

build **passing**

## Navigation

### Nixpkgs User's Guide

- [How to install Haskell packages](#)
  - [How to install a branch of a package](#)
- [How to create a development environment](#)
  - [How to install a compiler](#)
  - [How to install a compiler with libraries](#)
  - [How to install a compiler with libraries, hoogle and documentation indexes](#)
  - [How to install haskell-language-server](#)
  - [How to make haskell-language-server find a GHC from nix-shell](#)
  - [How to build a Haskell project using Stack](#)
  - [How to create ad hoc environments for `nix-shell`](#)
- [How to create Nix builds for your own private Haskell packages](#)
  - [How to build a stand-alone project](#)
  - [How to build projects that depend on each other](#)

### Nixpkgs Developer's Guide

### Frequently Asked Questions

### External Resources

## Quick search

Go


# Nixpkgs User's Guide

## How to install Haskell packages

Nixpkgs distributes build instructions for all Haskell packages registered on [Hackage](#), but strangely enough normal Nix package lookups don't seem to discover any of them, except for the default version of `ghc`, `cabal-install`, and `stack`:

```
$ nix-env -i alex
error: selector 'alex' matches no derivations
$ nix-env -qa ghc
ghc-8.10.2
```

The Haskell package set is not registered in the top-level namespace because it is *huge*. If all Haskell packages were visible to these commands, then name-based search/install would be much slower than they are now. We avoided that by keeping all Haskell-related packages in a separate attribute set called `haskellPackages`, which the following command will list:

 v: latest ▼

```
$ nix-env -f "<nixpkgs>" -qaP -A haskellPackages
haskellPackages.a50          a50-0.5
haskellPackages.AAI          AAI-0.2.0.1
haskellPackages.abacate      abacate-0.0.0.0
haskellPackages.abc-puzzle   abc-puzzle-0.2.1
haskellPackages.abcBridge    abcBridge-0.15
haskellPackages.abcnotation  abcnotation-1.9.0
haskellPackages.abeson       abeson-0.1.0.1
[... some 14000 entries omitted ...]
```

To install any of those packages into your profile, refer to them by their attribute path (first column):

```
nix-env -f "<nixpkgs>" -iA haskellPackages.Allure ...
```

The attribute path of any Haskell packages corresponds to the name of that particular package on Hackage: the package `cabal-install` has the attribute `haskellPackages.cabal-install`, and so on. (Actually, this convention causes trouble with packages like `3dmodels` and `4Blocks`, because these names are invalid identifiers in the Nix language. The issue of how to deal with these rare corner cases is currently unresolved.)

Haskell packages whose Nix name (second column) begins with a `haskell-` prefix are packages that provide a library whereas packages without that prefix provide just executables. Libraries may provide executables too, though: the package `haskell-pandoc`, for example, installs both a library and an application. You can install and use Haskell executables just like any other program in Nixpkgs, but using Haskell libraries for development is a bit trickier and we'll address that subject in great detail in section [How to create a development environment](#).

Attribute paths are deterministic inside of Nixpkgs, but the path necessary to reach Nixpkgs varies from system to system. We dodged that problem by giving `nix-env` an explicit `-f "<nixpkgs>"` parameter, but if you call `nix-env` without that flag, then chances are the invocation fails:

```
$ nix-env -iA haskellPackages.cabal-install
error: attribute 'haskellPackages' in selection path
      'haskellPackages.cabal-install' not found
```

On NixOS, for example, Nixpkgs does *not* exist in the top-level namespace by default. To figure out the proper attribute path, it's easiest to query for the path of a well-known Nixpkgs package, i.e.:

```
$ nix-env -qaP coreutils
nixos.coreutils  coreutils-8.23
```

If your system responds like that (most NixOS installations will), then the attribute path to `haskellPackages` is `nixos.haskellPackages`. Thus, if you want to use `nix-env` without giving an explicit `-f` flag, then that's the way to do it:

```
nix-env -qaP -A nixos.haskellPackages
nix-env -iA nixos.haskellPackages.cabal-install
```

Our current default compiler is GHC 8.10.x and the `haskellPackages` set contains packages built with that particular version. Nixpkgs contains the last three major releases of GHC and there is a whole family of package sets available that defines Hackage packages built with each of those compilers, too:

```
nix-env -f "<nixpkgs>" -qaP -A haskell.packages.ghc8104
nix-env -f "<nixpkgs>" -qaP -A haskell.packages.ghc901
```

