/include $HOME/.node-gyp/${nodejs.version}
  export npm_config_nodedir=${nodejs}
'';
```

- The `echo 9` step comes from this answer: <https://stackoverflow.com/a/49139496>

- Exporting the headers in `npm_config_nodedir` comes from this issue: <https://github.com/nodejs/node-gyp/issues/1191#issuecomment-301243919>

Outside Nixpkgs

There are some other tools available, which are written in the Nix language. These that can't be used inside Nixpkgs because they require [Import From Derivation](#), which is not allowed in Nixpkgs.

If you are packaging something outside Nixpkgs, consider the following:

npmlock2nix

[npmlock2nix](#) aims at building `node_modules` without code generation. It hasn't reached v1 yet, the API might be subject to change.

Pitfalls

There are some [problems with npm v7](#).

nix-npm-buildpackage

[nix-npm-buildpackage](#) aims at building `node_modules` without code generation. It hasn't reached v1 yet, the API might change. It supports both `package-lock.json` and `yarn.lock`.

Pitfalls

There are some [problems with npm v7](#).

lisp-modules

[Overview](#)

[The 90% use case example](#)

[Importing packages from Quicklisp](#)

[Defining packages manually inside Nixpkgs](#)

v: stable -

[Defining packages manually outside Nixpkgs](#)

[Overriding package attributes](#)

[Building Wrappers](#)

[Adding a new Lisp](#)

This document describes the Nixpkgs infrastructure for building Common Lisp systems that use [ASDF](#) (Another System Definition Facility). It lives in `pkgs/development/lisp-modules`.

Overview

The main entry point of the API are the Common Lisp implementation packages themselves (e.g. `abcl`, `ccl`, `clasp-common-lisp`, `clisp`, `ecl`, `sbcl`). They have the `pkgs` and `withPackages` attributes, which can be used to discover available packages and to build wrappers, respectively.

The `pkgs` attribute set contains packages that were automatically [imported](#) from Quicklisp, and any other [manually defined](#) ones. Not every package works for all the CL implementations (e.g. `nyxt` only makes sense for `sbcl`).

The `withPackages` function is of primary utility. It is used to build [runnable wrappers](#), with a pinned and pre-built [ASDF FASL](#) available in the ASDF environment variable, and `CL_SOURCE_REGISTRY/ASDF_OUTPUT_TRANSLATIONS` configured to [find the desired systems on runtime](#).

In addition, Lisps have the `withOverrides` function, which can be used to [substitute](#) any package in the scope of their `pkgs`. This will also be useful together with `overrideLispAttrs` when [dealing with slashy systems](#), because they should stay in the main package and be built by specifying the `systems` argument to `build-asdf-system`.

The 90% use case example

The most common way to use the library is to run ad-hoc wrappers like this:

```
nix-shell -p 'sbcl.withPackages (ps: with ps; [ alexandria ])'
```

v: stable -

Then, in a shell:

```
$ sbcl
* (load (sb-ext:posix-getenv "ASDF"))
* (asdf:load-system 'alexandria)
```

Also one can create a `pkgs.mkShell` environment in `shell.nix`/`flake.nix`:

```
let
  sbcl' = sbcl.withPackages (ps: [ ps.alexandria ]);
in mkShell {
  packages = [ sbcl' ];
}
```

Such a Lisp can be now used e.g. to compile your sources:

```
buildPhase = ''
  ${sbcl'}/bin/sbcl --load my-build-file.lisp
''
```

Importing packages from Quicklisp

To save some work of writing Nix expressions, there is a script that imports all the packages distributed by Quicklisp into `imported.nix`. This works by parsing its `releases.txt` and `systems.txt` files, which are published every couple of months on quicklisp.org.

The import process is implemented in the `import` directory as Common Lisp code in the `org.lispbuilds.nix` ASDF system. To run the script, one can execute `ql-import.lisp`:

```
cd pkgs/development/lisp-modules
nix-shell --run 'sbcl --script ql-import.lisp'
```

The script will:

v: stable -

1. Download the latest Quicklisp `systems.txt` and `releases.txt` files
2. Generate a temporary SQLite database of all QL systems in `packages.sqlite`
3. Generate an `imported.nix` file from the database

(The `packages.sqlite` file can be deleted at will, because it is regenerated each time the script runs.)

The maintainer's job is to:

1. Re-run the `ql-import.lisp` script when there is a new Quicklisp release
2. Add any missing native dependencies in `ql.nix`
3. For packages that still don't build, package them manually in `packages.nix`

Also, the `imported.nix` file **must not be edited manually!** It should only be generated as described in this section (by running `ql-import.lisp`).

Adding native dependencies

The Quicklisp files contain ASDF dependency data, but don't include native library (CFFI) dependencies, and, in the case of ABCL, Java dependencies.

The `ql.nix` file contains a long list of overrides, where these dependencies can be added.

Packages defined in `packages.nix` contain these dependencies naturally.

Trusting `systems.txt` and `releases.txt`

The previous implementation of `lisp-modules` didn't fully trust the Quicklisp data, because there were times where the dependencies specified were not complete and caused broken builds. It instead used a `nix-shell` environment to discover real dependencies by using the ASDF APIs.

The current implementation has chosen to trust this data, because it's faster to parse a text file than to build each system to generate its Nix file, and because that way packages can be mass-imported. Because of that, there may come a day where some packages will break, due to bugs in Quicklisp. In that case, the fix could be a manual override in `packages.nix` and `ql.nix`. v: stable -

A known fact is that Quicklisp doesn't include dependencies on slashy systems in its data. This is an example of a situation where such fixes were used, e.g. to replace the `systems` attribute of the affected packages. (See the definition of `iolib`).

Quirks

During Quicklisp import:

- `+` in names is converted to `_plus{_,}:cl+ssl->cl_plus_ssl`,
`alexandria+->alexandria_plus`
- `.` in names is converted to `_dot_:iolib.base->iolib_dot_base`
- names starting with a number have a `_` prepended (`3d-vectors->_3d-vectors`)
- `_` in names is converted to `__` for reversibility

Defining packages manually inside Nixpkgs

Packages that for some reason are not in Quicklisp, and so cannot be auto-imported, or don't work straight from the import, are defined in the `packages.nix` file.

In that file, use the `build-asdf-system` function, which is a wrapper around `mkDerivation` for building ASDF systems. Various other hacks are present, such as `build-with-compile-into-pwd` for systems which create files during compilation (such as `cl-unicode`).

The `build-asdf-system` function is documented [here](#). Also, `packages.nix` is full of examples of how to use it.

Defining packages manually outside Nixpkgs

Lisp derivations (`abcl`, `sbcl` etc.) also export the `buildASDFSystem` function, which is similar to `build-asdf-system` from `packages.nix`, but is part of the public API.

It takes the following arguments:

- `pname`: the package name

- **version**: the package version
- **src**: the package source
- **patches**: patches to apply to the source before build
- **nativeLibs**: native libraries used by CFFI and grovelling
- **javaLibs**: Java libraries for ABCL
- **lispLibs**: dependencies on other packages build with `buildASDFSystem`
- **systems**: list of systems to build

It can be used to define packages outside Nixpkgs, and, for example, add them into the package scope with `withOverrides`.

Including an external package in scope

A package defined outside Nixpkgs using `buildASDFSystem` can be woven into the Nixpkgs-provided scope like this:

```
let
  alexandria = sbcl.buildASDFSystem rec {
    pname = "alexandria";
    version = "1.4";
    src = fetchFromGitLab {
      domain = "gitlab.common-lisp.net";
      owner = "alexandria";
      repo = "alexandria";
      rev = "v${version}";
      hash = "sha256-1Hzxt65dZvg0FIljjjlSGgKYkj+YBLwJCACi5DZsKmQ=";
    };
  };
  sbcl' = sbcl.withOverrides (self: super: {
    inherit alexandria;
  });
in sbcl'.pkgs.alexandria
```

v: stable -

Overriding package attributes

Packages export the `overrideLispAttrs` function, which can be used to build a new package with different parameters.

Example of overriding `alexandria`:

```
sbcl.pkgs.alexandria.overrideLispAttrs (oldAttrs: rec {  
    version = "1.4";  
    src = fetchFromGitLab {  
        domain = "gitlab.common-lisp.net";  
        owner = "alexandria";  
        repo = "alexandria";  
        rev = "v${version}";  
        hash = "sha256-1Hzxt65dZvg0FIljjjlSGgKYkj+YBLwJCACi5DZsKmQ=";  
    };  
})
```

Dealing with slashy systems

Slashy (secondary) systems should not exist in their own packages! Instead, they should be included in the parent package as an extra entry in the `systems` argument to the `build-asdf-system/buildASDFSystem` functions.

The reason is that ASDF searches for a secondary system in the `.asd` of the parent package. Thus, having them separate would cause either one of them not to load cleanly, because one will contain FASLs of itself but not the other, and vice versa.

To package slashy systems, use `overrideLispAttrs`, like so:

```
ecl.pkgs.alexandria.overrideLispAttrs (oldAttrs: {  
    systems = oldAttrs.systems ++ [ "alexandria/tests" ];  
    lispLibs = oldAttrs.lispLibs ++ [ ecl.pkgs.rt ];  
})
```

See the [respective section](#) on using `withOverrides` for how to weave it back into `ecl.p` v: stable -

Note that sometimes the slashy systems might not only have more dependencies than the main one, but create a circular dependency between `.asd` files. Unfortunately, in this case an adhoc solution becomes necessary.

Building Wrappers

Wrappers can be built using the `withPackages` function of Common Lisp implementations (`abcl`, `ecl`, `sbcl` etc.):

```
nix-shell -p 'sbcl.withPackages (ps: [ ps.alexandria ps.bordeaux-threads ] )'
```

Such a wrapper can then be used like this:

```
$ sbcl
* (load (sb-ext:posix-getenv "ASDF"))
* (asdf:load-system 'alexandria)
* (asdf:load-system 'bordeaux-threads)
```

Loading ASDF

For best results, avoid calling (`require 'asdf`) When using the library-generated wrappers.

Use `(load (ext:getenv "ASDF"))` instead, supplying your implementation's way of getting an environment variable for `ext:getenv`. This will load the (pre-compiled to FASL) Nixpkgs-provided version of ASDF.

Loading systems

There, you can use `asdf:load-system`. This works by setting the right values for the `CL_SOURCE_REGISTRY/ASDF_OUTPUT_TRANSLATIONS` environment variables, so that systems are found in the Nix store and pre-compiled FASLs are loaded.

Adding a new Lisp

The function `wrapLisp` is used to wrap Common Lisp implementations. It adds the `pkgs`, `v: stable` -

`withPackages`, `withOverrides` and `buildASDFSystem` attributes to the derivation.

`wrapLisp` takes these arguments:

- `pkg`: the Lisp package
- `faslExt`: Implementation-specific extension for FASL files
- `program`: The name of executable file in `${pkg}/bin/` (Default: `pkg.pname`)
- `flags`: A list of flags to always pass to `program` (Default: `[]`)
- `asdf`: The ASDF version to use (Default: `pkgs.asdf_3_3`)
- `packageOverrides`: Package overrides config (Default: `(self: super: {})`)

This example wraps CLISP:

```
wrapLisp {  
  pkg = clisp;  
  faslExt = "fas";  
  flags = ["-E" "UTF8"];  
}
```

User's Guide to Lua Infrastructure

[Using Lua](#)

[Developing with Lua](#)

[Lua Reference](#)

Using Lua

Overview of Lua

Several versions of the Lua interpreter are available: luajit, lua 5.1, 5.2, 5.3. The attribute `luajit v: stable` -

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`.

Installing Lua and packages

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 luafsystem ])
```

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`, `luafsystem`) that you added to the environment.

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`,

```
{ # ...
```

v: stable -

```
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.

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 ]))  
];  
}
```

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";  
            }  
        };  
    };  
}
```

v: stable -

```
src = /home/my_luarocks/repository;
});

};

luaPackages = lua.pkgs;
}
```

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 luaf filesystem ])'
```

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.luaf filesystem
```

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.

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.

Packaging a library on luarocks

v: stable -

[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).

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 the package **luarocks-packages-updater**:

```
nix-shell -p luarocks-packages-updater --run luarocks-packages-updater
```

[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 to the overrides.nix.

You can try converting luarocks packages to nix packages with the command **nix-shell -p luarocks-nix** and then **luarocks nix PKG_NAME**.

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](#).

Lua Reference

Lua interpreters

Versions 5.1, 5.2, 5.3 and 5.4 of the lua interpreter are available as respectively **lua5_1**, **lua5_2**, **lua5_3** and **lua5_4**. Luajit is available too. The Nix expressions for the interpreters can be found in [pkgs/development/interpreters/lua-5](#). v: stable -

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**.

buildLuarocksPackage function

The **buildLuarocksPackage** function is implemented in `pkgs/development/interpreters/lua-5/build-luarocks-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/moonscript/34.0.4-1/luaposix";  
        hash     = "sha256-4mLJG8n4m6y4Fqd0meUDfs0b9RHSR0qa/KD5KCwrNXs=";  
    };  
    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";  
    };  
};
```

v: stable -

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.

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.

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 `lua` v: stable -

lua5_2:

```
with import <nixpkgs> {};  
  
lua5_2.withPackages (ps: [ps.lua])
```

Now, `ps` is set to `lua52Packages`, matching the version of the interpreter.

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

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.`

Maven

[Building a package using `maven.buildMavenPackage`](#)

[Manually using `mvn2nix`](#)

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.

Building a package using maven.buildMavenPackage

Consider the following package:

```
{ lib, fetchFromGitHub, jre, makeWrapper, maven }:

maven.buildMavenPackage rec {
  pname = "jd-cli";
  version = "1.2.1";

  src = fetchFromGitHub {
    owner = "intoolsweTrust";
    repo = pname;
    rev = "${pname}-${version}";
    hash = "sha256-rRttA5H0A0c44loBzbKH7Waoted3Is0gxGCD2VM0U/Q=";
  };

  mvnHash = "sha256-kLpjMj05uC94/5vGMwMlFzLKNFOKeyNvq/vmB6pHTAo=";

  nativeBuildInputs = [ makeWrapper ];

  installPhase = ''
    mkdir -p $out/bin $out/share/jd-cli
    install -Dm644 jd-cli/target/jd-cli.jar $out/share/jd-cli

    makeWrapper ${jre}/bin/java $out/bin/jd-cli \
      --add-flags "-jar $out/share/jd-cli/jd-cli.jar"
  '';

  meta = with lib;
  {
    description = "Simple command line wrapper around JD Core Java Decompiler";
    homepage = "https://github.com/intoolsweTrust/jd-cli";
    license = licenses.gpl3Plus;
    maintainers = with maintainers; [ majiir ];
  };
}
```

v: stable -

This package calls `maven.buildMavenPackage` to do its work. The primary difference from `stdenv.mkDerivation` is the `mvnHash` variable, which is a hash of all of the Maven dependencies.

Tip

After setting `maven.buildMavenPackage`, we then do standard Java `.jar` installation by saving the `.jar` to `$out/share/java` and then making a wrapper which allows executing that file; see [the section called “Java”](#) for additional generic information about packaging Java applications.

Stable Maven plugins

Maven defines default versions for its core plugins, e.g. `maven-compiler-plugin`. If your project does not override these versions, an upgrade of Maven will change the version of the used plugins, and therefore the derivation and hash.

When `maven` is upgraded, `mvnHash` for the derivation must be updated as well: otherwise, the project will be built on the derivation of old plugins, and fail because the requested plugins are missing.

This clearly prevents automatic upgrades of Maven: a manual effort must be made throughout nixpkgs by any maintainer wishing to push the upgrades.

To make sure that your package does not add extra manual effort when upgrading Maven, explicitly define versions for all plugins. You can check if this is the case by adding the following plugin to your (parent) POM:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>3.3.0</version>
  <executions>
    <execution>
      <id>enforce-plugin-versions</id>
      <goals>
        <goal>enforce</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

v: stable -

```
<configuration>
  <rules>
    <requirePluginVersions />
  </rules>
</configuration>
</execution>
</executions>
</plugin>
```

Manually using mvn2nix

Warning

This way is no longer recommended; see [the section called “Building a package using maven.buildMavenPackage”](#) for the simpler and preferred way.

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.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <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>

```

v: stable -

```
</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>.

