Nix Package Manager Guide

Version 2.3.10

Eelco Dolstra

Part I. Introduction

Chapter 1. About Nix

Nix is a *purely functional package manager*. This means that it treats packages like values in purely functional programming languages such as Haskell -- they are built by functions that don't have side-effects, and they never change after they have been built. Nix stores packages in the *Nix store*, usually the directory `/nix/store`, where each package has its own unique subdirectory such as

```
/nix/store/b6gvzjyb2pg0kjfwrjmg1vfhh54ad73z-firefox-33.1/
```

where `b6gvzjyb2pg0...` is a unique identifier for the package that captures all its dependencies (it's a cryptographic hash of the package's build dependency graph). This enables many powerful features.

Multiple versions

You can have multiple versions or variants of a package installed at the same time. This is especially important when different applications have dependencies on different versions of the same package -- it prevents the "DLL hell". Because of the hashing scheme, different versions of a package end up in different paths in the Nix store, so they don't interfere with each other.

An important consequence is that operations like upgrading or uninstalling an application cannot break other applications, since these operations never "destructively" update or delete files that are used by other packages.

Complete dependencies

Nix helps you make sure that package dependency specifications are complete. In general, when you're making a package for a package management system like RPM, you have to specify for each package what its dependencies are, but there are no guarantees that this specification is complete. If you forget a dependency, then the package will build and work correctly on *your* machine if you have the dependency installed, but not on the end user's machine if it's not there.

Since Nix on the other hand doesn't install packages in "global" locations like `/usr/bin` but in package-specific directories, the risk of incomplete dependencies is greatly reduced. This is because tools such as compilers don't search in per-packages directories such as `/nix/store/5lbfaxb722zp...-openssl-0.9.8d/include`, so if a package builds correctly on your system, this is because you specified the dependency explicitly. This takes care of the build-time dependencies.

Once a package is built, runtime dependencies are found by scanning binaries for the hash parts of Nix store paths (such as `r8vvq9kq...`). This sounds risky, but it works extremely well.

Multi-user support

Nix has multi-user support. This means that non-privileged users can securely install software. Each user can have a different *profile*, a set of packages in the Nix store that appear in the user's `PATH`. If a user installs a package that another user has already installed previously, the package won't be built or downloaded a second time. At the same time, it is not possible for one user to inject a Trojan horse into a package that might be used by another user.

Atomic upgrades and rollbacks

Since package management operations never overwrite packages in the Nix store but just add new versions in different paths, they are *atomic*. So during a package upgrade, there is no time window in which the package has some files from the old version and some files from the new version -- which would be bad because a program might well crash if it's started during that period.

And since packages aren't overwritten, the old versions are still there after an upgrade. This means that you can *roll back* to the old version:

```
$ nix-env --upgrade some-packages
$ nix-env --rollback
```

Garbage collection

When you uninstall a package like this...

```
$ nix-env --uninstall firefox
```

the package isn't deleted from the system right away (after all, you might want to do a rollback, or it might be in the profiles of other users). Instead, unused packages can be deleted safely by running the *garbage collector*:

```
$ nix-collect-garbage
```

This deletes all packages that aren't in use by any user profile or by a currently running program.

Functional package language

Packages are built from *Nix expressions*, which is a simple functional language. A Nix expression describes everything that goes into a package build action (a "derivation"): other packages, sources, the build script, environment variables for the build script, etc. Nix tries very hard to ensure that Nix expressions are *deterministic*: building a Nix expression twice should yield the same result.

Because it's a functional language, it's easy to support building variants of a package: turn the Nix expression into a function and call it any number of times with the appropriate arguments. Due to the hashing scheme, variants don't conflict with each other in the Nix store.

Transparent source/binary deployment

Nix expressions generally describe how to build a package from source, so an installation action like

```
$ nix-env --install firefox
```

could cause quite a bit of build activity, as not only Firefox but also all its dependencies (all the way up to the C library and the compiler) would have to built, at least if they are not already in the Nix store. This is a *source deployment model*. For most users, building from source is not very pleasant as it takes far too long. However, Nix can automatically skip building from source and instead use a *binary cache*, a web server that provides pre-built binaries. For instance, when asked to build `/nix/store/b6gvzjyb2pg0...-firefox-33.1` from source, Nix would first check if the file `https://cache.nixos.org/b6gvzjyb2pg0....narinfo` exists, and if so, fetch the pre-built binary referenced from there; otherwise, it would fall back to building from source.

Nix Packages collection

We provide a large set of Nix expressions containing hundreds of existing Unix packages, the *Nix Packages collection* (Nixpkgs).

Managing build environments

Nix is extremely useful for developers as it makes it easy to automatically set up the build environment for a package. Given a Nix expression that describes the dependencies of your package, the command **nix-shell** will build or download those dependencies if they're not already in your Nix store, and then start a Bash shell in which all necessary environment variables (such as compiler search paths) are set.

For example, the following command gets all dependencies of the Pan newsreader, as described by its Nix expression:

```
$ nix-shell '<nixpkgs>' -A pan
```

You're then dropped into a shell where you can edit, build and test the package:

```
[nix-shell]$ tar xf $src
[nix-shell]$ cd pan-*
[nix-shell]$ ./configure
[nix-shell]$ make
[nix-shell]$ ./pan/gui/pan
```

Portability

Nix runs on Linux and macOS.

NixOS

NixOS is a Linux distribution based on Nix. It uses Nix not just for package management but also to manage the system configuration (e.g., to build configuration files in `/etc`). This means, among other things, that it is easy to roll back the entire configuration of the system to an earlier state. Also, users can install software without root privileges. For more information and downloads, see the NixOS homepage.

License

Nix is released under the terms of the GNU LGPLv2.1 or (at your option) any later version.

Chapter 2. Quick Start

This chapter is for impatient people who don't like reading documentation. For more in-depth information you are kindly referred to subsequent chapters.

1. Install single-user Nix by running the following:

   ```
   $ bash <(curl -L https://nixos.org/nix/install)
   ```

   This will install Nix in `/nix`. The install script will create `/nix` using **sudo**, so make sure you have sufficient rights. (For other installation methods, see Part II, "Installation".)

2. See what installable packages are currently available in the channel:

```
$ nix-env -qa
docbook-xml-4.3
docbook-xml-4.5
firefox-33.0.2
hello-2.9
libxslt-1.1.28
...
```

3. Install some packages from the channel:

```
$ nix-env -i hello
```

   This should download pre-built packages; it should not build them locally (if it does, something went wrong).

4. Test that they work:

```
$ which hello
/home/eelco/.nix-profile/bin/hello
$ hello
Hello, world!
```

5. Uninstall a package:

```
$ nix-env -e hello
```

6. You can also test a package without installing it:

```
$ nix-shell -p hello
```

   This builds or downloads GNU Hello and its dependencies, then drops you into a Bash shell where the **hello** command is present, all without affecting your normal environment:

```
[nix-shell:~]$ hello
Hello, world!

[nix-shell:~]$ exit

$ hello
hello: command not found
```

7. To keep up-to-date with the channel, do:

```
$ nix-channel --update nixpkgs
$ nix-env -u '*'
```

   The latter command will upgrade each installed package for which there is a "newer" version (as determined by comparing the version numbers).

8. If you're unhappy with the result of a **nix-env** action (e.g., an upgraded package turned out not to work properly), you can go back:

```
$ nix-env --rollback
```

9. You should periodically run the Nix garbage collector to get rid of unused packages, since uninstalls or upgrades don't actually delete them:

```
$ nix-collect-garbage -d
```

Part II. Installation

This section describes how to install and configure Nix for first-time use.

Chapter 3. Supported Platforms

Nix is currently supported on the following platforms:

- Linux (i686, x86_64, aarch64).
- macOS (x86_64).

Chapter 4. Installing a Binary Distribution

If you are using Linux or macOS versions up to 10.14 (Mojave), the easiest way to install Nix is to run the following command:

```
$ sh <(curl -L https://nixos.org/nix/install)
```

If you're using macOS 10.15 (Catalina) or newer, consult the macOS installation instructions before installing.

As of Nix 2.1.0, the Nix installer will always default to creating a single-user installation, however opting in to the multi-user installation is highly recommended.

4.1. Single User Installation

To explicitly select a single-user installation on your system:

```
sh <(curl -L https://nixos.org/nix/install) --no-daemon
```

This will perform a single-user installation of Nix, meaning that `/nix` is owned by the invoking user. You should run this under your usual user account, *not* as root. The script will invoke **sudo** to create `/nix` if it doesn't already exist. If you don't have **sudo**, you should manually create `/nix` first as root, e.g.:

```
$ mkdir /nix
$ chown alice /nix
```

The install script will modify the first writable file from amongst `.bash_profile`, `.bash_login` and `.profile` to source `~/.nix-profile/etc/profile.d/nix.sh`. You can set the `NIX_INSTALLER_NO_MODIFY_PROFILE` environment variable before executing the install script to disable this behaviour.

You can uninstall Nix simply by running:

```
$ rm -rf /nix
```

4.2. Multi User Installation

The multi-user Nix installation creates system users, and a system service for the Nix daemon.