The name `haskellPackages` is really just a synonym for `haskell.packages.ghcXYZ` (where `XYZ` is current default GHC version in Nixpkgs), because we prefer that package set internally and recommend it to our users as their default choice, but ultimately you are free to compile your Haskell packages with any GHC version you please. The following command displays the complete list of available compilers:

```
$ nix-env -f "<nixpkgs>" -qaP -A haskell.compiler
haskell.compiler.ghc8102Binary      ghc-8.10.2-binary
haskell.compiler.ghc8102BinaryMinimal ghc-8.10.2-binary
haskell.compiler.ghc8104            ghc-8.10.4
haskell.compiler.integer-simple.ghc8104 ghc-8.10.4
haskell.compiler.ghcHEAD            ghc-8.11.20200824
haskell.compiler.native-bignum.ghcHEAD ghc-8.11.20200824
haskell.compiler.ghc865Binary       ghc-8.6.5-binary
haskell.compiler.ghc884             ghc-8.8.4
haskell.compiler.integer-simple.ghc884 ghc-8.8.4
haskell.compiler.ghc901             ghc-9.0.1
haskell.compiler.integer-simple.ghc901 ghc-9.0.1
```

We have no package sets for `jhc` or `uhc` yet, unfortunately, but for every version of GHC listed above, there exists a package set based on that compiler. Also, the attributes `haskell.compiler.ghcXYZ` and `haskell.packages.ghcXYZ.ghc` are synonymous for the sake of convenience.

## How to install a branch of a package

One of the nice things about Nix is that nixpkgs contains all information needed to build a package. This makes it easy to point a package to a different branch of the source and have Nix build a package for that branch.

Even though Haskell packages are typically generated based on the hackage releases, because hackage contains source packages this is still possible for hackage. You can use `overrideSrc` to override the source, for example:

```
my-hledger-lib = (haskell.lib.overrideSrc haskellPackages.hledger-lib {
  src = /home/aengelen/dev/hledger/hledger-lib;
});
my-hledger = (haskell.lib.overrideSrc haskellPackages.hledger {
  src = /home/aengelen/dev/hledger/hledger;
}).override {
  hledger-lib = my-hledger-lib;
};
hledger-web = haskell.lib.justStaticExecutables ((haskell.lib.overrideSrc haskellPacka
```

```
src = /home/aengelen/dev/hledger/hledger-web;
})
.override {
  hledger = my-hledger;
  hledger-lib = my-hledger-lib;
});
```

# How to create a development environment

## How to install a compiler

A simple development environment consists of a Haskell compiler and one or both of the tools `cabal-install` and `stack`. We saw in section [How to install Haskell packages](#) how you can install those programs into your user profile:

```
nix-env -f "<nixpkgs>" -iA haskellPackages.ghc haskellPackages.cabal-install
```

Instead of the default package set `haskellPackages`, you can also use the more precise name `haskell.compiler.ghc7102`, which has the advantage that it refers to the same GHC version regardless of what Nixpkgs considers “default” at any given time.

Once you’ve made those tools available in `$PATH`, it’s possible to build Hackage packages the same way people without access to Nix do it all the time:

```
cabal get lens-4.11 && cd lens-4.11
cabal install -j --dependencies-only
cabal configure
cabal build
```

If you enjoy working with Cabal sandboxes, then that’s entirely possible too: just execute the command

```
cabal sandbox init
```

before installing the required dependencies.

The `nix-shell` utility makes it easy to switch to a different compiler version; just enter the Nix shell environment with the command

```
nix-shell -p haskell.compiler.ghc784
```

to bring GHC 7.8.4 into `$PATH`. Alternatively, you can use Stack instead of `nix-shell` directly to select compiler versions and other build tools per-project. It uses `nix-shell` under the hood when Nix support is turned on. See [How to build a Haskell project using Stack](#).