Solving for dependencies

buildMaven with NixOS/mvn2nix-maven-plugin

`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-SNAPSHOT"
  }
}
```

v: stable -

```
"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/d027c8f2cnmj5yrynpbq2s6wmc9c
│           └── antlr-2.7.2.pom -> /nix/store/mv42fc5gizl8h5g5vpywz1nfiynr
└── avalon-framework
    └── avalon-framework
        └── 4.1.3
            └── avalon-framework-4.1.3.jar -> /nix/store/iv5fp39v: stable -
```

Double Invocation

Note

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](#).

Note

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 \
```

v: stable -

```
-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.

Warning

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 than using `buildMaven`.

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-exa";
  buildInputs = [ maven ];

  buildPhase = ''
    echo "Using repository ${repository}"
    mvn --offline -Dmaven.repo.local=${repository} package;
  '';

  installPhase = ''
    install -Dm644 target/${pname}-${version}.jar $out/share/java
  '';
}
```

Tip

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
```

v: stable -

2 directories, 1 file

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 than building an [UberJar](#) which may be the more traditional approach.

CLASSPATH

This method 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"
    v: stable -
```

```
nativeBuildInputs = [ makeWrapper ];
buildInputs = [ maven ];

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"
';
}

}
```

MANIFEST file via Maven Plugin

This method 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>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```
<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-repository.jar/META-INF/MANIFEST.MF
```

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";
```

v: stable -

```
src = builtins.fetchTarball
  "https://github.com/fzakaria/nixos-maven-example/archive/main.tar.gz"
nativeBuildInputs = [ makeWrapper ];
buildInputs = [ maven ];

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"
';
}
```

Note

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-ma
v: stable - ::
```

```
└─ share
    └─ java
        └─ maven-demo-1.0.jar
```

```
› $(nix-build --no-out-link --option tarball-ttl 1 runnable-jar.nix)/bin/r
NixOS 😊 is super cool 😊!
```

Nim

Overview

[Nim program packages in Nixpkgs](#)

[Nim library packages in Nixpkgs](#)

[buildNimPackage parameters](#)

Overview

The Nim compiler, a builder function, and some packaged libraries are available in Nixpkgs. Until now each compiler release has been effectively backwards compatible so only the latest version is available.

Nim program packages in Nixpkgs

Nim programs can be built using `nimPackages.buildNimPackage`. In the case of packages not containing exported library code the attribute `nimBinOnly` should be set to `true`.

The following example shows a Nim program that depends only on Nim libraries:

```
{ lib, nimPackages, fetchFromGitHub }:

nimPackages.buildNimPackage (finalAttrs: {
    pname = "ttop";
    version = "1.0.1";
    nimBinOnly = true;
```

v: stable -

```
src = fetchFromGitHub {  
    owner = "inv2004";  
    repo = "ttop";  
    rev = "v${finalAttrs.version}";  
    hash = "sha256-x4Uczksh6p3XX/IMr0FtBxIleVHdAPX9e8n32VAUTC4=";  
};  
  
buildInputs = with nimPackages; [ asciigraph illwill parsetoml zippy ];  
})
```

Nim library packages in Nixpkgs

Nim libraries can also be built using `nimPackages.buildNimPackage`, but often the product of a fetcher is sufficient to satisfy a dependency. The `fetchgit`, `fetchFromGitHub`, and `fetchNimble` functions yield an output that can be discovered during the `configurePhase` of `buildNimPackage`.

Nim library packages are listed in [pkgs/top-level/nim-packages.nix](#) and implemented at [pkgs/development/nim-packages](#).

The following example shows a Nim library that propagates a dependency on a non-Nim package:

```
{ lib, buildNimPackage, fetchNimble, SDL2 }:  
  
buildNimPackage (finalAttrs: {  
    pname = "sdl2";  
    version = "2.0.4";  
    src = fetchNimble {  
        inherit (finalAttrs) pname version;  
        hash = "sha256-Vtcj8goI4zZPQs2TbFoBF1cR5UqDt0ldaXSH/+/xULk=";  
    };  
    propagatedBuildInputs = [ SDL2 ];  
})
```

buildNimPackage parameters

v: stable -

All parameters from `stdenv.mkDerivation` function are still supported. The following are specific to `buildNimPackage`:

- `nimBinOnly ? false`: If `true` then build only the programs listed in the Nimble file in the packages sources.
- `nimbleFile`: Specify the Nimble file location of the package being built rather than discover the file at build-time.
- `nimRelease ? true`: Build the package in *release* mode.
- `nimDefines ? []`: A list of Nim defines. Key-value tuples are not supported.
- `nimFlags ? []`: A list of command line arguments to pass to the Nim compiler. Use this to specify defines with arguments in the form of `-d:${name}=${value}`.
- `nimDoc ? false`: Build and install HTML documentation.
- `buildInputs ? []`: The packages listed here will be searched for `*.nimble` files which are used to populate the Nim library path. Otherwise the standard behavior is in effect.

OCaml

[User guide](#)

[Packaging guide](#)

User guide

OCaml libraries are available in attribute sets of the form `ocaml-ng.ocamlPackages_X_XX` where X is to be replaced with the desired compiler version. For example, `ocamlgraph` compiled with OCaml 4.12 can be found in `ocaml-ng.ocamlPackages_4_12.ocamlgraph`. The compiler itself is also located in this set, under the name `ocaml`.

If you don't care about the exact compiler version, `ocamlPackages` is a top-level alias pointing to a recent version of OCaml.

v: stable -

OCaml applications are usually available top-level, and not inside `ocamlPackages`. Notable exceptions are build tools that must be built with the same compiler version as the compiler you intend to use like `dune` or `ocaml-lsp`.

To open a shell able to build a typical OCaml project, put the dependencies in `buildInputs` and add `ocamlPackages.ocaml` and `ocamlPackages.findlib` to `nativeBuildInputs` at least. For example:

```
let
  pkgs = import <nixpkgs> {};
  # choose the ocaml version you want to use
  ocamlPackages = pkgs.ocaml-ng.ocamlPackages_4_12;
in
pkgs.mkShell {
  # build tools
  nativeBuildInputs = with ocamlPackages; [ ocaml findlib dune_2 ocaml-lsp ];
  # dependencies
  buildInputs = with ocamlPackages; [ ocamlgraph ];
}
```

Packaging guide

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 `minimalOCamlVersion` (see note below) 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.

v: stable -

- It also accept `duneVersion` parameter (valid value are "1", "2", and "3"). The recommended practice it to set only if you don't want the default value and/or it depends on something else like package version. You might see a not-supported argument `useDune2`. The behavior was `useDune2 = true; => duneVersion = "2";` and `useDune2 = false; => duneVersion = "1";`. It was used at the time when `dune3` didn't existed.
- It sets the optional `doCheck` attribute such that tests will be run with `dune runtest -p angstrom` after the build (`dune build -p angstrom`) is complete, but only if the Ocaml version is at least "4.05".
- It uses the package `ocaml-syntax-shims` as a build input, `alcotest` and `ppx_let` as check inputs (because they are 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.

```
{ lib,
  fetchFromGitHub,
  buildDunePackage,
  ocaml,
  ocaml-syntax-shims,
  alcotest,
  result,
  bigstringaf,
  ppx_let }:  
  
buildDunePackage rec {
  pname = "angstrom";
  version = "0.15.0";  
  
  minimalOCamlVersion = "4.04";  
  
  src = fetchFromGitHub {
    owner   = "inhabitedtype";
    repo    = pname;
    rev     = version;
    hash    = "sha256-MK8o+iPGANEhrrTc1Kz9LBilx2bDPQt7Pp5P2libucI" v: stable -
```

```
};

checkInputs = [ alcotest ppx_let ];
buildInputs = [ ocaml-syntax-shims ];
propagatedBuildInputs = [ bigstringaf result ];
doCheck = lib.versionAtLeast ocaml.version "4.05";

meta = {
  homepage = "https://github.com/inhabitedtype/angstrom";
  description = "OCaml parser combinators built for speed and memory ef-";
  license = lib.licenses.bsd3;
  maintainers = with lib.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 = "1.0.2";

  minimalOCamlVersion = "4.02";

  src = fetchurl {
    url = "https://github.com/flowtype/ocaml-${pname}/releases/download/v${version}/ocaml-wtf8-${version}.tar.gz";
    hash = "sha256-d5/3KUBAWRj8tntr4RkJ74KWW7wvn/B/m1nx0npnzyc=";
  };

  meta = with lib; {
    homepage = "https://github.com/flowtype/ocaml-wtf8";
    description = "WTF-8 is a superset of UTF-8 that allows unpaired surroundin";
    license = licenses.mit;
```

v: stable -

```
    maintainers = [ maintainers.eqyiel ];
};

}
```

Note about `minimalOCamlVersion`. A deprecated version of this argument was spelled `minimumOCamlVersion`; setting the old attribute wrongly modifies the derivation hash and is therefore inappropriate. As a technical dept, currently packaged libraries may still use the old spelling: maintainers are invited to fix this when updating packages. Massive renaming is strongly discouraged as it would be challenging to review, difficult to test, and will cause unnecessary rebuild.

The build will automatically fail if two distinct versions of the same library are added to `buildInputs` (which usually happens transitively because of `propagatedBuildInputs`). Set `dontDetectOcamlConflicts` to true to disable this behavior.

Octave

[Introduction](#)

[Structure](#)

[Packaging Octave Packages](#)

Introduction

Octave is a modular scientific programming language and environment. A majority of the packages supported by Octave from their [website](#) are packaged in nixpkgs.

Structure

All Octave add-on packages are available in two ways:

1. Under the top-level `Octave` attribute, `octave.pkgs`.
2. As a top-level attribute, `octavePackages`.

Packaging Octave Packages

Nixpkgs provides a function `buildOctavePackage`, a generic package builder function for any Octave package that complies with the Octave's current packaging format.

All Octave packages are defined in [pkgs/top-level/octave-packages.nix](#) rather than `pkgs/all-packages.nix`. Each package is defined in their own file in the [pkgs/development/octave-modules](#) directory. Octave packages are made available through `all-packages.nix` through both the attribute `octavePackages` and `octave.pkgs`. You can test building an Octave package as follows:

```
$ nix-build -A octavePackages.symbolic
```

To install it into your user profile, run this command from the root of the repository:

```
$ nix-env -f. -iA octavePackages.symbolic
```

You can build Octave with packages by using the `withPackages` passed-through function.

```
$ nix-shell -p 'octave.withPackages (ps: with ps; [ symbolic ])'
```

This will also work in a `shell.nix` file.

```
{ pkgs ? import <nixpkgs> { }: pkgs.mkShell { nativeBuildInputs = with pkgs; [ (octave.withPackages (opkgs: with opkgs; [ symbolic ])) ]; }
```

buildOctavePackage Steps

The `buildOctavePackage` does several things to make sure things work properly.

v: stable -

1. Sets the environment variable `OCTAVE_HISTFILE` to `/dev/null` during package compilation so that the commands run through the Octave interpreter directly are not logged.
2. Skips the configuration step, because the packages are stored as gzipped tarballs, which Octave itself handles directly.
3. Change the hierarchy of the tarball so that only a single directory is at the top-most level of the tarball.
4. Use Octave itself to run the `pkg build` command, which unzips the tarball, extracts the necessary files written in Octave, and compiles any code written in C++ or Fortran, and places the fully compiled artifact in `$out`.

`buildOctavePackage` is built on top of `stdenv` in a standard way, allowing most things to be customized.

Handling Dependencies

In Octave packages, there are four sets of dependencies that can be specified:

`nativeBuildInputs`

Just like other packages, `nativeBuildInputs` is intended for architecture-dependent build-time-only dependencies.

`buildInputs`

Like other packages, `buildInputs` is intended for architecture-independent build-time-only dependencies.

`propagatedBuildInputs`

Similar to other packages, `propagatedBuildInputs` is intended for packages that are required for both building and running of the package. See [Symbolic](#) for how this works and why it is needed.

`requiredOctavePackages`

This is a special dependency that ensures the specified Octave packages are dependent on others, and are made available simultaneously when loading them in Octave.

Installing Octave Packages

By default, the `buildOctavePackage` function does *not* install the requested package into Octave for use. The function will only build the requested package. This is due to Octave maintaining an text-based database about which packages are installed where. To this end, when all the requested packages have been built, the Octave package and all its add-on packages are put together into an environment, similar to Python.

1. First, all the Octave binaries are wrapped with the environment variable `OCTAVE_SITE_INITFILE` set to a file in `$out`, which is required for Octave to be able to find the non-standard package database location.
2. Because of the way `buildEnv` works, all tarballs that are present (which should be all Octave packages to install) should be removed.
3. The path down to the default install location of Octave packages is recreated so that Nix-operated Octave can install the packages.
4. Install the packages into the `$out` environment while writing package entries to the database file. This database file is unique for each different (according to Nix) environment invocation.
5. Rewrite the Octave-wide startup file to read from the list of packages installed in that particular environment.
6. Wrap any programs that are required by the Octave packages so that they work with all the paths defined within the environment.

Perl

[Running Perl programs on the shell](#)

[Packaging Perl programs](#)

Running Perl programs on the shell

When executing a Perl script, it is possible you get an error such as `./myscript.pl: bad command`

v: stable -

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
```

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`.

modules. Here is an example of the former:

```
ClassC3 = buildPerlPackage rec {  
  pname = "Class-C3";  
  version = "0.21";  
  src = fetchurl {  
    url = "mirror://cpan/authors/id/F/FL/FLORA/${pname}-${version}.tar.gz";  
    hash = "sha256-/5GE5xHT0uYGOQxroqj6LMU7CtKn2s6vMVoSXxL4iK4=";  
  };  
};
```

Note the use of `mirror://cpan/`, and the `pname` and `version` in the URL definition to ensure that the `pname` 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
```

To install it with `nix-env` instead: `nix-env -f. -iA 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 `-I``dir` 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.)

v: stable -

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 {
  pname = "BerkeleyDB";
  version = "0.36";

  src = fetchurl {
    url = "mirror://cpan/authors/id/P/PM/PMQS/${pname}-${version}.tar.gz"
    hash = "sha256-4Y+HGgGQqc0fdiKcFIyMrWBEccVNVAMDBWZlFTMorh8=";
  };

  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 {
  pname = "Class-C3-Componentised";
  version = "1.0004";
  src = fetchurl {
    url = "mirror://cpan/authors/id/A/AS/ASH/${pname}-${version}" v: stable -
```

```
    hash = "sha256-AS09rV/FzJYZ0BH572Fxm2ZrFLMZLFATJng1NuU4FHc=";  
};  
propagatedBuildInputs = [  
  ClassC3 ClassInspector TestException MROCompat  
];  
};
```

On Darwin, if a script has too many `-I` 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 = "12.50";  
  
  src = fetchurl {  
    url = "https://exiftool.org/${pname}-${version}.tar.gz";  
    hash = "sha256-v0hB/FwQMC8PPVdnjDvxRpU6jAZcC6GMQfc0AH4uwKg=";  
  };  
  
  nativeBuildInputs = lib.optional stdenv.isDarwin shortenPerlShebang;  
  postInstall = lib.optionalString 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.

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:

v: stable -

```
$ nix-env -f "<nixpkgs>" -iA nix-generate-from-cpan
```

Substitute `<nixpkgs>` by the path of a nixpkgs clone to use the latest version.

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 {
  pname = "XML-Simple";
  version = "2.22";
  src = fetchurl {
    url = "mirror://cpan/authors/id/G/GR/GRANTM/XML-Simple-2.22.tar.gz";
    hash = "sha256-uUU08i6pZErl1q2ghtxDAPoQW+BQogM0vU79KMGY60k=";
  };
  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.

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.

PHP

v: stable -

[User Guide](#)

User Guide

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., `php81` is PHP 8.1.

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 8.1 package, i.e. the unwrapped one, is available as `php81.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`.