Supported Systems

- Linux running systemd, with SELinux disabled
- macOS

You can instruct the installer to perform a multi-user installation on your system:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
```

The multi-user installation of Nix will create build users between the user IDs 30001 and 30032, and a group with the group ID 30000. You should run this under your usual user account, *not* as root. The script will invoke **sudo** as needed.

Note

If you need Nix to use a different group ID or user ID set, you will have to download the tarball manually and edit the install script.

The installer will modify `/etc/bashrc`, and `/etc/zshrc` if they exist. The installer will first back up these files with a `.backup-before-nix` extension. The installer will also create `/etc/profile.d/nix.sh`.

You can uninstall Nix with the following commands:

```
sudo rm -rf /etc/profile/nix.sh /etc/nix /nix ~root/.nix-profile
   ↪ ~root/.nix-defexpr ~root/.nix-channels ~/.nix-profile ~/.nix-defexpr
   ↪ ~/.nix-channels

# If you are on Linux with systemd, you will need to run:
sudo systemctl stop nix-daemon.socket
sudo systemctl stop nix-daemon.service
sudo systemctl disable nix-daemon.socket
sudo systemctl disable nix-daemon.service
sudo systemctl daemon-reload

# If you are on macOS, you will need to run:
sudo launchctl unload /Library/LaunchDaemons/org.nixos.nix-daemon.plist
sudo rm /Library/LaunchDaemons/org.nixos.nix-daemon.plist
```

There may also be references to Nix in `/etc/profile`, `/etc/bashrc`, and `/etc/zshrc` which you may remove.

4.3. macOS Installation

Starting with macOS 10.15 (Catalina), the root filesystem is read-only. This means `/nix` can no longer live on your system volume, and that you'll need a workaround to install Nix.

The recommended approach, which creates an unencrypted APFS volume for your Nix store and a "synthetic" empty directory to mount it over at `/nix`, is least likely to impair Nix or your system.

Note

With all separate-volume approaches, it's possible something on your system (particularly daemons/services and restored apps) may need access to your Nix store before the volume is mounted. Adding additional encryption makes this more likely.

If you're using a recent Mac with a T2 chip, your drive will still be encrypted at rest (in which case "unencrypted" is a bit of a misnomer). To use this approach, just install Nix with:

```
$ sh <(curl -L https://nixos.org/nix/install)
   ↪ --darwin-use-unencrypted-nix-store-volume
```

If you don't like the sound of this, you'll want to weigh the other approaches and tradeoffs detailed in this section.

Eventual solutions?

All of the known workarounds have drawbacks, but we hope better solutions will be available in the future. Some that we have our eye on are:

1. A true firmlink would enable the Nix store to live on the primary data volume without the build problems caused by the symlink approach. End users cannot currently create true firmlinks.
2. If the Nix store volume shared FileVault encryption with the primary data volume (probably by using the same volume group and role), FileVault encryption could be easily supported by the installer without requiring manual setup by each user.

4.3.1. Change the Nix store path prefix

Changing the default prefix for the Nix store is a simple approach which enables you to leave it on your root volume, where it can take full advantage of FileVault encryption if enabled. Unfortunately, this approach also opts your device out of some benefits that are enabled by using the same prefix across systems:

- Your system won't be able to take advantage of the binary cache (unless someone is able to stand up and support duplicate caching infrastructure), which means you'll spend more time waiting for builds.
- It's harder to build and deploy packages to Linux systems.

It would also possible (and often requested) to just apply this change ecosystem-wide, but it's an intrusive process that has side effects we want to avoid for now.

4.3.2. Use a separate encrypted volume

If you like, you can also add encryption to the recommended approach taken by the installer. You can do this by pre-creating an encrypted volume before you run the installer--or you can run the installer and encrypt the volume it creates later.

In either case, adding encryption to a second volume isn't quite as simple as enabling FileVault for your boot volume. Before you dive in, there are a few things to weigh:

1. The additional volume won't be encrypted with your existing FileVault key, so you'll need another mechanism to decrypt the volume.
2. You can store the password in Keychain to automatically decrypt the volume on boot--but it'll have to wait on Keychain and may not mount before your GUI apps restore. If any of your launchd agents or apps depend on Nix-installed software (for example, if you use a Nix-installed login shell), the restore may fail or break.

   On a case-by-case basis, you may be able to work around this problem by using **wait4path** to block execution until your executable is available.

   It's also possible to decrypt and mount the volume earlier with a login hook--but this mechanism appears to be deprecated and its future is unclear.
3. You can hard-code the password in the clear, so that your store volume can be decrypted before Keychain is available.

If you are comfortable navigating these tradeoffs, you can encrypt the volume with something along the lines of:

```
alice$ diskutil apfs enableFileVault /nix -user disk
```

4.3.3. Symlink the Nix store to a custom location

Another simple approach is using `/etc/synthetic.conf` to symlink the Nix store to the data volume. This option also enables your store to share any configured FileVault encryption. Unfortunately, builds that resolve the symlink may leak the canonical path or even fail.

Because of these downsides, we can't recommend this approach.

4.3.4. Notes on the recommended approach

This section goes into a little more detail on the recommended approach. You don't need to understand it to run the installer, but it can serve as a helpful reference if you run into trouble.

1. In order to compose user-writable locations into the new read-only system root, Apple introduced a new concept called `firmlinks`, which it describes as a "bi-directional wormhole" between two filesystems. You can see the current firmlinks in `/usr/share/firmlinks`. Unfortunately, firmlinks aren't (currently?) user-configurable.

   For special cases like NFS mount points or package manager roots, synthetic.conf(5) supports limited user-controlled file-creation (of symlinks, and synthetic empty directories) at `/`. To create a synthetic empty directory for mounting at `/nix`, add the following line to `/etc/synthetic.conf` (create it if necessary):

   ```
   nix
   ```

2. This configuration is applied at boot time, but you can use **apfs.util** to trigger creation (not deletion) of new entries without a reboot:

   ```
   alice$ /System/Library/Filesystems/apfs.fs/Contents/Resources/apfs.util -B
   ```

3. Create the new APFS volume with diskutil:

```
alice$ sudo diskutil apfs addVolume diskX APFS 'Nix Store' -mountpoint /nix
```

4. Using **vifs**, add the new mount to `/etc/fstab`. If it doesn't already have other entries, it should look something like:

```
#
# Warning - this file should only be modified with vifs(8)
#
# Failure to do so is unsupported and may be destructive.
#
LABEL=Nix\040Store /nix apfs rw,nobrowse
```

   The nobrowse setting will keep Spotlight from indexing this volume, and keep it from showing up on your desktop.

4.4. Installing a pinned Nix version from a URL

NixOS.org hosts version-specific installation URLs for all Nix versions since 1.11.16, at `https://releases.nixos.org/nix/nix-v`

These install scripts can be used the same as the main NixOS.org installation script:

```
sh <(curl -L https://nixos.org/nix/install)
```

In the same directory of the install script are sha256 sums, and gpg signature files.

4.5. Installing from a binary tarball

You can also download a binary tarball that contains Nix and all its dependencies. (This is what the install script at `https://nixos.org/nix/install` does automatically.) You should unpack it somewhere (e.g. in `/tmp`), and then run the script named **install** inside the binary tarball:

```
alice$ cd /tmp
alice$ tar xfj nix-1.8-x86_64-darwin.tar.bz2
alice$ cd nix-1.8-x86_64-darwin
alice$ ./install
```

If you need to edit the multi-user installation script to use different group ID or a different user ID range, modify the variables set in the file named `install-multi-user`.

Chapter 5. Installing Nix from Source

If no binary package is available, you can download and compile a source distribution.

5.1. Prerequisites

- GNU Make.
- Bash Shell. The `./configure` script relies on bashisms, so Bash is required.
- A version of GCC or Clang that supports C++17.
- **pkg-config** to locate dependencies. If your distribution does not provide it, you can get it from http://www.freedesktop.org/wiki/Software/pkg-config.
- The OpenSSL library to calculate cryptographic hashes. If your distribution does not provide it, you can get it from https://www.openssl.org.
- The `libbrotlienc` and `libbrotlidec` libraries to provide implementation of the Brotli compression algorithm. They are available for download from the official repository https://github.com/google/brotli.

- The bzip2 compressor program and the `libbz2` library. Thus you must have bzip2 installed, including development headers and libraries. If your distribution does not provide these, you can obtain bzip2 from https://web.archive.org/web/20180624184756/http://www.bzip.org/.
- `liblzma`, which is provided by XZ Utils. If your distribution does not provide this, you can get it from https://tukaani.org/xz/.
- cURL and its library. If your distribution does not provide it, you can get it from https://curl.haxx.se/.
- The SQLite embedded database library, version 3.6.19 or higher. If your distribution does not provide it, please install it from http://www.sqlite.org/.
- The Boehm garbage collector to reduce the evaluator's memory consumption (optional). To enable it, install `pkgconfig` and the Boehm garbage collector, and pass the flag `--enable-gc` to **configure**.
- The `boost` library of version 1.66.0 or higher. It can be obtained from the official web site https://www.boost.org/.
- The `editline` library of version 1.14.0 or higher. It can be obtained from the its repository https://github.com/troglobit/editline.
- The **xmllint** and **xsltproc** programs to build this manual and the man-pages. These are part of the `libxml2` and `libxslt` packages, respectively. You also need the DocBook XSL stylesheets and optionally the DocBook 5.0 RELAX NG schemas. Note that these are only required if you modify the manual sources or when you are building from the Git repository.
- Recent versions of Bison and Flex to build the parser. (This is because Nix needs GLR support in Bison and reentrancy support in Flex.) For Bison, you need version 2.6, which can be obtained from the GNU FTP server. For Flex, you need version 2.5.35, which is available on SourceForge. Slightly older versions may also work, but ancient versions like the ubiquitous 2.5.4a won't. Note that these are only required if you modify the parser or when you are building from the Git repository.
- The `libseccomp` is used to provide syscall filtering on Linux. This is an optional dependency and can be disabled passing a `--disable-seccomp-sandboxing` option to the **configure** script (Not recommended unless your system doesn't support `libseccomp`). To get the library, visit https://github.com/seccomp/libseccomp.

5.2. Obtaining a Source Distribution

The source tarball of the most recent stable release can be downloaded from the Nix homepage. You can also grab the most recent development release.

Alternatively, the most recent sources of Nix can be obtained from its Git repository. For example, the following command will check out the latest revision into a directory called `nix`:

```
$ git clone https://github.com/NixOS/nix
```

Likewise, specific releases can be obtained from the tags of the repository.

5.3. Building Nix from Source

After unpacking or checking out the Nix sources, issue the following commands:

```
$ ./configure options...
$ make
$ make install
```

Nix requires GNU Make so you may need to invoke **gmake** instead.

When building from the Git repository, these should be preceded by the command:

```
$ ./bootstrap.sh
```

The installation path can be specified by passing the `--prefix=prefix` to **configure**. The default installation directory is `/usr/local`. You can change this to any location you like. You must have write permission to the *`prefix`* path.

Nix keeps its *store* (the place where packages are stored) in `/nix/store` by default. This can be changed using `--with-store-dir=path`.

Warning

It is best *not* to change the Nix store from its default, since doing so makes it impossible to use pre-built binaries from the standard Nixpkgs channels -- that is, all packages will need to be built from source.

Nix keeps state (such as its database and log files) in `/nix/var` by default. This can be changed using `--localstatedir=path`.

Chapter 6. Security

Nix has two basic security models. First, it can be used in "single-user mode", which is similar to what most other package management tools do: there is a single user (typically `root`) who performs all package management operations. All other users can then use the installed packages, but they cannot perform package management operations themselves.

Alternatively, you can configure Nix in "multi-user mode". In this model, all users can perform package management operations -- for instance, every user can install software without requiring root privileges. Nix ensures that this is secure. For instance, it's not possible for one user to overwrite a package used by another user with a Trojan horse.

6.1. Single-User Mode

In single-user mode, all Nix operations that access the database in `prefix`/var/nix/db or modify the Nix store in `prefix`/store must be performed under the user ID that owns those directories. This is typically `root`. (If you install from RPM packages, that's in fact the default ownership.) However, on single-user machines, it is often convenient to **chown** those directories to your normal user account so that you don't have to **su** to `root` all the time.

6.2. Multi-User Mode

To allow a Nix store to be shared safely among multiple users, it is important that users are not able to run builders that modify the Nix store or database in arbitrary ways, or that interfere with builds started by other users. If they could do so, they could install a Trojan horse in some package and compromise the accounts of other users.

To prevent this, the Nix store and database are owned by some privileged user (usually `root`) and builders are executed under special user accounts (usually named `nixbld1`, `nixbld2`, etc.). When a unprivileged user runs a Nix command, actions that operate on the Nix store (such as builds) are forwarded to a *Nix daemon* running under the owner of the Nix store/database that performs the operation.

Note

Multi-user mode has one important limitation: only `root` and a set of trusted users specified in `nix.conf` can specify arbitrary binary caches. So while unprivileged users may install packages from arbitrary Nix expressions, they may not get pre-built binaries.

Setting up the build users

The *build users* are the special UIDs under which builds are performed. They should all be members of the *build users group* `nixbld`. This group should have no other members. The build users should not be members of any other group. On Linux, you can create the group and users as follows:

```
$ groupadd -r nixbld
$ for n in $(seq 1 10); do useradd -c "Nix build user $n" \
    -d /var/empty -g nixbld -G nixbld -M -N -r -s "$(which nologin)" \
    nixbld$n; done
```

This creates 10 build users. There can never be more concurrent builds than the number of build users, so you may want to increase this if you expect to do many builds at the same time.

Running the daemon

The Nix daemon should be started as follows (as `root`):

```
$ nix-daemon
```

You'll want to put that line somewhere in your system's boot scripts.

To let unprivileged users use the daemon, they should set the `NIX_REMOTE` environment variable to `daemon`. So you should put a line like

```
export NIX_REMOTE=daemon
```

into the users' login scripts.

Restricting access

To limit which users can perform Nix operations, you can use the permissions on the directory `/nix/var/nix/daemon-socket`. For instance, if you want to restrict the use of Nix to the members of a group called `nix-users`, do

```
$ chgrp nix-users /nix/var/nix/daemon-socket
$ chmod ug=rwx,o= /nix/var/nix/daemon-socket
```

This way, users who are not in the `nix-users` group cannot connect to the Unix domain socket `/nix/var/nix/daemon-socket/so` so they cannot perform Nix operations.

Chapter 7. Environment Variables

To use Nix, some environment variables should be set. In particular, `PATH` should contain the directories `prefix`/bin and `~/.nix-profile/bin`. The first directory contains the Nix tools themselves, while `~/.nix-profile` is a symbolic link to the current *user environment* (an automatically generated package consisting of symlinks to installed packages). The simplest way to set the required environment variables is to include the file `prefix`/etc/profile.d/nix.sh in your `~/.profile` (or similar), like this:

```
source prefix/etc/profile.d/nix.sh
```

7.1. NIX_SSL_CERT_FILE

If you need to specify a custom certificate bundle to account for an HTTPS-intercepting man in the middle proxy, you must specify the path to the certificate bundle in the environment variable `NIX_SSL_CERT_FILE`.

If you don't specify a `NIX_SSL_CERT_FILE` manually, Nix will install and use its own certificate bundle.

1. Set the environment variable and install Nix

   ```
   $ export NIX_SSL_CERT_FILE=/etc/ssl/my-certificate-bundle.crt
   $ sh <(curl -L https://nixos.org/nix/install)
   ```

2. In the shell profile and rc files (for example, `/etc/bashrc`, `/etc/zshrc`), add the following line:

   ```
   export NIX_SSL_CERT_FILE=/etc/ssl/my-certificate-bundle.crt
   ```

Note

You must not add the export and then do the install, as the Nix installer will detect the presense of Nix configuration, and abort.

7.1.1. `NIX_SSL_CERT_FILE` with macOS and the Nix daemon

On macOS you must specify the environment variable for the Nix daemon service, then restart it:

```
$ sudo launchctl setenv NIX_SSL_CERT_FILE /etc/ssl/my-certificate-bundle.crt
$ sudo launchctl kickstart -k system/org.nixos.nix-daemon
```

7.1.2. Proxy Environment Variables

The Nix installer has special handling for these proxy-related environment variables: `http_proxy`, `https_proxy`, `ftp_proxy`, `no_proxy`, `HTTP_PROXY`, `HTTPS_PROXY`, `FTP_PROXY`, `NO_PROXY`.

If any of these variables are set when running the Nix installer, then the installer will create an override file at `/etc/systemd/system/nix-daemon.service.d/override.conf` so **nix-daemon** will use them.

Chapter 8. Upgrading Nix

Multi-user Nix users on macOS can upgrade Nix by running: **sudo -i sh -c 'nix-channel --update && nix-env -iA nixpkgs.nix && launchctl remove org.nixos.nix-daemon && launchctl load /Library/LaunchDaemons/org.nixos.nix-daemon.plist'**

Single-user installations of Nix should run this: **nix-channel --update; nix-env -iA nixpkgs.nix**

Part III. Package Management

This chapter discusses how to do package management with Nix, i.e., how to obtain, install, upgrade, and erase packages. This is the "user's" perspective of the Nix system -- people who want to *create* packages should consult Part IV, "Writing Nix Expressions".

Chapter 9. Basic Package Management

The main command for package management is **nix-env**. You can use it to install, upgrade, and erase packages, and to query what packages are installed or are available for installation.

In Nix, different users can have different "views" on the set of installed applications. That is, there might be lots of applications present on the system (possibly in many different versions), but users can have a specific selection of those active — where "active" just means that it appears in a directory in the user's `PATH`. Such a view on the set of installed applications is called a *user environment*, which is just a directory tree consisting of symlinks to the files of the active applications.

Components are installed from a set of *Nix expressions* that tell Nix how to build those packages, including, if necessary, their dependencies. There is a collection of Nix expressions called the Nixpkgs package collection that contains packages ranging from basic development stuff such as GCC and Glibc, to end-user applications like Mozilla Firefox. (Nix is however not tied to the Nixpkgs package collection; you could write your own Nix expressions based on Nixpkgs, or completely new ones.)

You can manually download the latest version of Nixpkgs from http://nixos.org/nixpkgs/download.html. However, it's much more convenient to use the Nixpkgs *channel*, since it makes it easy to stay up to date with new versions of Nixpkgs. (Channels are described in more detail in Chapter 12, *Channels*.) Nixpkgs is automatically added to your list of "subscribed" channels when you install Nix. If this is not the case for some reason, you can add it as follows:

```
$ nix-channel --add https://nixos.org/channels/nixpkgs-unstable
$ nix-channel --update
```

Note

On NixOS, you're automatically subscribed to a NixOS channel corresponding to your NixOS major release (e.g. `http://nixos.org/channels/nixos-14.12`). A NixOS channel is identical to the Nixpkgs channel, except that it contains only Linux binaries and is updated only if a set of regression tests succeed.

You can view the set of available packages in Nixpkgs:

```
$ nix-env -qa
aterm-2.2
bash-3.0
binutils-2.15
bison-1.875d
blackdown-1.4.2
bzip2-1.0.2
...
```

The flag `-q` specifies a query operation, and `-a` means that you want to show the "available" (i.e., installable) packages, as opposed to the installed packages. If you downloaded Nixpkgs yourself, or if you checked it out from GitHub, then you need to pass the path to your Nixpkgs tree using the `-f` flag:

```
$ nix-env -qaf /path/to/nixpkgs
```

where */path/to/nixpkgs* is where you've unpacked or checked out Nixpkgs.

You can select specific packages by name:

```
$ nix-env -qa firefox
firefox-34.0.5
firefox-with-plugins-34.0.5
```

and using regular expressions:

```
$ nix-env -qa 'firefox.*'
```

It is also possible to see the *status* of available packages, i.e., whether they are installed into the user environment and/or present in the system:

```
$ nix-env -qas
...
-PS bash-3.0
--S binutils-2.15
IPS bison-1.875d
...
```

The first character (`I`) indicates whether the package is installed in your current user environment. The second (`P`) indicates whether it is present on your system (in which case installing it into your user environment would be a very quick operation). The last one (`S`) indicates whether there is a so-called *substitute* for the package, which is Nix's mechanism for doing binary deployment. It just means that Nix knows that it can fetch a pre-built package from somewhere (typically a network server) instead of building it locally.

You can install a package using `nix-env -i`. For instance,

```
$ nix-env -i subversion
```

will install the package called `subversion` (which is, of course, the Subversion version management system).

Note

When you ask Nix to install a package, it will first try to get it in pre-compiled form from a *binary cache*. By default, Nix will use the binary cache `https://cache.nixos.org`; it contains binaries for most packages in Nixpkgs. Only if no binary is available in the binary cache, Nix will build the package from source. So if `nix-env -i subversion` results in Nix building stuff from source, then either the package is not built for your platform by the Nixpkgs build servers, or your version of Nixpkgs is too old or too new. For instance, if you have a very recent checkout of Nixpkgs, then the Nixpkgs build servers may not have had a chance to build everything and upload the resulting binaries to `https://cache.nixos.org`. The Nixpkgs channel is only updated after all binaries have been uploaded to the cache, so if you stick to the Nixpkgs channel (rather than using a Git checkout of the Nixpkgs tree), you will get binaries for most packages.

Naturally, packages can also be uninstalled:

```
$ nix-env -e subversion
```

Upgrading to a new version is just as easy. If you have a new release of Nix Packages, you can do:

```
$ nix-env -u subversion
```

This will *only* upgrade Subversion if there is a "newer" version in the new set of Nix expressions, as defined by some pretty arbitrary rules regarding ordering of version numbers (which generally do what you'd expect of them). To just unconditionally replace Subversion with whatever version is in the Nix expressions, use `-i` instead of `-u`; `-i` will remove whatever version is already installed.

You can also upgrade all packages for which there are newer versions:

```
$ nix-env -u
```

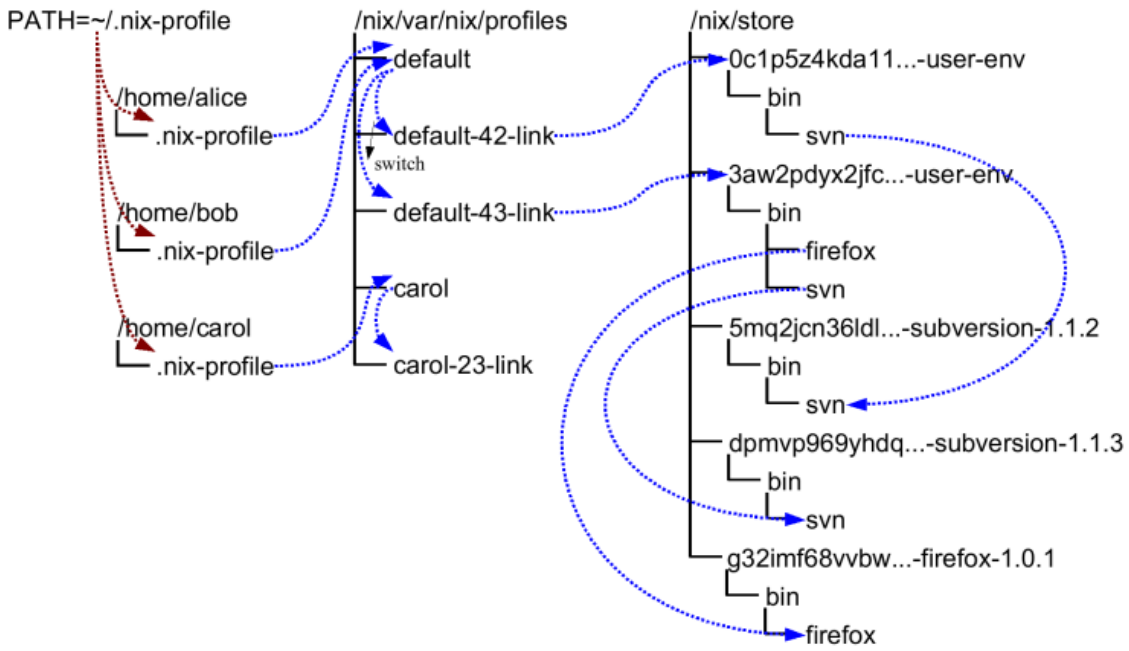
Sometimes it's useful to be able to ask what **nix-env** would do, without actually doing it. For instance, to find out what packages would be upgraded by `nix-env -u`, you can do

```
$ nix-env -u --dry-run
(dry run; not doing anything)
upgrading `libxslt-1.1.0' to `libxslt-1.1.10'
upgrading `graphviz-1.10' to `graphviz-1.12'
upgrading `coreutils-5.0' to `coreutils-5.2.1'
```

Chapter 10. Profiles

Profiles and user environments are Nix's mechanism for implementing the ability to allow different users to have different configurations, and to do atomic upgrades and rollbacks. To understand how they work, it's useful to know a bit about how Nix works. In Nix, packages are stored in unique locations in the *Nix store* (typically, `/nix/store`). For instance, a particular version of the Subversion package might be stored in a directory `/nix/store/dpmvp969yhdqs7lm2r1a3gng7pyq6vy4-subversion-1.1.3/`, while another version might be stored in `/nix/store/5mq2jcn36ldlmh93yj1n8s9c95pj7c5s-subversion-1.1.2`. The long strings prefixed to the directory names are cryptographic hashes[1] of *all* inputs involved in building the package — sources, dependencies, compiler flags, and so on. So if two packages differ in any way, they end up in different locations in the file system, so they don't interfere with each other. Figure 10.1, "User environments" shows a part of a typical Nix store.

Figure 10.1. User environments



Of course, you wouldn't want to type

```
$ /nix/store/dpmvp969yhdq...-subversion-1.1.3/bin/svn
```

every time you want to run Subversion. Of course we could set up the `PATH` environment variable to include the `bin` directory of every package we want to use, but this is not very convenient since changing `PATH` doesn't take effect for already existing processes. The solution Nix uses is to create directory trees of symlinks to *activated* packages. These are called *user environments* and they are packages themselves (though automatically generated by **nix-env**), so they too reside in the Nix store. For instance, in Figure 10.1, "User environments" the user environment `/nix/store/0c1p5z4kda11...-user-env` contains a symlink to just Subversion 1.1.2 (arrows in the figure indicate symlinks). This would be what we would obtain if we had done

```
$ nix-env -i subversion
```

on a set of Nix expressions that contained Subversion 1.1.2.

This doesn't in itself solve the problem, of course; you wouldn't want to type `/nix/store/0c1p5z4kda11...-user-env/bin/svn` either. That's why there are symlinks outside of the store that point to the user environments in the store; for instance, the symlinks `default-42-link` and `default-43-link` in the example. These are called *generations* since every time you perform a **nix-env** operation, a new user environment is generated based on the current one. For instance, generation 43 was created from generation 42 when we did

```
$ nix-env -i subversion firefox
```

on a set of Nix expressions that contained Firefox and a new version of Subversion.

Generations are grouped together into *profiles* so that different users don't interfere with each other if they don't want to. For example:

```
$ ls -l /nix/var/nix/profiles/
...
lrwxrwxrwx  1 eelco ... default-42-link -> /nix/store/0c1p5z4kda11...-user-env
lrwxrwxrwx  1 eelco ... default-43-link -> /nix/store/3aw2pdyx2jfc...-user-env
lrwxrwxrwx  1 eelco ... default -> default-43-link
```

This shows a profile called `default`. The file `default` itself is actually a symlink that points to the current generation. When we do a **nix-env** operation, a new user environment and generation link are created based on the current one, and finally the `default` symlink is made to point at the new generation. This last step is atomic on Unix, which explains how we can do atomic upgrades. (Note that the building/installing of new packages doesn't interfere in any way with old packages, since they are stored in different locations in the Nix store.)

If you find that you want to undo a **nix-env** operation, you can just do

```
$ nix-env --rollback
```

which will just make the current generation link point at the previous link. E.g., `default` would be made to point at `default-42-link`. You can also switch to a specific generation:

```
$ nix-env --switch-generation 43
```

which in this example would roll forward to generation 43 again. You can also see all available generations:

```
$ nix-env --list-generations
```

You generally wouldn't have `/nix/var/nix/profiles/some-profile`/bin in your `PATH`. Rather, there is a symlink `~/.nix-profile` that points to your current profile. This means that you should put `~/.nix-profile/bin` in your `PATH` (and indeed, that's what the initialisation script `/nix/etc/profile.d/nix.sh` does). This makes it easier to switch to a different profile. You can do that using the command **nix-env --switch-profile**:

```
$ nix-env --switch-profile /nix/var/nix/profiles/my-profile
2
$ nix-env --switch-profile /nix/var/nix/profiles/default
```

These commands switch to the `my-profile` and default profile, respectively. If the profile doesn't exist, it will be created automatically. You should be careful about storing a profile in another location than the `profiles` directory, since otherwise it might not be used as a root of the garbage collector (see Chapter 11, *Garbage Collection*).

All **nix-env** operations work on the profile pointed to by ~**/.nix-profile**, but you can override this using the `--profile` option (abbreviation `-p`):

```
$ nix-env -p /nix/var/nix/profiles/other-profile -i subversion
```

This will *not* change the ~**/.nix-profile** symlink.

---

[1] 160-bit truncations of SHA-256 hashes encoded in a base-32 notation, to be precise.

Chapter 11. Garbage Collection

nix-env operations such as upgrades (`-u`) and uninstall (`-e`) never actually delete packages from the system. All they do (as shown above) is to create a new user environment that no longer contains symlinks to the "deleted" packages.

Of course, since disk space is not infinite, unused packages should be removed at some point. You can do this by running the Nix garbage collector. It will remove from the Nix store any package not used (directly or indirectly) by any generation of any profile.

Note however that as long as old generations reference a package, it will not be deleted. After all, we wouldn't be able to do a rollback otherwise. So in order for garbage collection to be effective, you should also delete (some) old generations. Of course, this should only be done if you are certain that you will not need to roll back.

To delete all old (non-current) generations of your current profile:

```
$ nix-env --delete-generations old
```

Instead of `old` you can also specify a list of generations, e.g.,

```
$ nix-env --delete-generations 10 11 14
```

To delete all generations older than a specified number of days (except the current generation), use the `d` suffix. For example,

```
$ nix-env --delete-generations 14d
```

deletes all generations older than two weeks.

After removing appropriate old generations you can run the garbage collector as follows:

```
1 $ nix-store --gc
```

The behaviour of the gargage collector is affected by the `keep-derivations` (default: true) and `keep-outputs` (default: false) options in the Nix configuration file. The defaults will ensure that all derivations that are build-time dependencies of garbage collector roots will be kept and that all output paths that are runtime dependencies will be kept as well. All other derivations or paths will be collected. (This is usually what you want, but while you are developing it may make sense to keep outputs to ensure that rebuild times are quick.) If you are feeling uncertain, you can also first view what files would be deleted:

```
$ nix-store --gc --print-dead
```

Likewise, the option `--print-live` will show the paths that *won't* be deleted.

There is also a convenient little utility **nix-collect-garbage**, which when invoked with the `-d` (`--delete-old`) switch deletes all old generations of all profiles in `/nix/var/nix/profiles`. So

```
$ nix-collect-garbage -d
```

is a quick and easy way to clean up your system.

11.1. Garbage Collector Roots

The roots of the garbage collector are all store paths to which there are symlinks in the directory `prefix`/nix/var/nix/gcroots. For instance, the following command makes the path `/nix/store/d718ef...-foo` a root of the collector:

```
$ ln -s /nix/store/d718ef...-foo /nix/var/nix/gcroots/bar
```

That is, after this command, the garbage collector will not remove `/nix/store/d718ef...-foo` or any of its dependencies.

Subdirectories of `prefix`/nix/var/nix/gcroots are also searched for symlinks. Symlinks to non-store paths are followed and searched for roots, but symlinks to non-store paths *inside* the paths reached in that way are not followed to prevent infinite recursion.

Chapter 12. Channels

If you want to stay up to date with a set of packages, it's not very convenient to manually download the latest set of Nix expressions for those packages and upgrade using **nix-env**. Fortunately, there's a better way: *Nix channels*.

A Nix channel is just a URL that points to a place that contains a set of Nix expressions and a manifest. Using the command **nix-channel** you can automatically stay up to date with whatever is available at that URL.

You can "subscribe" to a channel using **nix-channel --add**, e.g.,

```
$ nix-channel --add https://nixos.org/channels/nixpkgs-unstable
```

subscribes you to a channel that always contains that latest version of the Nix Packages collection. (Subscribing really just means that the URL is added to the file `~/.nix-channels`, where it is read by subsequent calls to **nix-channel --update**.) You can "unsubscribe" using **nix-channel --remove**:

```
$ nix-channel --remove nixpkgs
```

To obtain the latest Nix expressions available in a channel, do

```
1 $ nix-channel --update
```

This downloads and unpacks the Nix expressions in every channel (downloaded from `url`/nixexprs.tar.bz2). It also makes the union of each channel's Nix expressions available by default to **nix-env** operations (via the symlink `~/.nix-defexpr/channels`). Consequently, you can then say

```
$ nix-env -u
```

to upgrade all packages in your profile to the latest versions available in the subscribed channels.

Chapter 13. Sharing Packages Between Machines

Sometimes you want to copy a package from one machine to another. Or, you want to install some packages and you know that another machine already has some or all of those packages or their dependencies. In that case there are mechanisms to quickly copy packages between machines.

13.1. Serving a Nix store via HTTP

You can easily share the Nix store of a machine via HTTP. This allows other machines to fetch store paths from that machine to speed up installations. It uses the same *binary cache* mechanism that Nix usually uses to fetch pre-built binaries from `https://cache.nixos.org`.

The daemon that handles binary cache requests via HTTP, **nix-serve**, is not part of the Nix distribution, but you can install it from Nixpkgs:

```
$ nix-env -i nix-serve
```

You can then start the server, listening for HTTP connections on whatever port you like:

```
$ nix-serve -p 8080
```

To check whether it works, try the following on the client:

```
$ curl http://avalon:8080/nix-cache-info
```

which should print something like:

```
StoreDir: /nix/store
WantMassQuery: 1
Priority: 30
```

On the client side, you can tell Nix to use your binary cache using `--option extra-binary-caches`, e.g.:

```
$ nix-env -i firefox --option extra-binary-caches http://avalon:8080/
```

The option `extra-binary-caches` tells Nix to use this binary cache in addition to your default caches, such as `https://cache.nixos.org`. Thus, for any path in the closure of Firefox, Nix will first check if the path is available on the server `avalon` or another binary caches. If not, it will fall back to building from source.

You can also tell Nix to always use your binary cache by adding a line to the `nix.conf` configuration file like this:

```
binary-caches = http://avalon:8080/ https://cache.nixos.org/
```

13.2. Copying Closures Via SSH

The command **nix-copy-closure** copies a Nix store path along with all its dependencies to or from another machine via the SSH protocol. It doesn't copy store paths that are already present on the target machine. For example, the following command copies Firefox with all its dependencies:

```
$ nix-copy-closure --to alice@itchy.example.org $(type -p firefox)
```

See nix-copy-closure(1) for details.

With **nix-store --export** and **nix-store --import** you can write the closure of a store path (that is, the path and all its dependencies) to a file, and then unpack that file into another Nix store. For example,

```
$ nix-store --export $(nix-store -qR $(type -p firefox)) > firefox.closure
```

writes the closure of Firefox to a file. You can then copy this file to another machine and install the closure:

```
$ nix-store --import < firefox.closure
```

Any store paths in the closure that are already present in the target store are ignored. It is also possible to pipe the export into another command, e.g. to copy and install a closure directly to/on another machine:

```
$ nix-store --export $(nix-store -qR $(type -p firefox)) | bzip2 | \
    ssh alice@itchy.example.org "bunzip2 | nix-store --import"
```

However, **nix-copy-closure** is generally more efficient because it only copies paths that are not already present in the target Nix store.

13.3. Serving a Nix store via SSH

You can tell Nix to automatically fetch needed binaries from a remote Nix store via SSH. For example, the following installs Firefox, automatically fetching any store paths in Firefox's closure if they are available on the server `avalon`:

```
$ nix-env -i firefox --substituters ssh://alice@avalon
```

This works similar to the binary cache substituter that Nix usually uses, only using SSH instead of HTTP: if a store path `P` is needed, Nix will first check if it's available in the Nix store on `avalon`. If not, it will fall back to using the binary cache substituter, and then to building from source.

Note

The SSH substituter currently does not allow you to enter an SSH passphrase interactively. Therefore, you should use **ssh-add** to load the decrypted private key into **ssh-agent**.

You can also copy the closure of some store path, without installing it into your profile, e.g.

```
$ nix-store -r /nix/store/m85bxg...-firefox-34.0.5 --substituters
    ↪ ssh://alice@avalon
```

This is essentially equivalent to doing

```
$ nix-copy-closure --from alice@avalon /nix/store/m85bxg...-firefox-34.0.5
```

You can use SSH's *forced command* feature to set up a restricted user account for SSH substituter access, allowing read-only access to the local Nix store, but nothing more. For example, add the following lines to `sshd_config` to restrict the user `nix-ssh`:

```
Match User nix-ssh
  AllowAgentForwarding no
  AllowTcpForwarding no
  PermitTTY no
  PermitTunnel no
  X11Forwarding no
  ForceCommand nix-store --serve
Match All
```

On NixOS, you can accomplish the same by adding the following to your `configuration.nix`:

```
nix.sshServe.enable = true;
nix.sshServe.keys = [ "ssh-dss AAAAB3NzaC1k... bob@example.org" ];
```

where the latter line lists the public keys of users that are allowed to connect.

13.4. Serving a Nix store via AWS S3 or S3-compatible Service

Nix has built-in support for storing and fetching store paths from Amazon S3 and S3 compatible services. This uses the same *binary* cache mechanism that Nix usually uses to fetch prebuilt binaries from `cache.nixos.org`.

The following options can be specified as URL parameters to the S3 URL:

`profile` The name of the AWS configuration profile to use. By default Nix will use the `default` profile.

`region` The region of the S3 bucket. `us-east-1` by default.

If your bucket is not in `us-east-1`, you should always explicitly specify the region parameter.

`endpoint` The URL to your S3-compatible service, for when not using Amazon S3. Do not specify this value if you're using Amazon S3.

Note

This endpoint must support HTTPS and will use path-based addressing instead of virtual host based addressing.

`scheme` The scheme used for S3 requests, `https` (default) or `http`. This option allows you to disable HTTPS for binary caches which don't support it.

Note

HTTPS should be used if the cache might contain sensitive information.

In this example we will use the bucket named `example-nix-cache`.

13.4.1. Anonymous Reads to your S3-compatible binary cache

If your binary cache is publicly accessible and does not require authentication, the simplest and easiest way to use Nix with your S3 compatible binary cache is to use the HTTP URL for that cache.

For AWS S3 the binary cache URL for example bucket will be exactly `https://example-nix-cache.s3.amazonaws.com` or `s3://example-nix-cache`. For S3 compatible binary caches, consult that cache's documentation.

Your bucket will need the following bucket policy:

```
{
    "Id": "DirectReads",
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowDirectReads",
            "Action": [
                "s3:GetObject",
                "s3:GetBucketLocation"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:s3:::example-nix-cache",
                "arn:aws:s3:::example-nix-cache/*"
            ],
            "Principal": "*"
        }
    ]
}
```

13.4.2. Authenticated Reads to your S3 binary cache

For AWS S3 the binary cache URL for example bucket will be exactly `s3://example-nix-cache`.

Nix will use the default credential provider chain for authenticating requests to Amazon S3.

Nix supports authenticated reads from Amazon S3 and S3 compatible binary caches.

Your bucket will need a bucket policy allowing the desired users to perform the `s3:GetObject` and `s3:GetBucketLocation` action on all objects in the bucket. The anonymous policy in Section 13.4.1, "Anonymous Reads to your S3-compatible binary cache" can be updated to have a restricted `Principal` to support this.

13.4.3. Authenticated Writes to your S3-compatible binary cache

Nix support fully supports writing to Amazon S3 and S3 compatible buckets. The binary cache URL for our example bucket will be `s3://example-nix-cache`.

Nix will use the default credential provider chain for authenticating requests to Amazon S3.

Your account will need the following IAM policy to upload to the cache:

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
    {
      "Sid": "UploadToCache",
      "Effect": "Allow",
      "Action": [
        "s3:AbortMultipartUpload",
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::example-nix-cache",
        "arn:aws:s3:::example-nix-cache/*"
      ]
    }
  ]
}
```

Example 13.1. Uploading with a specific credential profile for Amazon S3

nix copy --to 's3://example-nix-cache?profile=cache-upload&region=eu-west-2' nixpkgs.hello

Example 13.2. Uploading to an S3-Compatible Binary Cache

nix copy --to 's3://example-nix-cache?profile=cache-upload&scheme=https&endpoint=minio.example.com' nixpkgs.hello

Part IV. Writing Nix Expressions

This chapter shows you how to write Nix expressions, which instruct Nix how to build packages. It starts with a simple example (a Nix expression for GNU Hello), and then moves on to a more in-depth look at the Nix expression language.