If you’re using `cabal-install`, re-running `cabal configure` inside the spawned shell switches your build to use that compiler instead. If you’re working on a project that doesn’t depend on any additional system libraries outside of GHC, then it’s even sufficient to just run the `cabal configure`

command inside of the shell:

```
nix-shell -p haskell.compiler.ghc784 --command "cabal configure"
```

Afterwards, all other commands like `cabal build` work just fine in any shell environment, because the configure phase recorded the absolute paths to all required tools like GHC in its build configuration inside of the `dist/` directory. Please note, however, that `nix-collect-garbage` can break such an environment because the Nix store paths created by `nix-shell` aren't "alive" anymore once `nix-shell` has terminated. If you find that your Haskell builds no longer work after garbage collection, then you'll have to re-run `cabal configure` inside of a new `nix-shell` environment.

## How to install a compiler with libraries

GHC expects to find all installed libraries inside of its own `lib` directory. This approach works fine on traditional Unix systems, but it doesn't work for Nix, because GHC's store path is immutable once it's built. We cannot install additional libraries into that location. As a consequence, our copies of GHC don't know any packages except their own core libraries, like `base`, `containers`, `Cabal`, etc.

We can register additional libraries to GHC, however, using a special build function called `ghcWithPackages`. That function expects one argument: a function that maps from an attribute set of Haskell packages to a list of packages, which determines the libraries known to that particular version of GHC. For example, the Nix expression `ghcWithPackages (pkgs: [pkgs.mtl])` generates a copy of GHC that has the `mtl` library registered in addition to its normal core packages:

```
$ nix-shell -p "haskellPackages.ghcWithPackages (pkgs: [pkgs.mtl])"

[nix-shell:~]$ ghc-pkg list mtl
/nix/store/zy79...-ghc-7.10.2/lib/ghc-7.10.2/package.conf.d:
    mtl-2.2.1
```

This function allows users to define their own development environment by means of an override. After adding the following snippet to `~/.config/nixpkgs/config.nix`,

```
{
  packageOverrides = super: let self = super.pkgs; in
  {
    myHaskellEnv = self.haskell.packages.ghc7102.ghcWithPackages
      (haskellPackages: with haskellPackages; [
        # Libraries
        arrows async cgi criterion
        # Tools
        cabal-install haskintex
      ]);
  };
}
```

it's possible to install that compiler with `nix-env -f "<nixpkgs>" -iA myHaskellEnv`. If you'd like to switch that development environment to a different version of GHC, just replace the `ghc7102` bit in the previous definition with the appropriate name. Of course, it's also possible to define any

number of these development environments! (You can't install two of them into the same profile at the same time, though, because that would result in file conflicts.)

The generated `ghc` program is a wrapper script that re-directs the real GHC executable to use a new `lib` directory — one that we specifically constructed to contain all those packages the user requested:

```
$ cat $(type -p ghc)
#!/nix/store/xlxj...-bash-4.3-p33/bin/bash -e
export NIX_GHC=/nix/store/19sm...-ghc-7.10.2/bin/ghc
export NIX_GHCPKG=/nix/store/19sm...-ghc-7.10.2/bin/ghc-pkg
export NIX_GHC_DOCDIR=/nix/store/19sm...-ghc-7.10.2/share/doc/ghc/html
export NIX_GHC_LIBDIR=/nix/store/19sm...-ghc-7.10.2/lib/ghc-7.10.2
exec /nix/store/j50p...-ghc-7.10.2/bin/ghc "-B$NIX_GHC_LIBDIR" "$@"
```

The variables `$NIX_GHC`, `$NIX_GHCPKG`, etc. point to the *new* store path `ghcWithPackages` constructed specifically for this environment. The last line of the wrapper script then executes the real `ghc`, but passes the path to the new `lib` directory using GHC's `-B` flag.

The purpose of those environment variables is to work around an impurity in the popular [ghc-paths](#) library. That library promises to give its users access to GHC's installation paths. Only, the library can't possibly know that path when it's compiled, because the path GHC considers its own is determined only much later, when the user configures it through `ghcWithPackages`. So we [patched ghc-paths](#) to return the paths found in those environment variables at run-time rather than trying to guess them at compile-time.