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 extensi v: stable -

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 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";
}
```

Example setup for `phpfpm`

v: stable -

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;
};
```

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 ]')
```

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 }:
```

v: stable -

```
enabled ++ (with all; [ imagick redis ]))  
).packages.composer
```

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 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 [] ++ [  
                    ...  
                ];  
            });  
        };  
    };  
}
```

Building PHP projects

With [Composer](#), you can effectively build PHP projects by streamlining dependency management. As the de-facto standard dependency manager for PHP, Composer enables you to declare and manage the libraries your project relies on, ensuring a more organized and efficient development process.

Composer is not a package manager in the same sense as `Yum` or `Apt` are. Yes, it deals with “packages” or libraries, but it manages them on a per-project basis, installing them in a directory (e.g. `vendor`) inside your project. By default, it does not install anything globally. This idea is not new and Composer is strongly inspired by node’s `npm` and ruby’s `bundler`.

Currently, there is no other PHP tool that offers the same functionality as Composer. Consequently, incorporating a helper in Nix to facilitate building such applications is a logical choice.

In a Composer project, dependencies are defined in a `composer.json` file, while their spe v: stable -

versions are locked in a `composer.lock` file. Some Composer-based projects opt to include this `composer.lock` file in their source code, while others choose not to.

In Nix, there are multiple approaches to building a Composer-based project.

One such method is the `php.buildComposerProject` helper function, which serves as a wrapper around `mkDerivation`.

Using this function, you can build a PHP project that includes both a `composer.json` and `composer.lock` file. If the project specifies binaries using the `bin` attribute in `composer.json`, these binaries will be automatically linked and made accessible in the derivation. In this context, “binaries” refer to PHP scripts that are intended to be executable.

To use the helper effectively, add the `vendorHash` attribute, which enables the wrapper to handle the heavy lifting.

Internally, the helper operates in three stages:

1. It constructs a `composerRepository` attribute derivation by creating a composer repository on the filesystem containing dependencies specified in `composer.json`. This process uses the function `php.mkComposerRepository` which in turn uses the `php.composerHooks.composerRepositoryHook` hook. Internally this function uses a custom [Composer plugin](#) to generate the repository.
2. The resulting `composerRepository` derivation is then used by the `php.composerHooks.composerInstallHook` hook, which is responsible for creating the final `vendor` directory.
3. Any “binary” specified in the `composer.json` are linked and made accessible in the derivation.

As the autoloader optimization can be activated directly within the `composer.json` file, we do not enable any autoloader optimization flags.

To customize the PHP version, you can specify the `php` attribute. Similarly, if you wish to modify the Composer version, use the `composer` attribute. It is important to note that both attributes should be of the `derivation` type.

Here's an example of working code example using `php.buildComposerProject`:

v: stable -

```
{ php, fetchFromGitHub }:

php.buildComposerProject (finalAttrs: {
  pname = "php-app";
  version = "1.0.0";

  src = fetchFromGitHub {
    owner = "git-owner";
    repo = "git-repo";
    rev = finalAttrs.version;
    hash = "sha256-VcQRSss2dssfkJ+iUb5qT+FJ10GHifDzySigcmuVI+8=";
  };

  # PHP version containing the `ast` extension enabled
  php = php.buildEnv {
    extensions = ({ enabled, all }: enabled ++ (with all; [
      ast
    ]));
  };
}

# The composer vendor hash
vendorHash = "sha256-86s/F+/5cBAwBqZ2yaGRM5rTGLmou5//aLRK5SA0WiQ=";

# If the composer.lock file is missing from the repository, add it:
# composerLock = ./path/to/composer.lock;
})
```

In case the file `composer.lock` is missing from the repository, it is possible to specify it using the `composerLock` attribute.

The other method is to use all these methods and hooks individually. This has the advantage of building a PHP library within another derivation very easily when necessary.

Here's a working code example to build a PHP library using `mkDerivation` and separate functions and hooks:

```
{ stdenvNoCC, fetchFromGitHub, php }:

stdenvNoCC.mkDerivation (finalAttrs:
let
  src = fetchFromGitHub {
    owner = "git-owner";
    repo = "git-repo";
    rev = finalAttrs.version;
    hash = "sha256-VcQRSSs2dssfkJ+iUb5qT+FJ10GHiFDzySigcmuVI+8=";
  };
in {
  inherit src;
  pname = "php-app";
  version = "1.0.0";

  buildInputs = [ php ];

  nativeBuildInputs = [
    php.packages.composer
    # This hook will use the attribute `composerRepository`
    php.composerHooks.composerInstallHook
  ];
}

composerRepository = php.mkComposerRepository {
  inherit (finalAttrs) src;
  # Specifying a custom composer.lock since it is not present in the source
  composerLock = ./composer.lock;
  # The composer vendor hash
  vendorHash = "sha256-86s/F+/5cBAwBqZ2yaGRM5rTGLmou5//aLRK5SA0WiQ=";
};
})
```

pkg-config

[Writing packages providing pkg-config modules](#)

v: stable -

Accessing packages via pkg-config module name

pkg-config is a unified interface for declaring and querying built C/C++ libraries.

Nixpkgs provides a couple of facilities for working with this tool.

Writing packages providing pkg-config modules

Packages should set `meta.pkgConfigModules` with the list of package config modules they provide. They should also use `testers.testMetaPkgConfig` to check that the final built package matches that list. Additionally, the [validatePkgConfig setup hook](#), will do extra checks on to-be-installed `pkg-config` modules.

A good example of all these things is zlib:

```
{ pkg-config, testers, ... }:

stdenv.mkDerivation (finalAttrs: {
  ...
  nativeBuildInputs = [ pkg-config validatePkgConfig ];
  passthru.tests.pkg-config = testers.testMetaPkgConfig finalAttrs.finalPath;
  meta = {
    ...
    pkgConfigModules = [ "zlib" ];
  };
})
```

Accessing packages via pkg-config module name

Within Nixpkgs

v: stable -

A [setup hook](#) is bundled in the `pkg-config` package to bring a derivation's declared build inputs into the environment. This will populate environment variables like `PKG_CONFIG_PATH`, `PKG_CONFIG_PATH_FOR_BUILD`, and `PKG_CONFIG_PATH_HOST` based on:

- how `pkg-config` itself is depended upon
- how other dependencies are depended upon

For more details see the section on [specifying dependencies in general](#).

Normal `pkg-config` commands to look up dependencies by name will then work with those environment variables defined by the hook.

Externally

The `defaultPkgConfigPackages` package set is a set of aliases, named after the modules they provide. This is meant to be used by language-to-nix integrations. Hand-written packages should use the normal `Nixpkgs` attribute name instead.

Python

[Reference](#)

[User Guide](#)

[FAQ](#)

[Contributing](#)

[Package set maintenance](#)

[CPython Update Schedule](#)

Reference

Interpreters

v: stable -

Package	Aliases	Interpreter
python27	python2, python	CPython 2.7
python38		CPython 3.8
python39		CPython 3.9
python310		CPython 3.10
python311	python3	CPython 3.11
python312		CPython 3.12
python313		CPython 3.13
pypy27	pypy2, pypy	PyPy2.7
pypy39	pypy3	PyPy 3.9

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.

Missing `tkinter` module standard library

To reduce closure size the `Tkinter/tkinter` is available as a separate package, `pythonPackages.tkinter`.

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}`.

v: stable -

- **buildEnv**. Function to build python interpreter environments with extra packages bundled together. See [the section called “`python.buildEnv` function”](#) for usage and documentation.
- **withPackages**. Simpler interface to `buildEnv`. See [the section called “`python.withPackages` function”](#) for usage and documentation.
- **sitePackages**. Alias for `lib/${libPrefix}/site-packages`.
- **executable**. Name of the interpreter executable, e.g. `python3.10`.
- **pkgs**. Set of Python packages for that specific interpreter. The package set can be modified by overriding the interpreter and passing `packageOverrides`.

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.python3Packages`
- `pkgs.python38Packages`
- `pkgs.python39Packages`
- `pkgs.python310Packages`

v: stable -

- pkgs.python311Packages
- pkgs.python312Packages
- pkgs.python313Packages
- pkgs.pypyPackages

and the aliases

- pkgs.python2Packages pointing to pkgs.python27Packages
- pkgs.python3Packages pointing to pkgs.python311Packages
- pkgs.pythonPackages pointing to pkgs.python2Packages

buildPythonPackage function

The `buildPythonPackage` function is implemented in `pkgs/development/interpreters/python/mk-python-derivation.nix` using setup hooks.

The following is an example:

```
{ lib
, buildPythonPackage
, fetchPypi

# build-system
, setuptools-scm

# dependencies
, attrs
, pluggy
, py
, setuptools
, six

# tests
```

v: stable -

```
, hypothesis
}:

buildPythonPackage rec {
  pname = "pytest";
  version = "3.3.1";
  pyproject = true;

  src = fetchPypi {
    inherit pname version;
    hash = "sha256-z4Q23FnYaVNG/NOrKW3kZCXsqwDWQJb0vnn7Ueyy65M=";
  };

  postPatch = ''
    # don't test bash builtins
    rm testing/test_argcomplete.py
  '';

  nativeBuildInputs = [
    setuptools-scm
  ];

  propagatedBuildInputs = [
    attrs
    py
    setuptools
    six
    pluggy
  ];

  nativeCheckInputs = [
    hypothesis
  ];

  meta = with lib; {
    changelog = "https://github.com/pytest-dev/pytest/releases/tag/${version}";
    description = "Framework for writing tests";
    homepage = "https://github.com/pytest-dev/pytest";
  };
}
```

v: stable -

```
    license = licenses.mit;
    maintainers = with maintainers; [ domenkozar lovek323 madjar lsix ];
};

}
```

The `buildPythonPackage` mainly does four things:

- In the `buildPhase`, it calls `${python.pythonOnBuildForHost.interpreter} setup.py bdist_wheel` to build a wheel binary zipfile.
- In the `installPhase`, it installs the wheel file using `pip install *.whl`.
- In the `postFixup` phase, the `wrapPythonPrograms` bash function is called to wrap all programs in the `$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 run.

By default tests are run because `doCheck = true`. Test dependencies, like e.g. the test runner, should be added to `nativeCheckInputs`.

By default `meta.platforms` is set to the same value as the interpreter unless overridden otherwise.

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 `PYTHONNOUSER SITE` environment variable in wrapped programs.
- `pyproject`: Whether the pyproject format should be used. When set to `true`, `pypabu` v: stable - |

be used, and you can add the required build dependencies from `build-system.requires` to `nativeBuildInputs`. Note that the `pyproject` format falls back to using `setuptools`, so you can use `pyproject = true` even if the package only has a `setup.py`. When set to `false`, you can use the existing [hooks](#setup-hooks) or provide your own logic to build the package. This can be useful for packages that don't support the `pyproject` format. When unset, the legacy `setuptools` hooks are used for backwards compatibility.

- `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 `${name}` parameter. In case of libraries, this defaults to `"python3.8-"` for Python 3.8, etc., and in case of applications to `" "`.
- `pipInstallFlags ? []`: A list of strings. Arguments to be passed to `pip install`. To pass options to `python setup.py install`, use `--install-option`. E.g., `pipInstallFlags= [--install-option='--cpp_implementation']`.
- `pipBuildFlags ? []`: A list of strings. Arguments to be passed to `pip wheel`.
- `pypaBuildFlags ? []`: A list of strings. Arguments to be passed to `python -m build --wheel`.
- `pythonPath ? []`: List of packages to be added into `$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.
- `nativeCheckInputs` ? []: 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.

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 =  fetchPypi {
          pname = "pandas";
          inherit version;
          hash = "sha256-JQn+rtpy/0A2deLszSKEuxyttqBzcAil50H+JDHUdCE=";
        };
      });
    });
  v: stable -
```

```
};

in pkgs.python3.override {inherit packageOverrides; self = python;};

in python.withPackages(ps: [ ps.blaze ]).env
```

The next example shows a non trivial overriding of the `blas` implementation to be used through out all of the Python package set:

```
python3MyBlas = pkgs.python3.override {
  packageOverrides = self: super: {
    # We need toPythonModule for the package set to evaluate this
    blas = super.toPythonModule(super.pkgs.blas.override {
      blasProvider = super.pkgs.mkl;
    });
    lapack = super.toPythonModule(super.pkgs.lapack.override {
      lapackProvider = super.pkgs.mkl;
    });
  };
};
```

This is particularly useful for numpy and scipy users who want to gain speed with other blas implementations. Note that using `scipy = super.scipy.override { blas = super.pkgs.mkl; };` will likely result in compilation issues, because scipy dependencies need to use the same blas implementation as well.

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 `python3` or `python3Packages` (possibly specifying an interpreter version), like this:

```
{ lib
, python3Packages
, fetchPypi
}:

python3Packages.buildPythonApplication rec {
  pname = "luigi";
  version = "2.7.9";
  pyproject = true;

  src = fetchPypi {
    inherit pname version;
    hash = "sha256-Pe229rT0aHwA98s+nTHQMEFKZPo/yw6sot8MivFDvAw=";
  };

  nativeBuildInputs = [
    python3Packages.setuptools
    python3Packages.wheel
  ];

  propagatedBuildInputs = [
    python3Packages.tornado
    python3Packages.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 `python3Packages`.

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 python3Packages; toPythonApplication youtube-dl;
```

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!

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> {};

python3.buildEnv.override {
  extraLibs = [ python3Packages.pyramid ];
  ignoreCollisions = true;
}
```

and running `nix-build` will create

```
/nix/store/cf1xhjwzmdki7fasgr4kz6di72yklc15-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.

- **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 `PYTHONNOUSER SITE` environment variable in wrapped binaries in the environment.

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` 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> {};
```

v: stable -

```
(python3.withPackages (ps: with ps; [  
    numpy  
    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`.

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.
- `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`.
- `pypaBuildHook` to build a wheel using [`pypa/build`](#) and PEP 517/518. 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.
- `pythonRelaxDepsHook` will relax Python dependencies restrictions for the package. S v: stable -

usage.

- `pythonRemoveBinBytecode` to remove bytecode from the `/bin` folder.
- `setuptoolsBuildHook` to build a wheel using `setuptools`.
- `setuptoolsCheckHook` to run tests with `python setup.py test`.
- `sphinxHook` to build documentation and manpages using Sphinx.
- `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 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`.
- `unittestCheckHook` will run tests with `python -m unittest discover`. See [example usage](#).

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> {};
      python3Packages.buildPythonPackage {
    name = "myproject";
    buildInputs = with python3Packages; [ pyramid ];
    src = ./.;
}
```

Running `nix-shell` with no arguments should give you the environment in which the pac

v: stable -

be built with `nix-build`.

Shortcut to setup environments with C headers/libraries and Python packages:

```
nix-shell -p python3Packages.pyramid zlib libjpeg git
```

Note

There is a boolean value `lib.inNixShell` set to `true` if `nix-shell` is invoked.

User Guide

Using Python

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 3.11. The attribute `python` refers to CPython 2.7 for backwards-compatibility. It is also possible to refer to specific versions, e.g. `python311` refers to CPython 3.11, 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 `python3.pkgs.toolz`, and the CPython 3.11 version is `python311.pkgs.toolz`. The main package set contains aliases to these package sets, e.g. `pythonPackages` refers to `python.pkgs` and `python311Packages` to `python311.pkgs`.

v: stable -

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 3.11; 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.

Philosophically, 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.

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 3.11 session with `numpy` and `toolz` available, run:

v: stable -

```
$ nix-shell -p 'python311.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:

```
[nix-shell:~/src/nixpkgs]$ python3
Python 3.11.3 (main, Apr  4 2023, 22:36:41) [GCC 12.2.0] 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:

```
>>> 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 "python311.withPackages (ps: with ps; [ numpy toolz requests ])
this derivation will be built:
/nix/store/r19yf5qgfiakqlhkgjahbg3zg79549n4-python3-3.11.2-envdrv
building '/nix/store/r19yf5qgfiakqlhkgjahbg3zg79549n4-python3-3.11.2-env.drv'
created 273 symlinks in user environment
Python 3.11.2 (main, Feb  7 2023, 13:52:42) [GCC 12.2.0] 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 t v: stable -

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.

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 'python311.withPackages (ps: with ps; [ numpy ]) --run 'python3 dot.py'
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 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/e51209796c4262fb8908e3d6d72302fe4e96f5f
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 3.10, numpy, and system dependencies a year from now as it does today, because it will always use exactly git commit `e51209796c4262fb8908e3d6d72302fe4e96f5f` of Nixpkgs for all of the package v^{v: stable} -

This is also a great way to ensure the script executes identically on different servers.