Note

This chapter is mostly about the Nix expression language. For more extensive information on adding packages to the Nix Packages collection (such as functions in the standard environment and coding conventions), please consult its manual.

Chapter 14. A Simple Nix Expression

This section shows how to add and test the GNU Hello package to the Nix Packages collection. Hello is a program that prints out the text "Hello, world!".

To add a package to the Nix Packages collection, you generally need to do three things:

1. Write a Nix expression for the package. This is a file that describes all the inputs involved in building the package, such as dependencies, sources, and so on.
2. Write a *builder*. This is a shell script[2] that actually builds the package from the inputs.

3. Add the package to the file `pkgs/top-level/all-packages.nix`. The Nix expression written in the first step is a *function*; it requires other packages in order to build it. In this step you put it all together, i.e., you call the function with the right arguments to build the actual package.

14.1. Expression Syntax

Example 14.1. Nix expression for GNU Hello (`default.nix`)

```
{ stdenv, fetchurl, perl }:

stdenv.mkDerivation {
  name = "hello-2.1.1";
  builder = ./builder.sh;
  src = fetchurl {
    url = ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
    sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
  };
  inherit perl;
}
```

Example 14.1, "Nix expression for GNU Hello (`default.nix`)" shows a Nix expression for GNU Hello. It's actually already in the Nix Packages collection in `pkgs/applications/misc/hello/ex-1/default.nix`. It is customary to place each package in a separate directory and call the single Nix expression in that directory `default.nix`. The file has the following elements (referenced from the figure by number):

---

❶

This states that the expression is a *function* that expects to be called with three arguments: `stdenv`, `fetchurl`, and `perl`. They are needed to build Hello, but we don't know how to build them here; that's why they are function arguments. `stdenv` is a package that is used by almost all Nix Packages packages; it provides a "standard" environment consisting of the things you would expect in a basic Unix environment: a C/C++ compiler (GCC, to be precise), the Bash shell, fundamental Unix tools such as **cp**, **grep**, **tar**, etc. `fetchurl` is a function that downloads files. `perl` is the Perl interpreter.
Nix functions generally have the form `{ x, y, ..., z }: e` where `x`, `y`, etc. are the names of the expected arguments, and where *`e`* is the body of the function. So here, the entire remainder of the file is the body of the function; when given the required arguments, the body should describe how to build an instance of the Hello package.

❷

So we have to build a package. Building something from other stuff is called a *derivation* in Nix (as opposed to sources, which are built by humans instead of computers). We perform a derivation by calling `stdenv.mkDerivation`. `mkDerivation` is a function provided by `stdenv` that builds a package from a set of *attributes*. A set is just a list of key/value pairs where each key is a string and each value is an arbitrary Nix expression. They take the general form `{ name1` = *`expr1`*; `...` *`nameN`* = *`exprN`*; }.

❸ The attribute `name` specifies the symbolic name and version of the package. Nix doesn't really care about these things, but they are used by for instance **nix-env -q** to show a "human-readable" name for packages. This attribute is required by `mkDerivation`.

❹ The attribute `builder` specifies the builder. This attribute can sometimes be omitted, in which case `mkDerivation` will fill in a default builder (which does a `configure; make; make install`, in essence). Hello is sufficiently simple that the default builder would suffice, but in this case, we will show an actual builder for educational purposes. The value **./builder.sh** refers to the shell script shown in Example 14.2, "Build script for GNU Hello (`builder.sh`)", discussed below.

❺ The builder has to know what the sources of the package are. Here, the attribute `src` is bound to the result of a call to the **fetchurl** function. Given a URL and a SHA-256 hash of the expected contents of the file at that URL, this function builds a derivation that downloads the file and checks its hash. So the sources are a dependency that like all other dependencies is built before Hello itself is built.
Instead of `src` any other name could have been used, and in fact there can be any number of sources (bound to different attributes). However, `src` is customary, and it's also expected by the default builder (which we don't use in this example).

❻ Since the derivation requires Perl, we have to pass the value of the `perl` function argument to the builder. All attributes in the set are actually passed as environment variables to the builder, so declaring an attribute

```
perl = perl;
```

will do the trick: it binds an attribute `perl` to the function argument which also happens to be called `perl`. However, it looks a bit silly, so there is a shorter syntax. The `inherit` keyword causes the specified attributes to be bound to whatever variables with the same name happen to be in scope.

14.2. Build Script

Example 14.2. Build script for GNU Hello (`builder.sh`)

```
source $stdenv/setup

PATH=$perl/bin:$PATH

tar xvfz $src
cd hello-*
./configure --prefix=$out
make
make install
```

Example 14.2, "Build script for GNU Hello (`builder.sh`)" shows the builder referenced from Hello's Nix expression (stored in `pkgs/applications/misc/hello/ex-1/builder.sh`). The builder can actually be made a lot shorter by using the *generic builder* functions provided by `stdenv`, but here we write out the build steps to elucidate what a builder does. It performs the following steps:

❶	When Nix runs a builder, it initially completely clears the environment (except for the attributes declared in the derivation). For instance, the `PATH` variable is empty[3]. This is done to prevent undeclared inputs from being used in the build process. If for example the `PATH` contained `/usr/bin`, then you might accidentally use `/usr/bin/gcc`. So the first step is to set up the environment. This is done by calling the `setup` script of the standard environment. The environment variable `stdenv` points to the location of the standard environment being used. (It wasn't specified explicitly as an attribute in Example 14.1, "Nix expression for GNU Hello (`default.nix`)", but `mkDerivation` adds it automatically.)
❷	Since Hello needs Perl, we have to make sure that Perl is in the `PATH`. The `perl` environment variable points to the location of the Perl package (since it was passed in as an attribute to the derivation), so `$perl`/bin is the directory containing the Perl interpreter.
❸	Now we have to unpack the sources. The `src` attribute was bound to the result of fetching the Hello source tarball from the network, so the `src` environment variable points to the location in the Nix store to which the tarball was downloaded. After unpacking, we **cd** to the resulting source directory. The whole build is performed in a temporary directory created in `/tmp`, by the way. This directory is removed after the builder finishes, so there is no need to clean up the sources afterwards. Also, the temporary directory is always newly created, so you don't have to worry about files from previous builds interfering with the current build.
❹	GNU Hello is a typical Autoconf-based package, so we first have to run its `configure` script. In Nix every package is stored in a separate location in the Nix store, for instance `/nix/store/9a54ba97fb71b65fda531012d0443ce2-hello-2.1.1`. Nix computes this path by cryptographically hashing all attributes of the derivation. The path is passed to the builder through the `out` environment variable. So here we give `configure` the parameter `--prefix=$out` to cause Hello to be installed in the expected location.
❺	Finally we build Hello (`make`) and install it into the location specified by `out` (`make install`).

If you are wondering about the absence of error checking on the result of various commands called in the builder: this is because the shell script is evaluated with Bash's `-e` option, which causes the script to be aborted if any

command fails without an error check.

14.3. Arguments and Variables

Example 14.3. Composing GNU Hello (`all-packages.nix`)

```
...

rec {

  hello = import ../applications/misc/hello/ex-1  {
    inherit fetchurl stdenv perl;
  };

  perl = import ../development/interpreters/perl {
    inherit fetchurl stdenv;
  };

  fetchurl = import ../build-support/fetchurl {
    inherit stdenv; ...
  };

  stdenv = ...;

}
```

The Nix expression in Example 14.1, "Nix expression for GNU Hello (`default.nix`)" is a function; it is missing some arguments that have to be filled in somewhere. In the Nix Packages collection this is done in the file `pkgs/top-level/all-packages.nix`, where all Nix expressions for packages are imported and called with the appropriate arguments. Example 14.3, "Composing GNU Hello (`all-packages.nix`)" shows some fragments of `all-packages.nix`.

---

❶ This file defines a set of attributes, all of which are concrete derivations (i.e., not functions). In fact, we define a *mutually recursive* set of attributes. That is, the attributes can refer to each other. This is precisely what we want since we want to "plug" the various packages into each other.

❷ Here we *import* the Nix expression for GNU Hello. The import operation just loads and returns the specified Nix expression. In fact, we could just have put the contents of Example 14.1, "Nix expression for GNU Hello (`default.nix`)" in `all-packages.nix` at this point. That would be completely equivalent, but it would make the file rather bulky.
Note that we refer to `../applications/misc/hello/ex-1`, not `../applications/misc/hello/ex-1/default.nix`. When you try to import a directory, Nix automatically appends `/default.nix` to the file name.

❸ This is where the actual composition takes place. Here we *call* the function imported from `../applications/misc/hello/ex-1` with a set containing the things that the function expects, namely `fetchurl`, `stdenv`, and `perl`. We use inherit again to use the attributes defined in the surrounding scope (we could also have written `fetchurl = fetchurl;`, etc.). The result of this function call is an actual derivation that can be built by Nix (since when we fill in the arguments of the function, what we get is its body, which is the call to `stdenv.mkDerivation` in Example 14.1, "Nix expression for GNU Hello (`default.nix`)").

Note

Nixpkgs has a convenience function `callPackage` that imports and calls a function, filling in any missing arguments by passing the corresponding attribute from the Nixpkgs set, like this:

```
hello = callPackage
    ↪ ../applications/misc/hello/ex-1 {
    ↪ };
```

If necessary, you can set or override arguments:

```
1 hello = callPackage
    ↪ ../applications/misc/hello/ex-1 {
    ↪ stdenv = myStdenv; };
```

❹ Likewise, we have to instantiate Perl, `fetchurl`, and the standard environment.