To make sure that mechanism works properly all the time, we recommend that you set those variables to meaningful values in your shell environment, too, i.e. by adding the following code to your `~/.bashrc`:

```
if type >/dev/null 2>&1 -p ghc; then
  eval "$(egrep ^export "$(type -p ghc)")"
fi
```

If you are certain that you'll use only one GHC environment which is located in your user profile, then you can use the following code, too, which has the advantage that it doesn't contain any paths from the Nix store, i.e. those settings always remain valid even if a `nix-env -u` operation updates the GHC environment in your profile:

```
if [ -e ~/.nix-profile/bin/ghc ]; then
  export NIX_GHC="$HOME/.nix-profile/bin/ghc"
  export NIX_GHCPKG="$HOME/.nix-profile/bin/ghc-pkg"
  export NIX_GHC_DOCDIR="$HOME/.nix-profile/share/doc/ghc/html"
  export NIX_GHC_LIBDIR="$HOME/.nix-profile/lib/ghc-${NIX_GHC --numeric-version}"
fi
```

## How to install a compiler with libraries, hoogle and documentation indexes

If you plan to use your environment for interactive programming, not just compiling random Haskell code, you might want to replace `ghcWithPackages` in all the listings above with `ghcWithHoogle`.

This environment generator not only produces an environment with GHC and all the specified libraries, but also generates a `hoogle` and `haddock` indexes for all the packages, and provides a wrapper script around `hoogle` binary that uses all those things. A precise name for this thing would be “`ghcWithPackagesAndHoogleAndDocumentationIndexes`”, which is, regrettably, too long and scary.

For example, installing the following environment

```
{
  packageOverrides = super: let self = super.pkgs; in
  {
    myHaskellEnv = self.haskellPackages.ghcWithHoogle
      (haskellPackages: with haskellPackages; [
        # libraries
        arrows async cgi criterion
        # tools
        cabal-install haskintex
      ]);
  };
}
```

allows one to browse module documentation index [not too dissimilar to this](#) for all the specified packages and their dependencies by directing a browser of choice to `~/.nix-profile/share/doc/hoogle/index.html` (or `/run/current-system/sw/share/doc/hoogle/index.html` in case you put it in `environment.systemPackages` in NixOS).

After you’ve marveled enough at that try adding the following to your `~/.ghc/ghci.conf`

```
:def hoogle \s -> return $ ":! hoogle search -cl --count=15 \"\" ++ s ++ \"\"\"
:doc \s -> return $ ":! hoogle search -cl --info \"\" ++ s ++ \"\"\"
```

and test it by typing into `ghci`:

```
:hoogle a -> a
:doc a -> a
```

Be sure to note the links to `haddock` files in the output. With any modern and properly configured terminal emulator you can just click those links to navigate there.

Finally, you can run

```
hoogle server --local -p 8080
```

and navigate to <http://localhost:8080/> for your own local `Hoogle`. The `--local` flag makes the `hoogle` server serve files from your nix store over http, without the flag it will use `file://` URIs. Note, however, that Firefox and possibly other browsers disallow navigation from `http://` to `file://` URIs for security reasons, which might be quite an inconvenience. Versions before v5 did



not have this flag. See [this page](#) for workarounds.

For NixOS users there's a service which runs this exact command for you. Specify the `packages` you want documentation for and the `haskellPackages` set you want them to come from. Add the following to `configuration.nix`.

```
services.hoogle = {  
  enable = true;  
  packages = (hpkgs: with hpkgs; [text cryptonite]);  
  haskellPackages = pkgs.haskellPackages;  
};
```

## How to install haskell-language-server

In short: Install `pkgs.haskell-language-server` and use the `haskell-language-server-wrapper` command to run it. See the [hls README](#) 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`, `haskell-language-server-x.x.x` and `haskell-language-server-x.x.x` binaries, where `x.x.x` is the ghc version for which it is compiled. By default it includes binaries for all ghc versions that are provided in the binary caches. You can override that list with e.g.

```
pkgs.haskell-language-server.override { supportedGhcVersions = [ "884" "901" ]; }
```

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 above mentioned binary from your path.

Be careful when installing hls globally and using a pinned nixpkgs for a Haskell project in a nix-shell. If the nixpkgs versions deviate to much (e.g. use different `glibc` versions) hls might fail. It is recommended to then install hls in the nix-shell from the nixpkgs version pinned in there.

If you know, that you only use one ghc version, e.g. in a project specific nix-shell You can either use an override as given above or simply install `pkgs.haskellPackages.haskell-language-server` instead of the top-level attribute `pkgs.haskell-language-server`.

## How to make haskell-language-server find a GHC from nix-shell

If you use nix-shell for your development environments then `haskell-language-server` will not find an installed GHC or will find a GHC with an installed package set different from what your project uses.