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-shell section](#) of the Nix manual, `nix-shell` can also load an expression from a `.nix` file. Say we want to have Python 3.11, `numpy` and `toolz`, like before, in an environment. We can add a `shell.nix` file describing our dependencies:

```
with import <nixpkgs> {};
(python311.withPackages (ps: with ps; [
    numpy
    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 3.11 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> {};
```

v: stable -

```
let
  pythonEnv = python311.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.

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 {
```

v: stable -

```
name = "myEnv";
paths = [
    # A Python 3 interpreter with some packages
    (self.python3.withPackages (
        ps: with ps; [
            pyflakes
            pytest
            black
        ]
    ))
]

# 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.

Environment defined in /etc/nixos/configuration.nix

For the sake of completeness, here's how to install the environment system-wide on NixOS.

```
{ # ...
```

v: stable -

```
environment.systemPackages = with pkgs; [
  (python310.withPackages(ps: with ps; [ numpy toolz ]))
];
}
```

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.

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
, setuptools
, wheel
}:

buildPythonPackage rec {
  pname = "toolz";
  version = "0.10.0";
  pyproject = true;

  src = fetchPypi {
    inherit pname version;
    hash = "sha256-CP3V73yWSArRHBLUct4hrNMjWZlvaauLkpm1QP66RWA=";
  };

  nativeBuildInputs = [
    setuptools
    wheel
  ];

  # has no tests
  doCheck = false;

  pythonImportsCheck = [
    "toolz.itoold"
    "toolz.functor"
    "toolz.dicttool"
  ];

  meta = with lib; {
    changelog = "https://github.com/pytoolz/toolz/releases/tag/${version}";
    homepage = "https://github.com/pytoolz/toolz";
    description = "List processing tools and functional utilities";
    license = licenses.bsd3;
    maintainers = with maintainers; [ fridh ];
  };
}
```

v: stable -

```
};  
}
```

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. Since there are no tests, we rely on [pythonImportsCheck](#) to test whether the package can be imported. Furthermore, we specify some 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.

`python311.pkgs.toolz` refers to the `toolz` derivation corresponding to the CPython 3.11 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 = python311.pkgs.buildPythonPackage rec {  
      pname = "toolz";  
      version = "0.10.0";  
      pyproject = true;  
  
      src = fetchPypi {  
        inherit pname version;  
        hash = "sha256-CP3V73yWSArRHBLUct4hrNMjWZlvaauLkpm1QP66RWA=";  
      };  
    };  
  )
```

v: stable -

```
nativeBuildInputs = [
  python311.pkgs.setuptools
  python311.pkgs.wheel
];

# has no tests
doCheck = false;

meta = {
  homepage = "https://github.com/pytoolz/toolz/";
  description = "List processing tools and functional utilities";
  # [...]
};

in python311.withPackages (ps: with ps; [
  numpy
  my_toolz
])
).env
```

Executing `nix-shell` will result in an environment in which you can use Python 3.11 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.

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 `meta`.

[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 [nativeCheckInputs](#).

The following example shows which arguments are given to [buildPythonPackage](#) in order to build [datashape](#).

```
{ lib
, buildPythonPackage
, fetchPypi

# build dependencies
, setuptools, wheel

# dependencies
, numpy, multipledispatch, python-dateutil

# tests
, pytest
}:

buildPythonPackage rec {
  pname = "datashape";
  version = "0.4.7";
  pyproject = true;

  src = fetchPypi {
    inherit pname version;
    hash = "sha256-FLLvdm1MllKrgTGC6Gb0k0deZeVYvtCCLji/B7uhong=";
  };

  nativeBuildInputs = [
    setuptools
    wheel
  ];
}
```

v: stable -

```
propagatedBuildInputs = [
    multipledispatch
    numpy
    python-dateutil
];

nativeCheckInputs = [
    pytest
];

meta = with lib; {
    changelog = "https://github.com/blaze/datasource/releases/tag/\${version}";
    homepage = "https://github.com/ContinuumIO/datasource";
    description = "A data description language";
    license = licenses.bsd2;
    maintainers = with maintainers; [ fridh ];
};

}
```

We can see several runtime dependencies, `numpy`, `multipledispatch`, and `python-dateutil`. Furthermore, we have `nativeCheckInputs` with `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
, buildPythonPackage
, fetchPypi
, setuptools
, wheel
, libxml2
, libxslt
}:

buildPythonPackage rec {
  pname = "lxml";
  version = "3.4.4";
  pyproject = true;

  src = fetchPypi {
    inherit pname version;
    hash = "sha256-s9NiusRxFydHzaNRMjjxFcvWxfi45jGb9ql6eJJyQJk=";
  };

  nativeBuildInputs = [
    setuptools
    wheel
  ];

  buildInputs = [
    libxml2
    libxslt
  ];

  meta = with lib; {
    changelog = "https://github.com/lxml/lxml/releases/tag/lxml-${version}";
    description = "Pythonic binding for the libxml2 and libxslt libraries";
    homepage = "https://lxml.de";
    license = licenses.bsd3;
    maintainers = with maintainers; [ sjourdois ];
  };
}
```

v: stable -

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
, buildPythonPackage
, fetchPypi

# build dependencies
, setuptools
, wheel

# dependencies
, fftw
, fftwFloat
, fftwLongDouble
, numpy
, scipy
}:

buildPythonPackage rec {
  pname = "pyFFTW";
  version = "0.9.2";
  pyproject = true;

  src = fetchPypi {
    inherit pname version;
    hash = "sha256-9ru2r6kwhUCaskiFoaPNuJCfCVoUL01J40byvRt4kHQ=";
  };

  nativeBuildInputs = [
```

v: stable -

```
setupTools
wheel
];

buildInputs = [
  fftw
  fftwFloat
  fftwLongDouble
];

propagatedBuildInputs = [
  numpy
  scipy
];

preConfigure = ''
  export LDFLAGS="-L${fftw.dev}/lib -L${fftwFloat.out}/lib -L${fftwLongDoub
  export CFLAGS="-I${fftw.dev}/include -I${fftwFloat.dev}/include -I${fftwL
';

# Tests cannot import pyfftw. pyfftw works fine though.
doCheck = false;

meta = with lib; {
  changelog = "https://github.com/pyFFTW/pyFFTW/releases/tag/v\${version}";
  description = "A pythonic wrapper around FFTW, the FFT library, presented as a Python module";
  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.

Testing Python Packages

It is highly encouraged to have testing as part of the package build. This helps to avoid situations where tests fail due to missing dependencies or configuration issues.

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.

Using pytest

Pytest is the most common test runner for python repositories. A trivial test run would be:

```
nativeCheckInputs = [ pytest ];
checkPhase = ''
  runHook preCheck

  pytest

  runHook postCheck
';
```

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:

v: stable -

```
nativeCheckInputs = [ pytest ];
# avoid tests which need additional data or touch network
checkPhase = ''
  runHook preCheck

  pytest tests/ --ignore=tests/integration -k 'not download and not update'
  runHook postCheck
';
```

--ignore will tell pytest to ignore that file or directory from being collected as part of a test run. This is useful if 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 allowed, 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`.

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 analogous `pytestCheckHook` usage would be:

```
nativeCheckInputs = [
  pytestCheckHook
];
```

v: stable -

```
# requires additional data
pytestFlagsArray = [
    "tests/"
    "--ignore=tests/integration"
];

disabledTests = [
    # touches network
    "download"
    "update"
];

disabledTestPaths = [
    "tests/test_failing.py"
];
```

This is especially useful when tests need to be conditionally disabled, 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.

Using `pythonImportsCheck`

Although unit tests are highly preferred to validate correctness of a package, not all packag

v: stable -

suites that can be run 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 its 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`).

Using `pythonRelaxDepsHook`

It is common for upstream to specify a range of versions for its package dependencies. This makes sense, since it ensures that the package will be built with a subset of packages that is well tested. However, this commonly causes issues when packaging in Nixpkgs, because the dependencies that this package may need are too new or old for the package to build correctly. We also cannot package multiple versions of the same package since this may cause conflicts in `PYTHONPATH`.

One way to side step this issue is to relax the dependencies. This can be done by either removing the package version range or by removing the package declaration entirely. This can be done using the `pythonRelaxDepsHook` hook. For example, given the following `requirements.txt` file:

```
pkg1<1.0  
pkg2  
pkg3>=1.0,<=2.0
```

v: stable -

we can do:

```
nativeBuildInputs = [
  pythonRelaxDepsHook
];
pythonRelaxDeps = [
  "pkg1"
  "pkg3"
];
pythonRemoveDeps = [
  "pkg2"
];
```

which would result in the following `requirements.txt` file:

```
pkg1
pkg3
```

Another option is to pass `true`, that will relax/remove all dependencies, for example:

```
nativeBuildInputs = [ pythonRelaxDepsHook ];
pythonRelaxDeps = true;
```

which would result in the following `requirements.txt` file:

```
pkg1
pkg2
pkg3
```

In general you should always use `pythonRelaxDeps`, because `pythonRemoveDeps` will convert build errors into runtime errors. However `pythonRemoveDeps` may still be useful in exceptional cases, and also to remove dependencies wrongly declared by upstream (for example, declaring `black` as a runtime dependency instead of a dev dependency).

Keep in mind that while the examples above are done with `requirements.txt`,

v: stable -

`pythonRelaxDepsHook` works by modifying the resulting wheel file, so it should work with any of the [existing hooks](#).

Using `unittestCheckHook`

`unittestCheckHook` is a hook which will substitute the `setuptools test` command for a [checkPhase](#) which runs `python -m unittest discover`:

```
nativeCheckInputs = [
    unittestCheckHook
];

unittestFlagsArray = [
    "-s" "tests" "-v"
];
```

Using `sphinxHook`

The `sphinxHook` is a helpful tool to build documentation and manpages using the popular Sphinx documentation generator. It is setup to automatically find common documentation source paths and render them using the default `html` style.

```
outputs = [
    "out"
    "doc"
];

nativeBuildInputs = [
    sphinxHook
];
```

The hook will automatically build and install the artifact into the `doc` output, if it exists. It also provides an automatic diversion for the artifacts of the `man` builder into the `man` target.

```
outputs = [
```

v: stable -

```
"out"
"doc"
"man"
];

# Use multiple builders
sphinxBuilders = [
  "singlehtml"
  "man"
];

```

Overwrite `sphinxRoot` when the hook is unable to find your documentation source root.

```
# Configure sphinxRoot for uncommon paths
sphinxRoot = "weird/docs/path";
```

The hook is also available to packages outside the python ecosystem by referencing it using `sphinxHook` from top-level.

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 a 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 3.11 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](#).

v: stable -

```
with import <nixpkgs> {};
with python311Packages;

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.

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`.

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
, fetchPypi
, ... }:
  let
    toolz = buildPythonPackage rec {
      ... };
    in
    { toolz
    , ... };
```

v: stable -

```
, setuptools
, wheel
}:

buildPythonPackage rec {
  pname = "toolz";
  version = "0.10.0";
  pyproject = true;

  src = fetchPypi {
    inherit pname version;
    hash = "sha256-CP3V73yWSArRHBLUct4hrNMjWZlvaauLkpm1QP66RWA=";
  };

  nativeBuildInputs = [
    setuptools
    wheel
  ];

  meta = with lib; {
    changelog = "https://github.com/pytoolz/toolz/releases/tag/\${version}";
    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 = python310
  }
  Packages.buildPythonPackage;
```

v: stable -

```
};

in python310.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 `python3Packages`, and in this case the `python3` interpreter is automatically used.

FAQ

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.

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.python310.override {
    inherit packageOverrides;
  };
);
```

v: stable -

```
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.python310.override {
    inherit packageOverrides;
  }).withPackages (ps: [
    ps.blaze
  ])
).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: {
  python310 = let
    packageOverrides = python-self: python-super: {
```

v: stable -

```
    numpy = python-super.numpy_1_18;
};

in super.python310.override {inherit packageOverrides;};
} ); };

in newpkgs.inkscape
```

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"
```

install_data / data_files problems

v: stable -

If you get the following error:

```
could not create '/nix/store/6l1bvljpy8gazlsw2aw9skwwp4pmvyxw-python-2.7.  
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.pythonOnBuildForHost.interpreter} setup.py install_data --instal  
 sed -i '/ = data\_files/d' setup.py
```

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.

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. v: stable -

Create this `default.nix` file, together with a `requirements.txt` and 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 boot
    # the environment.
    pythonPackages.python

    # This executes 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
    # the following packages to be installed locally:
    taglib
    openssl
    git
    libxml2
    libxslt
    libzip
    zlib
  ];
}

# Run this command, only after creating the virtual environment
postVenvCreation = ''
```

v: stable -

```
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
      virtualenv --python=python ${venvDir}
    fi
  '';
}
```

```
else
    echo "Creating new venv environment in path: '${venvDir}'"
    # Note that the module venv was only introduced in python 3, so for
    # 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}/${pythonPackages.python.sitePackages}:/$I

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.

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: {
  python3 = super.python3.override {
    packageOverrides = python-self: python-super: {
      twisted = python-super.twisted.overridePythonAttrs (oldAttrs: {
        src = super.fetchPypi {
          pname = "Twisted";
          version = "19.10.0";
          hash = "sha256-c5S6fycq5yKnTz2Wnc9Zm8TvCTvDkg0HSKSQ8XJKUV0=";
          extension = "tar.bz2";
        };
      });
    };
  };
};
```

v: stable -

```
    });
};

};

};

};
```

`python3Packages.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 = python3Packages.override {
  overrides = self: super: {
    twisted = ...;
  };
}
```

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";
          hash = "sha256-c5S6fycq5yKnTz2Wnc9Zm8TvCTvDkg0HSKSQ8XJKUV0=";
          extension = "tar.bz2";
        };
      });
    };
  };
}
```

v: stable -

How to override a Python package for all Python versions using extensions?

The following overlay overrides the call to [buildPythonPackage](#) for the `foo` package for all interpreters by appending a Python extension to the `pythonPackagesExtensions` list of extensions.

```
final: prev: {
    pythonPackagesExtensions = prev.pythonPackagesExtensions ++ [
        (
            python-final: python-prev: {
                foo = python-prev.foo.overridePythonAttrs (oldAttrs: {
                    ...
                });
            }
        )
    ];
}
```

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](#)".

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 [nativeCheckInputs](#)

How to enable interpreter optimizations?

The Python interpreters are by default not built with optimizations enabled, because the builds are in that case not reproducible. To enable optimizations, override the interpreter of interest, e.g. `v: stable -`

```
let
  pkgs = import ./. {};
  mypython = pkgs.python3.override {
    enableOptimizations = true;
    reproducibleBuild = false;
    self = mypython;
  };
in mypython
```

How to add optional dependencies?

Some packages define optional dependencies for additional features. With `setuptools` this is called `extras_require` and `flit` calls it `extras-require`, while PEP 621 calls these `optional-dependencies`. A method for supporting this is by declaring the extras of a package in its `passthru`, e.g. in case of the package `dask`

```
passthru.optional-dependencies = {
  complete = [ distributed ];
};
```

and letting the package requiring the extra add the list to its dependencies

```
propagatedBuildInputs = [
  ...
] ++ dask.optional-dependencies.complete;
```

Note this method is preferred over adding parameters to builders, as that can result in packages depending on different variants and thereby causing collisions.

How to contribute a Python package to nixpkgs?

Packages inside nixpkgs must use the [buildPythonPackage](#) or [buildPythonApplication](#) function directly, because we can only provide security support for non-vendored dependencies.

We recommend [nix-init](#) for creating new python packages within nixpkgs, as it already prefers v: stable -

source, parses dependencies for common formats and prefills most things in `meta`.

Are Python interpreters built deterministically?

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`.

How to provide automatic tests to Python packages?

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.

Common issues

- Non-working tests can often be deselected. By default [`buildPythonPackage`](#) runs `python setup.py test`. which is deprecated. Most Python modules however do follow the standard test protocol where the `pytest` runner can be used instead. `pytest` supports the `-k` and `--ignore` parameters to ignore test methods or classes as well as whole files. For `pytestCheckHook` these are conveniently exposed as `disabledTests` and `disabledTestPaths` respectively.

```
buildPythonPackage {  
  # ...  
  nativeCheckInputs = [  
    pytestCheckHook  
  ];  
  
  disabledTests = [  
    "function_name"  
  ];
```

v: stable -

```
    "other_function"
];

disabledTestPaths = [
  "this/file.py"
];
}
```

- 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)`

Contributing

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 `python311Packages.numpy: 1.11 -> 1.12`. It is highly recommended to specify the current default version to enable automatic build by ofborg.
- Attribute names in `python-packages.nix` as well as `pnames` 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_Ba v: stable -`)

necessary, `pname` has to be given a different value within `fetchPypi`.