14.4. Building and Testing

You can now try to build Hello. Of course, you could do `nix-env -i hello`, but you may not want to install a possibly broken package just yet. The best way to test the package is by using the command **nix-build**, which builds a Nix expression and creates a symlink named `result` in the current directory:

```
$ nix-build -A hello
building path `/nix/store/632d2b22514d...-hello-2.1.1'
hello-2.1.1/
hello-2.1.1/intl/
hello-2.1.1/intl/ChangeLog
...

$ ls -l result
lrwxrwxrwx ... 2006-09-29 10:43 result -> /nix/store/632d2b22514d...-hello-2.1.1

$ ./result/bin/hello
Hello, world!
```

The `-A` option selects the `hello` attribute. This is faster than using the symbolic package name specified by the `name` attribute (which also happens to be `hello`) and is unambiguous (there can be multiple packages with the symbolic name `hello`, but there can be only one attribute in a set named `hello`).

nix-build registers the `./result` symlink as a garbage collection root, so unless and until you delete the `./result` symlink, the output of the build will be safely kept on your system. You can use **nix-build**'s `-o` switch to give the symlink another name.

Nix has transactional semantics. Once a build finishes successfully, Nix makes a note of this in its database: it registers that the path denoted by `out` is now "valid". If you try to build the derivation again, Nix will see that the path is already valid and finish immediately. If a build fails, either because it returns a non-zero exit code, because Nix or the builder are killed, or because the machine crashes, then the output paths will not be registered as valid. If you try to build the derivation again, Nix will remove the output paths if they exist (e.g., because the builder died half-way through `make install`) and try again. Note that there is no "negative caching": Nix doesn't remember that a build failed, and so a failed build can always be repeated. This is because Nix cannot distinguish between permanent failures (e.g., a compiler error due to a syntax error in the source) and transient failures (e.g., a disk full condition).

Nix also performs locking. If you run multiple Nix builds simultaneously, and they try to build the same derivation, the first Nix instance that gets there will perform the build, while the others block (or perform other derivations if available) until the build finishes:

```
$ nix-build -A hello
waiting for lock on `/nix/store/0h5b7hp8d4hqfrw8igvx97x1xawrjnac-hello-2.1.1x'
```

So it is always safe to run multiple instances of Nix in parallel (which isn't the case with, say, **make**).

If you have a system with multiple CPUs, you may want to have Nix build different derivations in parallel (insofar as possible). Just pass the option `-j N`, where *N* is the maximum number of jobs to be run in parallel, or set. Typically this should be the number of CPUs.

14.5. Generic Builder Syntax

Recall from Example 14.2, "Build script for GNU Hello (`builder.sh`)" that the builder looked something like this:

```
PATH=$perl/bin:$PATH
tar xvfz $src
cd hello-*
./configure --prefix=$out
make
make install
```

The builders for almost all Unix packages look like this -- set up some environment variables, unpack the sources, configure, build, and install. For this reason the standard environment provides some Bash functions that automate the build process. A builder using the generic build facilities in shown in Example 14.4, "Build script using the generic build functions".

Example 14.4. Build script using the generic build functions

```
buildInputs="$perl"

source $stdenv/setup

genericBuild
```

❶ The `buildInputs` variable tells `setup` to use the indicated packages as "inputs". This means that if a package provides a

❷ The function `genericBuild` is defined in the file `$stdenv/setup`.

❸ The final step calls the shell function `genericBuild`, which performs the steps that were done explicitly in Example 14.2,

Discerning readers will note that the `buildInputs` could just as well have been set in the Nix expression, like this:

```
  buildInputs = [ perl ];
```

The `perl` attribute can then be removed, and the builder becomes even shorter:

```
source $stdenv/setup
genericBuild
```

In fact, `mkDerivation` provides a default builder that looks exactly like that, so it is actually possible to omit the builder for Hello entirely.

---

[2] In fact, it can be written in any language, but typically it's a **bash** shell script.

[3] Actually, it's initialised to `/path-not-set` to prevent Bash from setting it to a default value.

[4] How does it work? `setup` tries to source the file `pkg`/nix-support/setup-hook of all dependencies. These "setup hooks" can then set up whatever environment variables they want; for instance, the setup hook for Perl sets the `PERL5LIB` environment variable to contain the `lib/site_perl` directories of all inputs.

Chapter 15. Nix Expression Language

The Nix expression language is a pure, lazy, functional language. Purity means that operations in the language don't have side-effects (for instance, there is no variable assignment). Laziness means that arguments to functions are evaluated only when they are needed. Functional means that functions are "normal" values that can be passed around and manipulated in interesting ways. The language is not a full-featured, general purpose language. Its main job is to describe packages, compositions of packages, and the variability within packages.

This section presents the various features of the language.

15.1. Values

Simple Values

Nix has the following basic data types:

- *Strings* can be written in three ways.

  The most common way is to enclose the string between double quotes, e.g., `"foo bar"`. Strings can span multiple lines. The special characters `"` and `\` and the character sequence `${` must be escaped by prefixing them with a backslash (`\`). Newlines, carriage returns and tabs can be written as `\n`, `\r` and `\t`, respectively.

  You can include the result of an expression into a string by enclosing it in `${...}`, a feature known as *antiquotation*. The enclosed expression must evaluate to something that can be coerced into a string (meaning that it must be a string, a path, or a derivation). For instance, rather than writing

  ```
  "--with-freetype2-library=" + freetype + "/lib"
  ```

  (where `freetype` is a derivation), you can instead write the more natural

  ```
  "--with-freetype2-library=${freetype}/lib"
  ```

  The latter is automatically translated to the former. A more complicated example (from the Nix expression for Qt):

  ```
  configureFlags = "
    -system-zlib -system-libpng -system-libjpeg
    ${if openglSupport then "-dlopen-opengl
      -L${mesa}/lib -I${mesa}/include
      -L${libXmu}/lib -I${libXmu}/include" else ""}
    ${if threadSupport then "-thread" else "-no-thread"}
  ";
  ```

Note that Nix expressions and strings can be arbitrarily nested; in this case the outer string contains various antiquotations that themselves contain strings (e.g., `"-thread"`), some of which in turn contain expressions (e.g., `${mesa}`).

The second way to write string literals is as an *indented string*, which is enclosed between pairs of *double single-quotes*, like so:

```
''
  This is the first line.
  This is the second line.
    This is the third line.
''
```

This kind of string literal intelligently strips indentation from the start of each line. To be precise, it strips from each line a number of spaces equal to the minimal indentation of the string as a whole (disregarding the indentation of empty lines). For instance, the first and second line are indented two space, while the third line is indented four spaces. Thus, two spaces are stripped from each line, so the resulting string is

```
"This is the first line.\nThis is the second line.\n  This is the third line.\n"
```

Note that the whitespace and newline following the opening `''` is ignored if there is no non-whitespace text on the initial line.

Antiquotation (`${expr}`) is supported in indented strings.

Since `${` and `''` have special meaning in indented strings, you need a way to quote them. `$` can be escaped by prefixing it with `''` (that is, two single quotes), i.e., `''$`. `''` can be escaped by prefixing it with `'`, i.e., `'''`. `$` removes any special meaning from the following `$`. Linefeed, carriage-return and tab characters can be written as `''\n`, `''\r`, `''\t`, and `''\` escapes any other character.

Indented strings are primarily useful in that they allow multi-line string literals to follow the indentation of the enclosing Nix expression, and that less escaping is typically necessary for strings representing languages such as shell scripts and configuration files because `''` is much less common than `"`. Example:

```
stdenv.mkDerivation {
  ...
  postInstall =
    ''
      mkdir $out/bin $out/etc
      cp foo $out/bin
      echo "Hello World" > $out/etc/foo.conf
      ${if enableBar then "cp bar $out/bin" else ""}
    '';
  ...
}
```

Finally, as a convenience, *URIs* as defined in appendix B of RFC 2396 can be written *as is*, without quotes. For instance, the string `"http://example.org/foo.tar.bz2"` can also be written as `http://example.org/foo.tar.bz2`.

- Numbers, which can be *integers* (like `123`) or *floating point* (like `123.43` or `.27e13`).

  Numbers are type-compatible: pure integer operations will always return integers, whereas any operation involving at least one floating point number will have a floating point number as a result.

- *Paths*, e.g., `/bin/sh` or `./builder.sh`. A path must contain at least one slash to be recognised as such; for instance, `builder.sh` is not a path[5]. If the file name is relative, i.e., if it does not begin with a slash, it is made absolute at parse time relative to the directory of the Nix expression that contained it. For instance, if a Nix expression in `/foo/bar/bla.nix` refers to `../xyzzy/fnord.nix`, the absolute path is `/foo/xyzzy/fnord.nix`.

  If the first component of a path is a `~`, it is interpreted as if the rest of the path were relative to the user's home directory. e.g. `~/foo` would be equivalent to `/home/edolstra/foo` for a user whose home directory is `/home/edolstra`.

Paths can also be specified between angle brackets, e.g. `<nixpkgs>`. This means that the directories listed in the environment variable `NIX_PATH` will be searched for the given file or directory name.

- *Booleans* with values `true` and `false`.
- The null value, denoted as `null`.

Lists

Lists are formed by enclosing a whitespace-separated list of values between square brackets. For example,

```
[ 123 ./foo.nix "abc" (f { x = y; }) ]
```

defines a list of four elements, the last being the result of a call to the function `f`. Note that function calls have to be enclosed in parentheses. If they had been omitted, e.g.,

```
[ 123 ./foo.nix "abc" f { x = y; } ]
```

the result would be a list of five elements, the fourth one being a function and the fifth being a set.

Note that lists are only lazy in values, and they are strict in length.

Sets

Sets are really the core of the language, since ultimately the Nix language is all about creating derivations, which are really just sets of attributes to be passed to build scripts.

Sets are just a list of name/value pairs (called *attributes*) enclosed in curly brackets, where each value is an arbitrary expression terminated by a semicolon. For example:

```
{ x = 123;
  text = "Hello";
  y = f { bla = 456; };
}
```

This defines a set with attributes named `x`, `text`, `y`. The order of the attributes is irrelevant. An attribute name may only occur once.