The simplest solution to this problem is to launch your editor from within the nix-shell environment:

```
$ nix-shell
```



```
[nix-shell] $ code .
```

However, launching a nix-shell every time you want to edit a file is somewhat tedious, so an alternative is to use direnv. There are [several solutions](#) that will propagate information from a nix-shell to a direnv envrc file. You can then use a direnv support plugin in your editor (emacs has one, vscode has one) to get the right environment for the server launch.

Yet another solution is to use a plugin that loads the nix-shell directly in the editor, such as [Nix Environment Selector](#) for VSCode.

## How to build a Haskell project using Stack

[Stack](#) is a popular build tool for Haskell projects. It has first-class support for Nix. Stack can optionally use Nix to automatically select the right version of GHC and other build tools to build, test and execute apps in an existing project downloaded from somewhere on the Internet. Pass the `-nix` flag to any `stack` command to do so, e.g.

```
git clone --recurse-submodules https://github.com/yesodweb/wai.git
cd wai
stack --nix build
```

If you want `stack` to use Nix by default, you can add a `nix` section to the `stack.yaml` file, as explained in the [Stack documentation](#). For example:

```
nix:
  enable: true
  packages: [pkgconfig zeromq zlib]
```

The example configuration snippet above tells Stack to create an ad hoc environment for `nix-shell` as in the below section, in which the `pkgconfig`, `zeromq` and `zlib` packages from Nixpkgs are available. All `stack` commands will implicitly be executed inside this ad hoc environment.

Some projects have more sophisticated needs. For examples, some ad hoc environments might need to expose Nixpkgs packages compiled in a certain way, or with extra environment variables. In these cases, you'll need a `shell` field instead of `packages`:

```
nix:
  enable: true
  shell-file: shell.nix
```

For more on how to write a `shell.nix` file see the below section. You'll need to express a derivation. Note that Nixpkgs ships with a convenience wrapper function around `mkDerivation` called `haskell.lib.buildStackProject` to help you create this derivation in exactly the way Stack expects. However for this to work you need to disable the sandbox, which you can do by using `--option sandbox relaxed` or `--option sandbox false` to the Nix command. All of the same inputs as `mkDerivation` can be provided. For example, to build a Stack project that including packages that link against a version of the R library compiled with special options turned on:

```
with (import <nixpkgs> { });
```

```
let R = pkgs.R.override { enableStrictBarrier = true; };
in
haskell.lib.buildStackProject {
  name = "HaskellR";
  buildInputs = [ R zeromq zlib ];
}
```

You can select a particular GHC version to compile with by setting the `ghc` attribute as an argument to `buildStackProject`. Better yet, let Stack choose what GHC version it wants based on the snapshot specified in `stack.yaml` (only works with Stack `>= 1.1.3`):

```
{nixpkgs ? import <nixpkgs> { }, ghc ? nixpkgs.ghc}:

with nixpkgs;

let R = pkgs.R.override { enableStrictBarrier = true; };
in
haskell.lib.buildStackProject {
  name = "HaskellR";
  buildInputs = [ R zeromq zlib ];
  inherit ghc;
}
```

## How to create ad hoc environments for `nix-shell`

The easiest way to create an ad hoc development environment is to run `nix-shell` with the appropriate GHC environment given on the command-line:

```
nix-shell -p "haskellPackages.ghcWithPackages (pkgs: with pkgs; [mtl pandoc])"
```

For more sophisticated use-cases, however, it's more convenient to save the desired configuration in a file called `shell.nix` that looks like this:

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "ghc7102" }:
let
  inherit (nixpkgs) pkgs;
  ghc = pkgs.haskell.packages.${compiler}.ghcWithPackages (ps: with ps; [
    monad-par mtl
  ]);
in
pkgs.stdenv.mkDerivation {
  name = "my-haskell-env-0";
  buildInputs = [ ghc ];
  shellHook = "eval $(egrep ^export ${compiler}/bin/ghc)";
}
```

Now run `nix-shell` — or even `nix-shell --pure` — to enter a shell environment that has the appropriate compiler in `$PATH`. If you use `--pure`, then add all other packages that your development environment needs into the `buildInputs` attribute. If you'd like to switch to a different compiler version, then pass an appropriate `compiler` argument to the expression, i.e. `nix-shell --`

`argstr compiler ghc784.`

If you need such an environment because you'd like to compile a Hackage package outside of Nix — i.e. because you're hacking on the latest version from Git —, then the package set provides suitable nix-shell environments for you already! Every Haskell package has an `env` attribute that provides a shell environment suitable for compiling that particular package. If you'd like to hack the `lens` library, for example, then you just have to check out the source code and enter the appropriate environment:

```
$ cabal get lens-4.11 && cd lens-4.11
Downloading lens-4.11...
Unpacking to lens-4.11/

$ nix-shell "<nixpkgs>" -A haskellPackages.lens.env
[nix-shell:/tmp/lens-4.11]$
```

At point, you can run `cabal configure`, `cabal build`, and all the other development commands. Note that you need `cabal-install` installed in your `$PATH` already to use it here — the `nix-shell` environment does not provide it.

## How to create Nix builds for your own private Haskell packages

If your own Haskell packages have build instructions for Cabal, then you can convert those automatically into build instructions for Nix using the `cabal2nix` utility, which you can install into your profile by running `nix-env -i cabal2nix`.

## How to build a stand-alone project

For example, let's assume that you're working on a private project called `foo`. To generate a Nix build expression for it, change into the project's top-level directory and run the command:

```
cabal2nix . > foo.nix
```

Then write the following snippet into a file called `default.nix`:

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "ghc7102" }:
nixpkgs.pkgs.haskell.packages.${compiler}.callPackage ./foo.nix { }
```

Finally, store the following code in a file called `shell.nix`:

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "ghc7102" }:
(import ./default.nix { inherit nixpkgs compiler; }).env
```

At this point, you can run `nix-build` to have Nix compile your project and install it into a Nix store path. The local directory will contain a symlink called `result` after `nix-build` returns that points into that location. Of course, passing the flag `--argstr compiler ghc763` allows switching the build

to any version of GHC currently supported.

Furthermore, you can call `nix-shell` to enter an interactive development environment in which you can use `cabal configure` and `cabal build` to develop your code. That environment will automatically contain a proper GHC derivation with all the required libraries registered as well as all the system-level libraries your package might need.

If your package does not depend on any system-level libraries, then it's sufficient to run

```
nix-shell --command "cabal configure"
```

once to set up your build. `cabal-install` determines the absolute paths to all resources required for the build and writes them into a config file in the `dist/` directory. Once that's done, you can run `cabal build` and any other command for that project even outside of the `nix-shell` environment. This feature is particularly nice for those of us who like to edit their code with an IDE, like Emacs' `haskell-mode`, because it's not necessary to start Emacs inside of `nix-shell` just to make it find out the necessary settings for building the project; `cabal-install` has already done that for us.

If you want to do some quick-and-dirty hacking and don't want to bother setting up a `default.nix` and `shell.nix` file manually, then you can use the `--shell` flag offered by `cabal2nix` to have it generate a stand-alone `nix-shell` environment for you. With that feature, running

```
cabal2nix --shell . > shell.nix
nix-shell --command "cabal configure"
```

is usually enough to set up a build environment for any given Haskell package. You can even use that generated file to run `nix-build`, too:

```
nix-build shell.nix
```

## How to build projects that depend on each other

If you have multiple private Haskell packages that depend on each other, then you'll have to register those packages in the Nixpkgs set to make them visible for the dependency resolution performed by `callPackage`. First of all, change into each of your projects top-level directories and generate a `default.nix` file with `cabal2nix`:

```
cd ~/src/foo && cabal2nix . > default.nix
cd ~/src/bar && cabal2nix . > default.nix
```

Then edit your `~/.config/nixpkgs/config.nix` file to register those builds in the default Haskell package set:

```
{
  packageOverrides = super:
  {
    haskellPackages = super.haskellPackages.override {
      overrides = self: super: {
        foo = self.callPackage ../src/foo {};
      };
    };
  };
}
```

```
        bar = self.callPackage ../src/bar {};  
    };  
};  
}
```

Once that's accomplished, `nix-env -f "<nixpkgs>" -qA haskellPackages` will show your packages like any other package from Hackage, and you can build them

```
nix-build "<nixpkgs>" -A haskellPackages.foo
```

or enter an interactive shell environment suitable for building them:

```
nix-shell "<nixpkgs>" -A haskellPackages.bar.env
```