- Packages from sources such as GitHub and GitLab that do not exist on PyPI should not use a name that is already used on PyPI. When possible, they should use the package repository name prefixed with the owner (e.g. organization) name and using a - as delimiter.
- Attribute names in `python-packages.nix` should be sorted alphanumerically to avoid merge conflicts and ease locating attributes.

Package set maintenance

The whole Python package set has a lot of packages that do not see regular updates, because they either are a very fragile component in the Python ecosystem, like for example the `hypothesis` package, or packages that have no maintainer, so maintenance falls back to the package set maintainers.

Updating packages in bulk

There is a tool to update a lot of python libraries in bulk, it exists at `maintainers/scripts/update-python-libraries` with this repository.

It can quickly update minor or major versions for all packages selected and create update commits, and supports the `fetchPypi`, `fetchurl` and `fetchFromGitHub` fetchers. When updating lots of packages that are hosted on GitHub, exporting a `GITHUB_API_TOKEN` is highly recommended.

Updating packages in bulk leads to lots of breakages, which is why a stabilization period on the `python-unstable` branch is required.

If a package is fragile and often breaks during these bulk updates, it may be reasonable to set `passthru.skipBulkUpdate = true` in the derivation. This decision should not be made on a whim and should always be supported by a qualifying comment.

Once the branch is sufficiently stable it should normally be merged into the `staging` branch.

An exemplary call to update all python libraries between minor versions would be:

```
$ maintainers/scripts/update-python-libraries --target minor -- v:stable - l
```

C_Python Update Schedule

With [PEP 602](#), CPython now follows a yearly release cadence. In nixpkgs, all supported interpreters are made available, but only the most recent two interpreters package sets are built; this is a compromise between being the latest interpreter, and what the majority of the Python packages support.

New CPython interpreters are released in October. Generally, it takes some time for the majority of active Python projects to support the latest stable interpreter. To help ease the migration for Nixpkgs users between Python interpreters the schedule below will be used:

When	Event
After YY.11 Release	Bump CPython package set window. The latest and previous latest stable should now be built.
After YY.05 Release	Bump default CPython interpreter to latest stable.

In practice, this means that the Python community will have had a stable interpreter for ~2 months before attempting to update the package set. And this will allow for ~7 months for Python applications to support the latest interpreter.

Qt

[Nix expression for a Qt package \(default.nix\)](#)

[Locating runtime dependencies](#)

Writing Nix expressions for Qt libraries and applications is largely similar as for other C++ software. This section assumes some knowledge of the latter.

The major caveat with Qt applications is that Qt uses a plugin system to load additional modules at runtime, from a list of well-known locations. In Nixpkgs, we patch QtCore to instead use an environment variable, and wrap Qt applications to set it to the right paths. This effectively makes the runtime dependencies pure and explicit at build-time, at the cost of introducing an extra indirection.

v: stable -

Nix expression for a Qt package (default.nix)

```
{ stdenv, lib, qtbase, wrapQtAppsHook }:

stdenv.mkDerivation {
  pname = "myapp";
  version = "1.0";

  buildInputs = [ qtbase ];
  nativeBuildInputs = [ wrapQtAppsHook ];
}
```

It is important to 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.

Additionally all Qt packages must include `wrapQtAppsHook` in `nativeBuildInputs`, or you must explicitly set `dontWrapQtApps`.

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 }:
```

v: stable -

```
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.

R

[Installation](#)

[RStudio](#)

[Updating the package set](#)

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;
```

v: stable -

```
buildInputs = with rPackages; [
  R
  ggplot2
  knitr
];
};

}
```

and then run `nix-shell .` to be dropped into a shell with those packages available.

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:

v: stable -

```
{ 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.

Updating the package set

There is a script and associated environment for regenerating the package sets and synchronising the `rPackages` tree to the current CRAN and matching BIOC release. These scripts are found in the `pkgs/development/r-modules` directory and executed as follows:

```
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.nix  
mv bioc-annotation-packages.nix.new bioc-annotation-packages.nix
```

```
Rscript generate-r-packages.R bioc-experiment > bioc-experiment-packages.nix  
mv bioc-experiment-packages.nix.new bioc-experiment-packages.nix
```

`generate-r-packages.R <repo>` reads `<repo>-packages.nix`, therefore the renaming.

Some packages require overrides to specify external dependencies or other patches and special requirements. These overrides are specified in the `pkgs/development/r-modules/overrides` directory. The `overrides` directory contains files for each package that have been modified from the upstream version. These files typically contain patches, configuration changes, or other modifications specific to the NixOS build system.

file. As the `*-packages.nix` contents are automatically generated it should not be edited and broken builds should be addressed using overrides.

Ruby

[Using Ruby](#)

[Developing with Ruby](#)

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_7.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.

Temporary Ruby environment with `nix-shell`

Rather than having a single Ruby environment shared by all Ruby development projects on

v: stable -

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.

Load Ruby environment from .nix expression

As explained [in the nix-shell section](#) of 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.

v: stable -

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
```

Or immediately require `nokogiri` in `pry`:

```
$ nix-shell -p "ruby.withPackages (ps: with ps; [ nokogiri pry ])" --run
```

Or run a script using this environment:

```
$ nix-shell -p "ruby.withPackages (ps: with ps; [ nokogiri pry ])" --run
```

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-client ])"

require 'nokogiri'
require 'rest-client'

body = RestClient.get('http://example.com').body
puts Nokogiri::HTML(body).at('h1').text
```

Developing with Ruby

Using an existing Gemfile

v: stable -

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 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 yo v: stable -

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) ]; }
```

Sometimes a Gemfile references other files. Such as `.ruby-version` or vendored gems. When copying the Gemfile to the nix store we need to copy those files alongside. This can be done using `extraConfigPaths`. For example:

```
gems = bundlerEnv {
  name = "gems-for-some-project";
  gemdir = ./;
  extraConfigPaths = [ "${./}/.ruby-version" ];
};
```

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> { } }:
```

v: stable -

```
let
  myRuby = pkgs.ruby.override {
    defaultGemConfig = pkgs.defaultGemConfig // {
      pg = attrs: {
        buildFlags =
        [ "--with-pg-config=${pkgs."postgresql_${pg_version}"/bin/pg_config}" ];
      };
    };
  };
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_config}" ];
      };
    };
  };
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: {
```

v: stable -

```
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_version"'
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.

Platform-specific gems

Right now, bundix has some issues with pre-built, platform-specific gems: [bundix PR #68](#). Until this is solved, you can tell bundler to not use platform-specific gems and instead build them from source each time:

- globally (will be set in `~/.config/.bundle/config`):

```
$ bundle config set force_ruby_platform true
```

- locally (will be set in `<project-root>/ .bundle/config`):

v: stable -

```
$ bundle config set --local force_ruby_platform true
```

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
```

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 running the `bundle lock` and `bundix` steps again.

v: stable -

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.

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" ];

  nativeBuildInputs = [ makeWrapper ];

  postBuild = ''
```

v: stable -

```
wrapProgram $out/bin/r10k --prefix PATH : ${lib.makeBinPath [ git gnu
..;
}
```

Rust

[buildRustPackage](#): Compiling Rust applications with Cargo

[buildRustCrate](#): Compiling Rust crates using Nix instead of Cargo

[Using community maintained Rust toolchains](#)

[Using git bisect](#) on the Rust compiler

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 [community maintained Rust toolchains](#).

buildRustPackage: Compiling Rust applications with Cargo

Rust applications are packaged by using the `buildRustPackage` helper from `rustPlatform`:

```
{ lib, fetchFromGitHub, rustPlatform }:

rustPlatform.buildRustPackage rec {
  pname = "ripgrep";
```

v: stable -

```
version = "12.1.1";

src = fetchFromGitHub {
  owner = "BurntSushi";
  repo = pname;
  rev = version;
  hash = "sha256-+s5RBC3XSgb8omTbUNLywZnP6jSxZBKSS1BmX0jRF8M=";
};

cargoHash = "sha256-jtBw4ahSl88L0iuCXxQgZVm1EcboWRJMNTjxLVTTzts=";

meta = with lib; {
  description = "A fast line-oriented regex search tool, similar to ag and ack";
  homepage = "https://github.com/BurntSushi/ripgrep";
  license = licenses.unlicense;
  maintainers = [];
};
}
```

`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:

Exception: If the application has cargo `git` dependencies, the `cargoHash/cargoSha256` approach will not work, and you will need to copy the `Cargo.lock` file of the application to `nixpkgs` and continue with the next section for specifying the options of the `cargoLock` section.

```
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;
```

v: stable -

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;
    hash = "sha256-aDQA4A5mScX9or3Lyiv/5GyAehidnpKKE0grhbP1Ctc=";
  };

  cargoHash = "sha256-tbrTbutUs5aPSV+yE0IBUZAAytgmZV7Eqxia7g+9zRs=";
  cargoDepsName = pname;

  # ...
}
```

Importing a `Cargo.lock` file

v: stable -

Using `cargoSha256` or `cargoHash` is tedious when using `buildRustPackage` within a project, since it requires that the hash is updated after every change to `Cargo.lock`. Therefore, `buildRustPackage` also supports vendoring dependencies directly from a `Cargo.lock` file using the `cargoLock` argument. For example:

```
rustPlatform.buildRustPackage {  
  pname = "myproject";  
  version = "1.0.0";  
  
  cargoLock = {  
    lockFile = ./Cargo.lock;  
  };  
  
  # ...  
}
```

This will retrieve the dependencies using fixed-output derivations from the specified lockfile.

One caveat is that `Cargo.lock` cannot be patched in the `patchPhase` because it runs after the dependencies have already been fetched. If you need to patch or generate the lockfile you can alternatively set `cargoLock.lockFileContents` to a string of its contents:

```
rustPlatform.buildRustPackage {  
  pname = "myproject";  
  version = "1.0.0";  
  
  cargoLock = let  
    fixupLockFile = path: f (builtins.readFile path);  
  in {  
    lockFileContents = fixupLockFile ./Cargo.lock;  
  };  
  
  # ...  
}
```

Note that setting `cargoLock.lockFile` or `cargoLock.lockFileContents` doesn't a v: stable -

`Cargo.lock` to your `src`, and a `Cargo.lock` is still required to build a rust package. A simple fix is to use:

```
postPatch = ''  
  ln -s ${./Cargo.lock} Cargo.lock  
'';
```

The output hash of each dependency that uses a git source must be specified in the `outputHashes` attribute. For example:

```
rustPlatform.buildRustPackage rec {  
  pname = "myproject";  
  version = "1.0.0";  
  
  cargoLock = {  
    lockFile = ./Cargo.lock;  
    outputHashes = {  
      "finalfusion-0.14.0" = "17f4bsdzpcshwh74w5z119xjy2if6l2wgyjy56v621sl  
    };  
  };  
  
  # ...  
}
```

If you do not specify an output hash for a git dependency, building the package will fail and inform you of which crate needs to be added. To find the correct hash, you can first use `lib.fakeSha256` or `lib.fakeHash` as a stub hash. Building the package (and thus the vendored dependencies) will then inform you of the correct hash.

For usage outside nixpkgs, `allowBuiltInFetchGit` could be used to avoid having to specify `outputHashes`. For example:

```
rustPlatform.buildRustPackage rec {  
  pname = "myproject";  
  version = "1.0.0";
```

v: stable -

```
cargoLock = {  
    lockFile = ./Cargo.lock;  
    allowBuiltinFetchGit = true;  
};  
  
# ...  
}
```

Cargo features

You can disable default features using `buildNoDefaultFeatures`, and extra features can be added with `buildFeatures`.

If you want to use different features for check phase, you can use `checkNoDefaultFeatures` and `checkFeatures`. They are only passed to `cargo test` and not `cargo build`. If left unset, they default to `buildNoDefaultFeatures` and `buildFeatures`.

For example:

```
rustPlatform.buildRustPackage rec {  
  pname = "myproject";  
  version = "1.0.0";  
  
  buildNoDefaultFeatures = true;  
  buildFeatures = [ "color" "net" ];  
  
  # disable network features in tests  
  checkFeatures = [ "color" ];  
  
  # ...  
}
```

Cross compilation

By default, Rust packages are compiled for the host platform, just like any other package is v: stable -

--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:

v: stable -

```
--target /nix/store/asdfasdfsadf-thumb-crazy.json # contains {"foo":"","
```

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.

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., `--package foo`, 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](#).

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.

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.

This can be achieved with `--skip` in `checkFlags`:

```
rustPlatform.buildRustPackage {  
  /* ... */  
  checkFlags = [  
    # reason for disabling test  
    "--skip=example::tests:example_test"  
  ];  
}
```

Using cargo-nextest

Tests can be run with [cargo-nextest](#) by setting `useNextest = true`. The same options still apply, but nextest accepts a different set of arguments and the settings might need to be adapted to be compatible with cargo-nextest.

```
rustPlatform.buildRustPackage {  
  /* ... */  
  useNextest = true;  
}
```

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;  
}
```

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.

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/`.

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
  ];
}
```

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.

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/oEMqi860mt3zTQcXFWqrek0=";
};
```

The `src` attribute is required, as well as a hash specified through one of the `hash` attribute. 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.

If a `Cargo.lock` file is available, you can alternatively use the `importCargoLock` function. In contrast to `fetchCargoTarball`, this function does not require a hash (unless git dependencies are used) and fetches every dependency as a separate fixed-output derivation. `importCargoLock` can be used as follows:

```
cargoDeps = rustPlatform.importCargoLock {  
    lockFile = ./Cargo.lock;  
};
```

If the `Cargo.lock` file includes git dependencies, then their output hashes need to be specified since they are not available through the lock file. For example:

```
cargoDeps = rustPlatform.importCargoLock {  
    lockFile = ./Cargo.lock;  
    outputHashes = {  
        "rand-0.8.3" = "0ya2hia3cn31qa8894s3av2s8j5bjwb6yq92k0jsnlx7jid0jwqa"  
    };  
};
```

If you do not specify an output hash for a git dependency, building `cargoDeps` will fail and inform you of which crate needs to be added. To find the correct hash, you can first use `lib.fakeSha256` or `lib.fakeHash` as a stub hash. Building `cargoDeps` will then inform you of the correct hash.

Hooks

`rustPlatform` provides the following hooks to automate Cargo builds:

- **cargoSetupHook**: configure Cargo to use dependencies 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`.

v: stable -

- **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. Features can be specified with `cargoBuildNoDefaultFeatures` and `cargoBuildFeatures`. 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`. Features can be specified with `cargoCheckNoDefaultFeatures` and `cargoCheckFeatures`. 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`.
- **cargoNextestHook**: run tests using [cargo-nextest](#). The same options for `cargoCheckHook` also applies to `cargoNextestHook`.
- **cargoInstallHook**: install binaries and static/shared libraries that were built using `cargoBuildHook`.
- **bindgenHook**: for crates which use `bindgen` as a build dependency, lets `bindgen` find `libclang` and `libclang` find the libraries in `buildInputs`.

Examples

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:

v: stable -

```
{ fetchFromGitHub
, buildPythonPackage
, cargo
, rustPlatform
, rustc
, 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-miW//pn0mww2i6SOGbkrAIdc/JMDT4FJLqdMFojZeoY=";
};

sourceRoot = "${src.name}/bindings/python";

nativeBuildInputs = [
  cargo
  rustPlatform.cargoSetupHook
  rustc
  setuptools-rust
];

# ...
}
```

v: stable -

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;
    hash = "sha256-xGDilsjL0nls3MfVbGKnj80KCUCCzzXlis5PmHzpNcQ=";
  };

  cargoDeps = rustPlatform.fetchCargoTarball {
    inherit src;
    sourceRoot = "${pname}-${version}/${cargoRoot}";
    name = "${pname}-${version}";
    hash = "sha256-PS562W4L1NimqDV2H0jl5vYhL08H9est/pbIxSdYVfo=";
  };

  cargoRoot = "src/rust";

  # ...
}
```

Python package using maturin

v: stable -

Python packages that use [Maturin](#) can be built with `fetchCargoTarball`, `cargoSetupHook`, and `maturinBuildHook`. For example, the following (partial) derivation builds the `networkx` 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 = "networkx";
  version = "0.6.0";

  src = fetchFromGitHub {
    owner = "Qiskit";
    repo = "networkx";
    rev = version;
    hash = "sha256-11n30ldg3y3y6qxg3hbj837pnbwjkqw3nxq6frds647mmmprrd20=";
  };

  cargoDeps = rustPlatform.fetchCargoTarball {
    inherit src;
    name = "${pname}-${version}";
    hash = "sha256-he0BK8qi2nuc/Ib+I/vLzZ1fUUD/G/KTw9d7M4Hz500=";
  };

  format = "pyproject";

  nativeBuildInputs = with rustPlatform; [ cargoSetupHook maturinBuildHook
# ...
}
```

v: stable -

buildRustCrate: Compiling Rust crates using Nix instead of Cargo

Simple operation

When run, `cargo build` produces a file called `Cargo.lock`, containing pinned versions of all dependencies. Nixpkgs contains a tool called `crate2Nix` (`nix-shell -p crate2nix`), 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.

See [crate2nix's documentation](#) for instructions on how to use it.

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 separate 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> {};
```

v: stable -

```
((import ./hello.nix).hello {}).override {
  crateOverrides = defaultCrateOverrides // {
    hello = attrs: lib.optionalAttrs (lib.versionAtLeast attrs.version "1"
      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 = []; };
  };
}
```

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"
  '';
};
```

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 its *normal* compilation mechanism (`pkgConfig`).

v: stable -

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
```

Using community maintained Rust toolchains

Note

The following projects cannot be used within Nixpkgs since [Import From Derivation](#) (IFD) is disallowed in Nixpkgs. To package things that require Rust nightly, `RUSTC_BOOTSTRAP = true`; can sometimes be used as a hack.

There are two community maintained approaches to Rust toolchain management:

- [oxalica's Rust overlay](#)
- [fenix](#)

Despite their names, both projects provides a similar set of packages and overlays under different APIs.

Oxalica's overlay allows you to select a particular Rust version without you providing a hash or a flake input, but comes with a larger git repository than fenix.

Fenix also provides rust-analyzer nightly in addition to the Rust toolchains.

Both oxalica's overlay and fenix better integrate with nix and cache optimizations. Because of this and ergonomics, either of those community projects should be preferred to the Mozilla's Rust overlay ([nixpkgs-mozilla](#)).

The following documentation demonstrates examples using fenix and oxalica's Rust overlay with **nix-shell** and building derivations. More advanced usages like flake usage are documented in their own repositories.

Using Rust nightly with **nix-shell**

Here is a simple `shell.nix` that provides Rust nightly (default profile) using fenix:

```
with import <nixpkgs> { };
let
  fenix = callPackage
    (fetchFromGitHub {
      owner = "nix-community";
      repo = "fenix";
      # commit from: 2023-03-03
      rev = "e2ea04982b892263c4d939f1cc3bf60a9c4deaa1";
      hash = "sha256-As0im1A8KKtMWIxG+lXh5Q4P2bh0ZjoUhFWJ1EuZNNk=";
    })
    { };
in
mkShell {
```

v: stable -

```
name = "rust-env";
nativeBuildInputs = [
  # Note: to use stable, just replace `default` with `stable`
  fenix.default.toolchain

  # Example Build-time Additional Dependencies
  pkg-config
];
buildInputs = [
  # Example Run-time Additional Dependencies
  openssl
];

# Set Environment Variables
RUST_BACKTRACE = 1;
}
```

Save this to `shell.nix`, then run:

```
$ rustc --version
rustc 1.69.0-nightly (13471d3b2 2023-03-02)
```

To see that you are using nightly.

Oxalica's Rust overlay has more complete examples of `shell.nix` (and cross compilation) under its [examples directory](#).

Using Rust nightly in a derivation with `buildRustPackage`

You can also use Rust nightly to build rust packages using `makeRustPlatform`. The below snippet demonstrates invoking `buildRustPackage` with a Rust toolchain from oxalica's overlay:

```
with import <nixpkgs>
{
  overlays = [
    (import (fetchTarball "https://github.com/oxalica/rust-overl v: stable - ")
```

```
];
};

let
  rustPlatform = makeRustPlatform {
    cargo = rust-bin.stable.latest.minimal;
    rustc = rust-bin.stable.latest.minimal;
  };
in

rustPlatform.buildRustPackage rec {
  pname = "ripgrep";
  version = "12.1.1";

  src = fetchFromGitHub {
    owner = "BurntSushi";
    repo = "ripgrep";
    rev = version;
    hash = "sha256-+s5RBC3XSgb8omTbUNLywZnP6jSxZBKSS1BmX0jRF8M=";
  };
}

cargoHash = "sha256-l1vL2ZdtDRxSGvP0X/l3nMw8+6WF67KPutJEzUROjg8=";

doCheck = false;

meta = with lib; {
  description = "A fast line-oriented regex search tool, similar to ag and grep";
  homepage = "https://github.com/BurntSushi/ripgrep";
  license = with licenses; [ mit unlicense ];
  maintainers = with maintainers; [];
};
}
```

Follow the below steps to try that snippet.

1. save the above snippet as `default.nix` in that directory
2. cd into that directory and run `nix-build`

v: stable -

Fenix also has examples with `buildRustPackage`, [crane](#), [naersk](#), and cross compilation in its [Examples](#) section.

Using `git bisect` on the Rust compiler

Sometimes an upgrade of the Rust compiler (`rustc`) will break a downstream package. In these situations, being able to `git bisect` the `rustc` version history to find the offending commit is quite useful. Nixpkgs makes it easy to do this.

First, roll back your nixpkgs to a commit in which its `rustc` used *the most recent one which doesn't have the problem*. You'll need to do this because of `rustc`'s extremely aggressive version-pinning.

Next, add the following overlay, updating the Rust version to the one in your rolled-back nixpkgs, and replacing `/git/scratch/rust` with the path into which you have `git cloned` the `rustc` git repository:

```
(final: prev: /*lib.optionalAttrs prev.stdenv.targetPlatform.isAArch64*/
rust_1_72 =
  lib.updateManyAttrsByPath [
    path = [ "packages" "stable" ];
    update = old: old.overrideScope(final: prev: {
      rustc = prev.rustc.overrideAttrs (_: {
        src = lib.cleanSource /git/scratch/rust;
        # do *not* put passthru.isReleaseTarball=true here
      });
    });
  ]
  prev.rust_1_72;
})
```

If the problem you're troubleshooting only manifests when cross-compiling you can uncomment the `lib.optionalAttrs` in the example above, and replace `isAArch64` with the target that is having problems. This will speed up your bisect quite a bit, since the host compiler won't need to be rebuilt.

Now, you can start a `git bisect` in the directory where you checked out the `rustc` source code. It is recommended to select the endpoint commits by searching backwards from `origin/main v: stable -`

commits which added the release notes for the versions in question. If you set the endpoints to commits on the release branches (i.e. the release tags), git-bisect will often get confused by the complex merge-commit structures it will need to traverse.

The command loop you'll want to use for bisecting looks like this:

```
git bisect {good,bad} # depending on result of last build
git submodule update --init
CARGO_NET_OFFLINE=false cargo vendor \
--sync ./src/tools/cargo/Cargo.toml \
--sync ./src/tools/rust-analyzer/Cargo.toml \
--sync ./compiler/rustc_codegen_cranelift/Cargo.toml \
--sync ./src/bootstrap/Cargo.toml
nix-build $NIXPKGS -A package-broken-by-rust-changes
```

The `git submodule update --init` and `cargo vendor` commands above require network access, so they can't be performed from within the `rustc` derivation, unfortunately.

Swift

[Module search paths](#)

[Core libraries](#)

[Packaging with SwiftPM](#)

[Considerations for custom build tools](#)

The Swift compiler is provided by the `swift` package:

```
# Compile and link a simple executable.
nix-shell -p swift --run 'swiftc -' <<< 'print("Hello world!")'
# Run it!
./main
```

v: stable -

The `swift` package also provides the `swift` command, with some caveats:

- Swift Package Manager (SwiftPM) is packaged separately as `swiftpm`. If you need functionality like `swift build`, `swift run`, `swift test`, you must also add the `swiftpm` package to your closure.
- On Darwin, the `swift repl` command requires an Xcode installation. This is because it uses the system LLDB debugserver, which has special entitlements.

Module search paths

Like other toolchains in Nixpkgs, the Swift compiler executables are wrapped to help Swift find your application's dependencies in the Nix store. These wrappers scan the `buildInputs` of your package derivation for specific directories where Swift modules are placed by convention, and automatically add those directories to the Swift compiler search paths.

Swift follows different conventions depending on the platform. The wrappers look for the following directories:

- On Darwin platforms: `lib/swift/macosx` (If not targeting macOS, replace `macosx` with the Xcode platform name.)
- On other platforms: `lib/swift/linux/x86_64` (Where `linux` and `x86_64` are from lowercase `uname -sm`.)
- For convenience, Nixpkgs also adds `lib/swift` to the search path. This can save a bit of work packaging Swift modules, because many Nix builds will produce output for just one target any way.

Core libraries

In addition to the standard library, the Swift toolchain contains some additional 'core libraries' that, on Apple platforms, are normally distributed as part of the OS or Xcode. These are packaged separately in Nixpkgs, and can be found (for use in `buildInputs`) as:

- `swiftPackages.Dispatch`
- `swiftPackages.Foundation`

v: stable -

- `swiftPackages.XCTest`

Packaging with SwiftPM

Nixpkgs includes a small helper `swiftpm2nix` that can fetch your SwiftPM dependencies for you, when you need to write a Nix expression to package your application.

The first step is to run the generator:

```
cd /path/to/my/project
# Enter a Nix shell with the required tools.
nix-shell -p swift swiftpm swiftpm2nix
# First, make sure the workspace is up-to-date.
swift package resolve
# Now generate the Nix code.
swiftpm2nix
```

This produces some files in a directory `nix`, which will be part of your Nix expression. The next step is to write that expression:

```
{ stdenv, swift, swiftpm, swiftpm2nix, fetchFromGitHub }:

let
  # Pass the generated files to the helper.
  generated = swiftpm2nix.helpers ./nix;
in

stdenv.mkDerivation rec {
  pname = "myproject";
  version = "0.0.0";

  src = fetchFromGitHub {
    owner = "nixos";
    repo = pname;
    rev = version;
    hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
```

v: stable -

```
};

# Including SwiftPM as a nativeBuildInput provides a buildPhase for you
# This by default performs a release build using SwiftPM, essentially:
#   swift build -c release
nativeBuildInputs = [ swift swiftpm ];

# The helper provides a configure snippet that will prepare all dependencies
# in the correct place, where SwiftPM expects them.
configurePhase = generated.configure;

installPhase = ''
  # This is a special function that invokes swiftpm to find the location
  # of the binaries it produced.
  binPath="$(swiftpmBinPath)"
  # Now perform any installation steps.
  mkdir -p $out/bin
  cp $binPath/myproject $out/bin/
 '';
}

}
```

Custom build flags

If you'd like to build a different configuration than `release`:

```
swiftpmBuildConfig = "debug";
```

It is also possible to provide additional flags to `swift build`:

```
swiftpmFlags = [ "--disable-dead-strip" ];
```

The default `buildPhase` already passes `-j` for parallel building.

If these two customization options are insufficient, provide your own `buildPhase` that invokes `swift build`.

v: stable -

Running tests

Including `swiftpm` in your `nativeBuildInputs` also provides a default `checkPhase`, but it must be enabled with:

```
doCheck = true;
```

This essentially runs: `swift test -c release`

Patching dependencies

In some cases, it may be necessary to patch a SwiftPM dependency. SwiftPM dependencies are located in `.build/checkouts`, but the `swiftpm2nix` helper provides these as symlinks to read-only `/nix/store` paths. In order to patch them, we need to make them writable.

A special function `swiftpmMakeMutable` is available to replace the symlink with a writable copy:

```
configurePhase = generated.configure ++ ''
  # Replace the dependency symlink with a writable copy.
  swiftpmMakeMutable swift-crypto
  # Now apply a patch.
  patch -p1 -d .build/checkouts/swift-crypto -i ${./some-fix.patch}
'';
```

Considerations for custom build tools

Linking the standard library

The `swift` package has a separate `lib` output containing just the Swift standard library, to prevent Swift applications needing a dependency on the full Swift compiler at run-time. Linking with the Nixpkgs Swift toolchain already ensures binaries correctly reference the `lib` output.

Sometimes, Swift is used only to compile part of a mixed codebase, and the link step is manual. Custom build tools often locate the standard library relative to the `swift` compiler executable, and while the result will work, when this path ends up in the binary, it will have the Swift compiler as an unintended dependency.

v: stable -

In this case, you should investigate how your build process discovers the standard library, and override the path. The correct path will be something like:

```
"${swift.swift.lib}/${swift.swiftModuleSubdir}"
```

TeX Live

[User's guide \(experimental new interface\)](#)

[User's guide](#)

[Custom packages](#)

Since release 15.09 there is a new TeX Live packaging that lives entirely under attribute `texlive`.

User's guide (experimental new interface)

Release 23.11 ships with a new interface that will eventually replace `texlive.combine`.

- For basic usage, use some of the prebuilt environments available at the top level, such as `texliveBasic`, `texliveSmall`. For the full list of prebuilt environments, inspect `texlive.schemes`.
- Packages cannot be used directly but must be assembled in an environment. To create or add packages to an environment, use

```
texliveSmall.withPackages (ps: with ps; [ collection-langkorean algorithm ])
```

The function `withPackages` can be called multiple times to add more packages.

- **Note.** Within Nixpkgs, packages should only use prebuilt environments as inputs, such as `texliveSmall` or `texliveInfraOnly`, and should not depend directly on `texlive`. Further dependencies should be added by calling `withPackages`. This is to ensure that there is a consistent and simple way to override the inputs.
- `texlive.withPackages` uses the same logic as `buildEnv`. Only parts of a package

v: stable -

in an environment: its ‘runtime’ files (`tex` output), binaries (`out` output), and support files (`tlpkg` output). Moreover, man and info pages are assembled into separate `man` and `info` outputs. To add only the TeX files of a package, or its documentation (`texdoc` output), just specify the outputs:

```
texlive.withPackages (ps: with ps; [  
    texdoc # recommended package to navigate the documentation  
    perlPackages.LaTeXML.tex # tex files of LaTeXML, omit binaries  
    cm-super  
    cm-super.texdoc # documentation of cm-super  
)
```

- All packages distributed by TeX Live, which contains most of CTAN, are available and can be found under `texlive.pkgs`:

```
$ nix repl  
nix-repl> :l <nixpkgs>  
nix-repl> texlive.pkgs.[TAB]
```

Note that the packages in `texlive.pkgs` are only provided for search purposes and must not be used directly.

- **Experimental and subject to change without notice:** to add the documentation for all packages in the environment, use

```
texliveSmall.__overrideTeXConfig { withDocs = true; }
```

This can be applied before or after calling `withPackages`.

The function currently support the parameters `withDocs`, `withSources`, and `requireTeXPackages`.

User’s guide

- For basic usage just pull `texlive.combined.scheme-basic` for an environment with basic LaTeX support.

v: stable -

- It typically won't work to use separately installed packages together. Instead, you can build a custom set of packages like this. Most CTAN packages should be available:

```
texlive.combine {  
    inherit (texlive) scheme-small collection-langkorean algorithms cm-sup  
}
```

- 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.hasManpages || pkg  
        # 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.
- TeX Live packages are also available under `texlive.pkgs` as derivations with outputs `out`, `tex`, `texdoc`, `texsource`, `tlpkg`, `man`, `info`. They cannot be installed outside of `texlive.combine` but are available for other uses. To repackage a font, for instance, use

```
stdenvNoCC.mkDerivation rec {
  src = texlive.pkgs.iwona;

  inherit (src) pname version;

  installPhase = ''
    runHook preInstall
    install -Dm644 fonts/opentype/nowacki/iwona/*.otf -t $out/share/font
    runHook postInstall
  '';
}

}
```

See `biber`, `iwona` for complete examples.

Custom packages

You may find that you need to use an external TeX package. A derivation for such package has to provide the contents of the "texmf" directory in its "`tex`" output, according to the [TeX Directory Structure](#). Dependencies on other TeX packages can be listed in the attribute `tlDeps`.

The functions `texlive.combine` and `texlive.withPackages` recognise the following outputs:

- "`out`": contents are linked in the TeX Live environment, and binaries in the `$out/bin` folder are wrapped;
- "`tex$TEXMFDIST`; files should follow the TDS (for instance `$tex/tex/latex/foiltex/foiltex.cls`);
- "`texdoc`", "`texsourcetex`";
- "`tlpkg$TEXMFROOT/tlpkg`;
- "`man`", "`info`", ...: the other outputs are combined into separate outputs.

When using `pkgFilter`, `texlive.combine` will assign `tlType` respectively "`bin`", "`run`", "`doc`", "`source`", "`tlpkg`" to the above outputs.

Here is a (very verbose) example. See also the packages `auctex`, `eukleides`, `mftrace` f v: stable -

examples.

```
with import <nixpkgs> {};

let
  foiltex = stdenvNoCC.mkDerivation {
    pname = "latex-foiltex";
    version = "2.1.4b";

    outputs = [ "tex" "texdoc" ];
    passthru.tlDeps = with texlive; [ latex ];

    srcs = [
      (fetchurl {
        url = "http://mirrors.ctan.org/macros/latex/contrib/foiltex/foiltex.sty";
        hash = "sha256-/2I2xHXpZi0S988uFsGuPV6hhMw8e0U5m/P8myf42R0=";
      })
      (fetchurl {
        url = "http://mirrors.ctan.org/macros/latex/contrib/foiltex/foiltex.ltx";
        hash = "sha256-KTm3pkd+Cpu0nSE2WfsNEa56PeXBaNfx/s002Vv0kyc=";
      })
    ];
    unpackPhase = ''
      runHook preUnpack

      for _src in $srcs; do
        cp "$_src" ${stripHash "$_src"}
      done

      runHook postUnpack
    '';
  };

  nativeBuildInputs = [
    (texliveSmall.withPackages (ps: with ps; [ cm-super hypdoc latexmk ]))
    # multiple-outputs.sh fails if $out is not defined
    (writeShellScript "force-tex-output.sh" ''
      v: stable -
```

```
        out="''${tex-}"
    '')
];

dontConfigure = true;

buildPhase = ''
  runHook preBuild

  # Generate the style files
  latex foiltex.ins

  # Generate the documentation
  export HOME=.
  latexmk -pdf foiltex.dtx

  runHook postBuild
';

installPhase = ''
  runHook preInstall

  path="$tex/tex/latex/foiltex"
  mkdir -p "$path"
  cp *.{cls,def,clo,sty} "$path/"

  path="$texdoc/doc/tex/latex/foiltex"
  mkdir -p "$path"
  cp *.pdf "$path/"

  runHook postInstall
';

meta = with lib; {
  description = "A LaTeX2e class for overhead transparencies";
  license = licenses.unfreeRedistributable;
  maintainers = with maintainers; [ veprbl ];
  platforms = platforms.all;
```

v: stable -

```
};

};

latex_with_foiltex = texliveSmall.withPackages (_: [ 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
''
```

Titanium

[Building a Titanium app](#)

[Emulating or simulating the app](#)

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.

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:

v: stable -

- 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: v: stable -

- `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.

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 {}`.

Vim

[Custom configuration](#)

[Managing plugins with Vim packages](#)

[Managing plugins with vim-plug](#)

[Adding new plugins to nixpkgs](#)

[Updating plugins in nixpkgs](#)

[How to maintain an out-of-tree overlay of vim plugins ?](#)

Both Neovim and Vim can be configured to include your favorite plugins and additional libraries.

v: stable -

Loading can be deferred; see examples.

At the moment we support two different methods for managing plugins:

- Vim packages (*recommended*)
- vim-plug (vim only)

Right now two Vim packages are available: `vim` which has most features that require extra dependencies disabled and `vim-full` which has them configurable and enabled by default.

Note

`vim_configurable` is a deprecated alias for `vim-full` and refers to the fact that its build-time features are configurable. It has nothing to do with user configuration, and both the `vim` and `vim-full` packages can be customized as explained in the next section.

Custom configuration

Adding custom `.vimrc` lines can be done using the following code:

```
vim-full.customize {
    # `name` optionally 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`. You can also omit `name` to customize Vim itself. See the [definition of `vimUtils.makeCustomizable`](#) for all supported options.

For Neovim the `configure` argument can be overridden to achieve the same:

```
neovim.override {
```

v: stable -

```
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 Neovim:

```
neovim-qt.override {
  neovim = neovim.override {
    configure = {
      customRC = ''
        # your custom configuration
      '';
    };
  };
}
```

Managing plugins with Vim packages

To store your plugins in Vim packages (the native Vim plugin manager, see `:help packages`) the following example can be used:

```
vim-full.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 listed
    # 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 line
    # autocmd FileType php :packadd phpCompletion
  };
};
```

v: stable -

{

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 confus:  
    opt = [ ];  
  };  
};  
}
```

The resulting package can be added to `packageOverrides` in `~/.nixpkgs/config.nix` to make it installable:

```
{  
  packageOverrides = pkgs: with pkgs; {  
    myVim = vim-full.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 code from the example section here  
      };  
    };  
  };  
};
```

v: stable -

{

After that you can install your special grafted `myVim` or `myNeovim` packages.

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";
      hash = "sha256-bL33/S+caNmEYGcMLNCnFZyEYUOUmSsedCVBn4tV3g=";
    };
  };
in
{
  environment.systemPackages = [
    (
      pkgs.neovim.override {
        configure = {
          packages.myPlugins = with pkgs.vimPlugins; {
            start = [
              vim-go # already packaged plugin
              easygrep # custom package
            ];
            opt = [];
          };
          # ...
        };
      }
    )
  ]
}
```

v: stable -

```
];
}
```

If your package requires building specific parts, use instead `pkgs.vimUtils.buildVimPlugin`.

Specificities for some plugins

Treesitter

By default `nvim-treesitter` encourages you to download, compile and install the required Treesitter grammars at run time with `:TSInstall`. This works poorly on NixOS. Instead, to install the `nvim-treesitter` plugins with a set of precompiled grammars, you can use `nvim-treesitter.withPlugins` function:

```
(pkgs.neovim.override {
  configure = {
    packages.myPlugins = with pkgs.vimPlugins; {
      start = [
        (nvim-treesitter.withPlugins (
          plugins: with plugins; [
            nix
            python
          ]
        ))
      ];
    };
  });
})
```

To enable all grammars packaged in `nixpkgs`, use `pkgs.vimPlugins.nvim-treesitter.withAllGrammars`.

Managing plugins with vim-plug

To use [vim-plug](#) to manage your Vim plugins the following example can be used:

v: stable -

```
vim-full.customize {  
    vimrcConfig.packages.myVimPackage = with pkgs.vimPlugins; {  
        # loaded on launch  
        plug.plugins = [ youcompleteme fugitive phpCompletion elm-vim ];  
    };  
}
```

Note: this is not possible anymore for Neovim.

Adding new plugins to nixpkgs

Nix expressions for Vim plugins are stored in [pkgs/applications/editors/vim/plugins](#). For the vast majority of plugins, Nix expressions are automatically generated by running [`nix-shell -p vimPluginsUpdater --run vim-plugins-updater`](#). This creates a [generated.nix](#) file based on the plugins listed in [vim-plugin-names](#).

After running the updater, if nvim-treesitter received an update, also run [`nvim-treesitter/update.py`](#) to update the tree sitter grammars for [nvim-treesitter](#).

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`.

v: stable -

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 the Language Server Protocol integration with Vim/Neovim.

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/applications/editors/vim/plugins/update.py
```

Alternatively, set the number of processes to a lower count to avoid rate-limiting.

```
nix-shell -p vimPluginsUpdater --run 'vim-plugins-updater --proc 1'
```

How to maintain an out-of-tree overlay of vim plugins ?

You can use the updater script to generate basic packages out of a custom vim plugin list:

```
nix-shell -p vimPluginsUpdater --run vim-plugins-updater -i vim-plugin-names
```

with the contents of `vim-plugin-names` being for example:

```
repo,branch,alias
pwntester/octo.nvim,,
```

You can then reference the generated vim plugins via:

v: stable -

```
myVimPlugins = pkgs.vimPlugins.extend (pkgs.callPackage ./generated.nix {});
```

Packages

Table of Contents

[Citrix Workspace](#)

[darwin.linux-builder](#)

[DLib](#)

[Eclipse](#)

[Elm](#)

[Emacs](#)

[Firefox](#)

[Fish](#)

[FUSE](#)

[ibus-engines.typing-booster](#)

[Kakoune](#)

[Linux kernel](#)

[Locales](#)

[/etc files](#)

[Nginx](#)

[OpenGL](#)

[Interactive shell helpers](#)

[Steam](#)

[Cataclysm: Dark Days Ahead](#)

[Urxtv](#)

[WeeChat](#)

[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.

Citrix Workspace

[Basic usage](#)

[Citrix Self-service](#)

[Custom certificates](#)

The [Citrix Workspace App](#) is a remote desktop viewer which provides access to [XenDesktop](#) installations.

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.

Citrix Self-service

v: stable -

The [self-service](#) 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
```

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; }
```

darwin.linux-builder

[Example flake usage](#)

[Reconfiguring the remote builder](#)

[Troubleshooting the generated configuration](#)

`darwin.linux-builder` provides a way to bootstrap a Linux remote builder on a macO v: stable -

This requires macOS version 12.4 or later.

The remote builder runs on host port 31022 by default. You can change it by overriding `virtualisation.darwin-builder.hostPort`. See the [example](#).

You will also need to be a trusted user for your Nix installation. In other words, your `/etc/nix/nix.conf` should have something like:

```
extra-trusted-users = <your username goes here>
```

To launch the remote builder, run the following flake:

```
$ nix run nixpkgs#darwin.linux-builder
```

That will prompt you to enter your `sudo` password:

```
+ sudo --reset-timestamp /nix/store/...-install-credentials.sh ./keys  
Password:
```

... so that it can install a private key used to `ssh` into the build server. After that the script will launch the virtual machine and automatically log you in as the `builder` user:

```
<<< Welcome to NixOS 22.11.20220901.1bd8d11 (aarch64) - ttyAMA0 >>>  
  
Run 'nixos-help' for the NixOS manual.  
  
nixos login: builder (automatic login)  
  
[builder@nixos:~]$
```

Note: When you need to stop the VM, run `shutdown now` as the `builder` user.

To delegate builds to the remote builder, add the following options to your `nix.conf` file:

v: stable -

```
# - Replace ${ARCH} with either aarch64 or x86_64 to match your host machine
# - Replace ${MAX_JOBS} with the maximum number of builds (pick 4 if you're
builders = ssh-ng://builder@linux-builder ${ARCH}-linux /etc/nix/builder.conf

# Not strictly necessary, but this will reduce your disk utilization
builders-use-substitutes = true
```

To allow Nix to connect to a remote builder not running on port 22, you will also need to create a new file at `/etc/ssh/ssh_config.d/100-linu-x-builder.conf`:

```
Host linux-builder
  Hostname localhost
  HostKeyAlias linux-builder
  Port 31022
```

... and then restart your Nix daemon to apply the change:

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

Example flake usage

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixpkgs-22.11-darwin";
    darwin.url = "github:lnl7/nix-darwin/master";
    darwin.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = { self, darwin, nixpkgs, ... }@inputs:
  let

    inherit (darwin.lib) darwinSystem;
    system = "aarch64-darwin";
    pkgs = nixpkgs.legacyPackages."${system}";
```

v: stable -

```
linuxSystem = builtins.replaceStrings [ "darwin" ] [ "linux" ] system

darwin-builder = nixpkgs.lib.nixosSystem {
    system = linuxSystem;
    modules = [
        "${nixpkgs}/nixos/modules/profiles/macos-builder.nix"
        { virtualisation = {
            host.pkgs = pkgs;
            darwin-builder.workingDirectory = "/var/lib/darwin-builder";
        }};
    ];
};

in {

    darwinConfigurations = {
        machine1 = darwinSystem {
            inherit system;
            modules = [
                {
                    nix.distributedBuilds = true;
                    nix.buildMachines = [
                        {
                            hostName = "ssh://builder@localhost";
                            system = linuxSystem;
                            maxJobs = 4;
                            supportedFeatures = [ "kvm" "benchmark" "big-parallel" ];
                        }];
                };
            ];
        };

        launchd.daemons.darwin-builder = {
            command = "${darwin-builder.config.system.build.macos-builder}";
            serviceConfig = {
                KeepAlive = true;
                RunAtLoad = true;
                StandardOutPath = "/var/log/darwin-builder.log";
                StandardErrorPath = "/var/log/darwin-builder.log";
            };
        };
    };
}
```

v: stable -

```
];
};

};

};

}
```

Reconfiguring the remote builder

Initially you should not change the remote builder configuration else you will not be able to use the binary cache. However, after you have the remote builder running locally you may use it to build a modified remote builder with additional storage or memory.

To do this, you just need to set the `virtualisation.darwin-builder.*` parameters as in the example below and rebuild.

```
darwin-builder = nixpkgs.lib.nixosSystem {
    system = linuxSystem;
    modules = [
        "${nixpkgs}/nixos/modules/profiles/macos-builder.nix"
    {
        virtualisation.host.pkgs = pkgs;
        virtualisation.darwin-builder.diskSize = 5120;
        virtualisation.darwin-builder.memorySize = 1024;
        virtualisation.darwin-builder.hostPort = 33022;
        virtualisation.darwin-builder.workingDirectory = "/var/lib/darw
    }
];
};
```

You may make any other changes to your VM in this attribute set. For example, you could enable Docker or X11 forwarding to your Darwin host.

Troubleshooting the generated configuration

The `linux-builder` package exposes the attributes `nixosConfig` and `nixosOptions` that allow you to inspect the generated NixOS configuration in the `nix repl`. For example:

v: stable -

```
$ nix repl --file ~/src/nixpkgs --argstr system aarch64-darwin

nix-repl> darwin.linux-builder.nixosConfig.nix.package
«derivation /nix/store/...-nix-2.17.0.drv»

nix-repl> :p darwin.linux-builder.nixosOptions.virtualisation.memorySize.0
[ { file = "/home/user/src/nixpkgs/nixos/modules/profiles/macos-builder.n
```

DLib

[Compiling without AVX support](#)

[DLib](#) is a modern, C++-based toolkit which provides several machine learning algorithms.

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; }; }
```

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 `nix search -f eclipse`.

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 use the `v: stable` option:

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";
          hash = "sha256-123...";
        };
        srcPlugin = fetchurl {
          url = "http://.../plugins/myplugin1.jar";
          hash = "sha256-123...";
        };
      });
      (plugins.buildEclipseUpdateSite {
        name = "myplugin2-1.0";
        src = fetchurl {
          stripRoot = false;
          url = "http://.../myplugin2.zip";
          hash = "sha256-123...";
        };
      });
    ]);
  };
}
```

v: stable -

```
    });
];
};
}
```

Elm

To start a development environment, run:

```
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](#).

Emacs

[Configuring Emacs](#)

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 those 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.melpaStablePacka
      company
      counsel
      flycheck
      ivy
      v: stable -
```

```
    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 provides 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" ''
      (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))
  }
}
```

v: stable -

```
( "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.melpaStablePacka
(runCommand "default.el" {} ''
  mkdir -p $out/share/emacs/site-lisp
  cp ${myEmacsConfig} $out/share/emacs/site-lisp/default. v: stable -
```

```
  ' )
  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 personal 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 GNU-devel ELPA, and the highest for packages manually defined in `pkgs/applications/editors/emacs/elisp-packages/manual-packages`). But you can't control these priorities when some package is installed as a dependency. You can override it on a 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).withPackages
(p: with p; [
  # here both these package will use haskell-mode of our own choice
  ghc-mod
  dante
])
```

[Build wrapped Firefox with extensions and policies](#)

[Troubleshooting](#)

Build wrapped Firefox with extensions and policies

The `wrapFirefox` function allows to pass policies, preferences and extensions that are available to Firefox. With the help of `fetchFirefoxAddon` this allows to build a Firefox version that already comes with add-ons pre-installed:

```
{  
    # Nix firefox addons only work with the firefox-esr package.  
    myFirefox = wrapFirefox firefox-esr-unwrapped {  
        nixExtensions = [  
            (fetchFirefoxAddon {  
                name = "ublock"; # Has to be unique!  
                url = "https://addons.mozilla.org/firefox/downloads/file/3679754/";  
                hash = "sha256-2e73AbmYZlZXCP5ptYVcFjQYdjDp4iPoEPEOSCVF5sA=";  
            })  
        ];  
  
        extraPolicies = {  
            CaptivePortal = false;  
            DisableFirefoxStudies = true;  
            DisablePocket = true;  
            DisableTelemetry = true;  
            DisableFirefoxAccounts = true;  
            FirefoxHome = {  
                Pocket = false;  
                Snippets = false;  
            };  
            UserMessaging = {  
                ExtensionRecommendations = false;  
                SkipOnboarding = true;  
            };  
            SecurityDevices = {  
                v: stable -  
            };  
        };  
    };  
}
```

```
# Use a proxy module rather than `nixpkgs.config.firefox.smartcard`  
"PKCS#11 Proxy Module" = "${pkgs.p11-kit}/lib/p11-kit-proxy.so";  
};  
};  
  
extraPrefs = ''  
  // Show more ssl cert infos  
  lockPref("security.identityblock.show_extended_validation", true);  
'';  
};  
}
```

If `nixExtensions` != `null`, then all manually installed add-ons 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 add-ons do not have a valid signature, which is why signature verification is disabled. This does not compromise security because downloaded add-ons are checksummed and manual add-ons can't be installed. Also, make sure that the `name` field of `fetchFirefoxAddon` is unique. If you remove an add-on from the `nixExtensions` array, rebuild and start Firefox: the removed add-on will be completely removed with all of its settings.

Troubleshooting

If add-ons are marked as broken or the signature is invalid, make sure you have Firefox ESR installed. Normal Firefox does not provide the ability anymore to disable signature verification for add-ons thus nix add-ons get disabled by the normal Firefox binary.

If add-ons do not appear installed despite being defined in your nix configuration file, reset the local add-on state of your Firefox profile by clicking `Help -> More Troubleshooting Information -> Refresh Firefox`. This can happen if you switch from manual add-on mode to nix add-on mode and then back to manual mode and then again to nix add-on mode.

Fish

[Vendor Fish scripts](#)

[Packaging Fish plugins](#)

v: stable -

Fish wrapper

Fish is a “smart and user-friendly command line shell” with support for plugins.

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.

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.

Fish wrapper

The `wrapFish` package is a wrapper around Fish which can be used to create Fish shells initialized 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.

v: stable -

```
wrapFish {  
    pluginPkgs = with fishPlugins; [ pure foreign-env ];  
    completionDirs = [];  
    functionDirs = [];  
    confDirs = [ "/path/to/some/fish/init/dir/" ];  
}
```

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/w8bi72bssv0bnxhfw3xr1mvn7myf37x-sshfs-fuse-2  
Reason: image not found  
[1] 92299 abort      /nix/store/w8bi72bssv0bnxhfw3xr1mvn7myf37x-sshfs-
```

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 use `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 projects do define `FUSE_USE_VERSION`, but only inside C source files. This results in the `configure` script failing to find the header file.

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`.

ibus-engines.typing-booster

[Activating the engine](#)

[Using custom hunspell dictionaries](#)

[Built-in emoji picker](#)

This package is an ibus-based completion method to speed up typing.

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";
  };
}
```

v: stable -

```
ibus.engines = with pkgs.ibus-engines; [ typing-booster ];  
};  
}
```

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`.

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-color-emoji` is needed:

On NixOS, it can be installed using the following expression:

```
{ pkgs, ... }: {  
  fonts.packages = with pkgs; [ noto-fonts-color-emoji ];  
}
```

Kakoune

Kakoune can be built to autoload plugins:

```
(kakoune.override {  
  plugins = with pkgs.kakounePlugins; [ parinfer-rust ];  
})
```

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.iwlwif:
  ++ ...;
```

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 the `kernels` attribute set in `linux-kernels.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:
 - a. Make a 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`).
 - b. Copy the config file for this platform (e.g., `config-2.6.22-i686-smp`) to `.config` in the kernel source tree.
 - c. Run `make oldconfig ARCH={i386,x86_64,um}` and answer all questions:

v: stable -

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`).

d. If needed, you can also run `make menuconfig`:

```
$ nix-env -f "<nixpkgs>" -iA ncurses
$ export NIX_CFLAGS_LINK=-lncurses
$ make menuconfig ARCH=arch
```

e. Copy `.config` over the new config file (e.g., `config-2.6.22-i686-smp`).

4. Test building the kernel: `nix-build -A linuxKernel.kernels.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 `linux-kernels.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.

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.

/etc files

Certain calls in `glibc` require access to runtime files found in `/etc` such as `/etc/protocols` or `/etc/services` – `getprotobynumber` is one such function.

v: stable -

On non-NixOS distributions these files are typically provided by packages (i.e., [netbase](#)) if not already pre-installed in your distribution. This can cause non-reproducibility for code if they rely on these files being present.

If [iana-etc](#) is part of your `buildInputs`, then it will set the environment variables `NIX_ETC_PROTOCOLS` and `NIX_ETC_SERVICES` to the corresponding files in the package through a setup hook.

```
> nix-shell -p iana-etc

[nix-shell:~]$ env | grep NIX_ETC
NIX_ETC_SERVICES=/nix/store/aj866hr8fad8flnggwdhrlm0g799ccz-iana-etc-2021-02-10T14:22:12Z
NIX_ETC_PROTOCOLS=/nix/store/aj866hr8fad8flnggwdhrlm0g799ccz-iana-etc-2021-02-10T14:22:12Z
```

Nixpkg's version of [glibc](#) has been patched to check for the existence of these environment variables. If the environment variables are *not* set, then it will attempt to find the files at the default location within `/etc`.

Nginx

[ETags on static files served from the Nix store](#)

[Nginx](#) is a reverse proxy and lightweight webserver.

ETags on static files served from the Nix store

HTTP has a couple of 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).

v: stable -

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.

OpenGL

[NixOS Desktop](#)

[Nix on GNU/Linux](#)

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.

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.

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 `libglvnc` v: stable -

`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.

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 be used in the `.bashrc` like this:

```
source "$(fzf-share)/completion.bash"
source "$(fzf-share)/key-bindings.bash"
```

Steam

[Steam in Nix](#)

[How to play](#)

[Troubleshooting](#)

[steam-run](#)

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 v: stable -

\$HOME.

Nix problems and constraints:

- We don't have `/bin/bash` and many scripts point there. Same thing for `/usr/bin/python`.
- We don't have the dynamic loader in `/lib`.
- The `steam.sh` script in `$HOME` cannot 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.

How to play

Use `programs.steam.enable = true;` if you want to add steam to `systemPackages` and also enable a few workarounds as well as Steam controller support or other Steam supported controllers such as the DualShock 4 or Nintendo Switch Pro Controller.

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 conflicts with the one dynamically loaded by `radeon_dri.so`. If you get the error:

v: stable -

```
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: ja
```

you need to add:

```
steam.override { withJava = true; };
```

steam-run

The FHS-compatible chroot used for Steam can also be used to run other Linux games that expect a FHS environment. To use it, install the `steam-run` package and run the game with:

```
steam-run ./foo
```

Cataclysm: Dark Days Ahead

[How to install Cataclysm DDA](#)

[Important note for overriding packages](#)

[Customizing with mods](#)

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 `v: stable -`

`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/archive/refs/heads/main.zip"
```

The default configuration directory is `~/.cataclysm-dda`. If you prefer `$XDG_CONFIG_HOME/cataclysm-dda`, override the derivation:

```
cataclysm-dda.override {  
    useXdgDir = true;  
}
```

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
```

v: stable -

```
# 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
```

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:

let

```
customMods = self: super: lib.recursiveUpdate super {  
    # Modify existing mod
```

v: stable -

```
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 = "...";
    hash = "...";
  };
  # Path to be installed in the unpacked source (default: ".")
  modRoot = "contents/under/this/path/will/beinstalled";
};

# Add another soundpack
soundpack.Fantastic = cataclysmdDA.buildSoundPack {
  # ditto
};

# Add another tileset
tileset.SuperDuper = cataclysmdDA.buildTileSet {
  # ditto
};
};

in
cataclysmdda.withMods (mods: with mods.extend customMods; [
  tileset.UndeadPeople
  mod.Awesome
  soundpack.Fantastic
  tileset.SuperDuper
])
)
```

Urxvt

[Configuring urxvt](#)

[Packaging urxvt plugins](#)

Urxvt, also known as rxvt-unicode, is a highly customizable terminal emulator.

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

v: stable -

```
$ nix repl  
:l <nixpkgs>  
map (p: p.name) pkgs.rxvt-unicode.plugins
```

Alternatively, if your shell is bash or zsh and have completion enabled, 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 ];  
  };  
}
```

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.

v: stable -

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.

The project

[Channel Status](#)
[Packages search](#)
[Options search](#)
[Reproducible Builds Status](#)
[Security](#)

Get in Touch

[Forum](#)
[Matrix Chat](#)
[Commercial support](#)

Contribute

[Contributing Guide](#)
[Donate](#)

Stay up to date

[Blog](#)
[Newsletter](#)

```
plugins = builtins.attrvalues.availablePlugins // i
  python = availablePlugins.python.withPackages (ps: with ps; [ pycrypt...)
};

}; }
```

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 libera irc.libera.chat
    '';
  };
}
```

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"
    '';
  };
}
```

v: stable -

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";
    hash = "...";
  };
  passthru.scripts = [ "foo.py" "bar.lua" ];
  installPhase = ''
    mkdir $out/share
    cp foo.py $out/share
    cp bar.lua $out/share
  '';
}
```

X.org

[Katamari Tarballs](#)

[Individual Tarballs](#)

[Generating Nix Expressions](#)

[Overriding the Generator](#)

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.

v: stable -

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{ \
| sort > "tarballs-$release.list"
```

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.ime`.

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.)

Overriding the Generator

If the expression for a package requires derivation attributes that the generator cannot figure out, you can override the generator by defining them in the `pkgs/servers/x11/xorg/default.nix` file.

automatically (say, patches or a `postInstall` hook), you should modify `pkgs/servers/x11/xorg/overrides.nix`.

Development of Nixpkgs

This section shows you how Nixpkgs is being developed and how you can interact with the contributors and the latest updates. If you are interested in contributing yourself, see [CONTRIBUTING.md](#).

Table of Contents

[Opening issues](#)

Opening issues

- Make sure you have a [GitHub account](#)
- Make sure there is no open issue on the topic
- [Submit a new issue](#) by choosing the kind of topic and fill out the template

Contributing to Nixpkgs

Table of Contents

[Quick Start to Adding a Package](#)

[Coding conventions](#)

[Submitting changes](#)

[Vulnerability Roundup](#)

[Reviewing contributions](#)

[Contributing to Nixpkgs documentation](#)

Quick Start to Adding a Package

This section has been moved to [pkgs/README.md](#).

Coding conventions

Table of Contents

[Syntax](#)

[Package naming](#)

[File naming and organisation](#)

[Fetching Sources](#)

[Obtaining source hash](#)

[Patches](#)

[Package tests](#)

This section has been moved to [CONTRIBUTING.md](#).

v: stable -

Syntax

This section has been moved to [CONTRIBUTING.md](#).

Package naming

This section has been moved to [pkgs/README.md](#).

File naming and organisation

Versioning

This section has been moved to [CONTRIBUTING.md](#).

Versioning

This section has been moved to [pkgs/README.md](#).

Fetching Sources

This section has been moved to [pkgs/README.md](#).

Obtaining source hash

Obtaining hashes securely

This section has been moved to [pkgs/README.md](#).

Obtaining hashes securely

This section has been moved to [pkgs/README.md](#).

Patches

v: stable -

This section has been moved to [pkgs/README.md](#).

Package tests

[Writing inline package tests](#)

[Writing larger package tests](#)

[Running package tests](#)

[Examples of package tests](#)

[Linking NixOS module tests to a package](#)

[Import From Derivation](#)

This section has been moved to [pkgs/README.md](#).

Writing inline package tests

This section has been moved to [pkgs/README.md](#).

Writing larger package tests

This section has been moved to [pkgs/README.md](#).

Running package tests

This section has been moved to [pkgs/README.md](#).

Examples of package tests

This section has been moved to [pkgs/README.md](#).

Linking NixOS module tests to a package

v: stable -

This section has been moved to [pkgs/README.md](#).

Import From Derivation

This section has been moved to [pkgs/README.md](#).

Submitting changes

Table of Contents

[Submitting changes](#)

[Submitting security fixes](#)

[Deprecating/removing packages](#)

[Pull Request Template](#)

[Hotfixing pull requests](#)

[Commit policy](#)

This section has been moved to [CONTRIBUTING.md](#).

Submitting changes

This section has been moved to [CONTRIBUTING.md](#).

Submitting security fixes

This section has been moved to [pkgs/README.md](#).

Deprecating/removing packages

[Steps to remove a package from Nixpkgs](#)

This section has been moved to [pkgs/README.md](#).

Steps to remove a package from Nixpkgs

This section has been moved to [pkgs/README.md](#).

Pull Request Template

[Tested using sandboxing](#)

[Built on platform\(s\)](#)

[Tested via one or more NixOS test\(s\) if existing and applicable for the change \(look inside nixos/tests\)](#)

[Tested compilation of all pkgs that depend on this change using `nixpkgs-review`](#)

[Tested execution of all binary files \(usually in `./result/bin/`\)](#)

[Meets Nixpkgs contribution standards](#)

This section has been moved to [CONTRIBUTING.md](#).

Tested using sandboxing

This section has been moved to [CONTRIBUTING.md](#).

Built on platform(s)

This section has been moved to [CONTRIBUTING.md](#).

Tested via one or more NixOS test(s) if existing and applicable for the change (look inside nixos/tests)

This section has been moved to [CONTRIBUTING.md](#).

Tested compilation of all pkgs that depend on this change us v: stable -

nixpkgs-review

This section has been moved to [CONTRIBUTING.md](#).

Tested execution of all binary files (usually in `./result/bin/`)

This section has been moved to [CONTRIBUTING.md](#).

Meets Nixpkgs contribution standards

This section has been moved to [CONTRIBUTING.md](#).

Hotfixing pull requests

This section has been moved to [CONTRIBUTING.md](#).

Commit policy

[Branches](#)

This section has been moved to [CONTRIBUTING.md](#).

Branches

This section has been moved to [CONTRIBUTING.md](#).

Master branch

This section has been moved to [CONTRIBUTING.md](#).

Staging branch

This section has been moved to [CONTRIBUTING.md](#).

Staging-next branch

v: stable -

This section has been moved to [CONTRIBUTING.md](#).

Stable release branches

This section has been moved to [CONTRIBUTING.md](#).

Automatically backporting a Pull Request

This section has been moved to [CONTRIBUTING.md](#).

Manually backporting changes

This section has been moved to [CONTRIBUTING.md](#).

Acceptable backport criteria

This section has been moved to [CONTRIBUTING.md](#).

Vulnerability Roundup

Table of Contents

[Issues](#)

[Triaging and Fixing](#)

This section has been moved to [pkgs/README.md](#).

Issues

This section has been moved to [pkgs/README.md](#).

Triaging and Fixing

This section has been moved to [pkgs/README.md](#).

Reviewing contributions

Table of Contents

[Package updates](#)

[New packages](#)

[Module updates](#)

[New modules](#)

[Individual maintainer list](#)

[Maintainer teams](#)

[Other submissions](#)

[Merging pull requests](#)

This section has been moved to [CONTRIBUTING.md](#).

Package updates

This section has been moved to [pkgs/README.md](#).

New packages

This section has been moved to [pkgs/README.md](#).

Module updates

This section has been moved to [nixos/README.md](#).

New modules

This section has been moved to [nixos/README.md](#).

v: stable -

Individual maintainer list

This section has been moved to [maintainers/README.md](#).

Maintainer teams

This section has been moved to [maintainers/README.md](#).

Other submissions

This section has been moved to [CONTRIBUTING.md](#).

Merging pull requests

This section has been moved to [CONTRIBUTING.md](#).

Contributing to Nixpkgs documentation

Table of Contents

[devmode](#)

[Syntax](#)

This section has been moved to [doc/README.md](#).

devmode

This section has been moved to [doc/README.md](#).

Syntax

This section has been moved to [doc/README.md](#).