Attributes can be selected from a set using the `.` operator. For instance,

```
{ a = "Foo"; b = "Bar"; }.a
```

evaluates to `"Foo"`. It is possible to provide a default value in an attribute selection using the `or` keyword. For example,

```
{ a = "Foo"; b = "Bar"; }.c or "Xyzzy"
```

will evaluate to `"Xyzzy"` because there is no `c` attribute in the set.

You can use arbitrary double-quoted strings as attribute names:

```
{ "foo ${bar}" = 123; "nix-1.0" = 456; }."foo ${bar}"
```

This will evaluate to `123` (Assuming `bar` is antiquotable). In the case where an attribute name is just a single antiquotation, the quotes can be dropped:

```
{ foo = 123; }.${bar} or 456
```

This will evaluate to `123` if `bar` evaluates to `"foo"` when coerced to a string and `456` otherwise (again assuming `bar` is antiquotable).

In the special case where an attribute name inside of a set declaration evaluates to `null` (which is normally an error, as `null` is not antiquotable), that attribute is simply not added to the set:

```
{ ${if foo then "bar" else null} = true; }
```

This will evaluate to `{}` if `foo` evaluates to `false`.

A set that has a `__functor` attribute whose value is callable (i.e. is itself a function or a set with a `__functor` attribute whose value is callable) can be applied as if it were a function, with the set itself passed in first , e.g.,

```
let add = { __functor = self: x: x + self.x; };
    inc = add // { x = 1; };
in inc 1
```

evaluates to `2`. This can be used to attach metadata to a function without the caller needing to treat it specially, or to implement a form of object-oriented programming, for example.

15.2. Language Constructs

Recursive sets

Recursive sets are just normal sets, but the attributes can refer to each other. For example,

```
rec {
  x = y;
  y = 123;
}.x
```

evaluates to `123`. Note that without `rec` the binding `x = y;` would refer to the variable `y` in the surrounding scope, if one exists, and would be invalid if no such variable exists. That is, in a normal (non-recursive) set, attributes are not added to the lexical scope; in a recursive set, they are.

Recursive sets of course introduce the danger of infinite recursion. For example,

```
rec {
  x = y;
  y = x;
}.x
```

does not terminate[6].

Let-expressions

A let-expression allows you to define local variables for an expression. For instance,

```
let
  x = "foo";
  y = "bar";
in x + y
```

evaluates to `"foobar"`.

Inheriting attributes

When defining a set or in a let-expression it is often convenient to copy variables from the surrounding lexical scope (e.g., when you want to propagate attributes). This can be shortened using the `inherit` keyword. For instance,

```
let x = 123; in
{ inherit x;
  y = 456;
}
```

is equivalent to

```
let x = 123; in
{ x = x;
  y = 456;
}
```

and both evaluate to `{ x = 123; y = 456; }`. (Note that this works because `x` is added to the lexical scope by the `let` construct.) It is also possible to inherit attributes from another set. For instance, in this fragment from `all-packages.nix`,

```
  graphviz = (import ../tools/graphics/graphviz) {
    inherit fetchurl stdenv libpng libjpeg expat x11 yacc;
    inherit (xlibs) libXaw;
  };

  xlibs = {
    libX11 = ...;
    libXaw = ...;
    ...
  }

  libpng = ...;
  libjpg = ...;
  ...
```

the set used in the function call to the function defined in `../tools/graphics/graphviz` inherits a number of variables from the surrounding scope (`fetchurl` ... `yacc`), but also inherits `libXaw` (the X Athena Widgets) from the `xlibs` (X11 client-side libraries) set.

Summarizing the fragment

```
...
inherit x y z;
inherit (src-set) a b c;
...
```

is equivalent to

```
...
x = x; y = y; z = z;
a = src-set.a; b = src-set.b; c = src-set.c;
...
```

when used while defining local variables in a let-expression or while defining a set.

Functions

Functions have the following form:

```
pattern: body
```

The pattern specifies what the argument of the function must look like, and binds variables in the body to (parts of) the argument. There are three kinds of patterns:

- If a pattern is a single identifier, then the function matches any argument. Example:

  ```
  let negate = x: !x;
      concat = x: y: x + y;
  in if negate true then concat "foo" "bar" else ""
  ```

  Note that `concat` is a function that takes one argument and returns a function that takes another argument. This allows partial parameterisation (i.e., only filling some of the arguments of a function); e.g.,

  ```
  map (concat "foo") [ "bar" "bla" "abc" ]
  ```

  evaluates to `[ "foobar" "foobla" "fooabc" ]`.

- A *set pattern* of the form `{ name1, name2, ..., nameN }` matches a set containing the listed attributes, and binds the values of those attributes to variables in the function body. For example, the function

  ```
  { x, y, z }: z + y + x
  ```

  can only be called with a set containing exactly the attributes `x`, `y` and `z`. No other attributes are allowed. If you want to allow additional arguments, you can use an ellipsis (`...`):

  ```
  { x, y, z, ... }: z + y + x
  ```

  This works on any set that contains at least the three named attributes.

  It is possible to provide *default values* for attributes, in which case they are allowed to be missing. A default value is specified by writing `name ? e`, where *`e`* is an arbitrary expression. For example,

  ```
  { x, y ? "foo", z ? "bar" }: z + y + x
  ```

  specifies a function that only requires an attribute named `x`, but optionally accepts `y` and `z`.

- An `@`-pattern provides a means of referring to the whole value being matched:

  ```
  args@{ x, y, z, ... }: z + y + x + args.a
  ```

  but can also be written as:

  ```
  { x, y, z, ... } @ args: z + y + x + args.a
  ```

  Here `args` is bound to the entire argument, which is further matched against the pattern `{ x, y, z, ... }`. `@`-pattern makes mainly sense with an ellipsis(`...`) as you can access attribute names as `a`, using `args.a`, which was given as an additional attribute to the function.

  Warning

  The `args@` expression is bound to the argument passed to the function which means that attributes with defaults that aren't explicitly specified in the function call won't cause an evaluation error, but won't exist in `args`.

  For instance

  ```
  let
    function = args@{ a ? 23, ... }: args;
  in
  function {}
  ```

  will evaluate to an empty attribute set.

Note that functions do not have names. If you want to give them a name, you can bind them to an attribute, e.g.,

```
let concat = { x, y }: x + y;
in concat { x = "foo"; y = "bar"; }
```

Conditionals

Conditionals look like this:

```
if e1 then e2 else e3
```

where *`e1`* is an expression that should evaluate to a Boolean value (`true` or `false`).

Assertions

Assertions are generally used to check that certain requirements on or between features and dependencies hold. They look like this:

```
assert e1; e2
```

where *e1* is an expression that should evaluate to a Boolean value. If it evaluates to `true`, *e2* is returned; otherwise expression evaluation is aborted and a backtrace is printed.

Example 15.1. Nix expression for Subversion

```
{ localServer ? false
, httpServer ? false
, sslSupport ? false
, pythonBindings ? false
, javaSwigBindings ? false
, javahlBindings ? false
, stdenv, fetchurl
, openssl ? null, httpd ? null, db4 ? null, expat, swig ? null, j2sdk ? null
}:

assert localServer -> db4 != null;
assert httpServer -> httpd != null && httpd.expat == expat;
assert sslSupport -> openssl != null && (httpServer -> httpd.openssl == openssl);
assert pythonBindings -> swig != null && swig.pythonSupport;
assert javaSwigBindings -> swig != null && swig.javaSupport;
assert javahlBindings -> j2sdk != null;

stdenv.mkDerivation {
  name = "subversion-1.1.1";
  ...
  openssl = if sslSupport then openssl else null;
  ...
}
```

Example 15.1, "Nix expression for Subversion" show how assertions are used in the Nix expression for Subversion.

❶ This assertion states that if Subversion is to have support for local repositories, then Berkeley DB is needed. So if the Sub

❷ This is a more subtle condition: if Subversion is built with Apache (`httpServer`) support, then the Expat library (an XM

❸ This assertion says that in order for Subversion to have SSL support (so that it can access `https` URLs), an OpenSSL lib

❹ The conditional here is not really related to assertions, but is worth pointing out: it ensures that if SSL support is disable

With-expressions

A *with-expression*,

```
with e1; e2
```

introduces the set *e1* into the lexical scope of the expression *e2*. For instance,

```
let as = { x = "foo"; y = "bar"; };
in with as; x + y
```

evaluates to `"foobar"` since the `with` adds the `x` and `y` attributes of `as` to the lexical scope in the expression `x + y`. The most common use of `with` is in conjunction with the `import` function. E.g.,

```
with (import ./definitions.nix); ...
```

makes all attributes defined in the file `definitions.nix` available as if they were defined locally in a `let`-expression.

The bindings introduced by `with` do not shadow bindings introduced by other means, e.g.

```
let a = 3; in with { a = 1; }; let a = 4; in with { a = 2; }; ...
```

establishes the same scope as

```
1 let a = 1; in let a = 2; in let a = 3; in let a = 4; in ...
```

Comments

Comments can be single-line, started with a `#` character, or inline/multi-line, enclosed within `/* ... */`.

15.3. Operators

Table 15.1, "Operators" lists the operators in the Nix expression language, in order of precedence (from strongest to weakest binding).

Table 15.1. Operators

Name	Syntax	Associativity
Select	*e* `.` *attrpath* [`or` *def*]	none
Application	*e1 e2*	left
Arithmetic Negation	`-` *e*	none
Has Attribute	*e* `?` *attrpath*	none
List Concatenation	*e1* `++` *e2*	right
Multiplication	*e1* `*` *e2*,	left
Division	*e1* `/` *e2*	left
Addition	*e1* `+` *e2*	left
Subtraction	*e1* `-` *e2*	left
String Concatenation	*string1* `+` *string2*	left
Not	`!` *e*	none
Update	*e1* `//` *e2*	right
Less Than	*e1* `<` *e2*,	none
Less Than or Equal To	*e1* `<=` *e2*	none
Greater Than	*e1* `>` *e2*	none
Greater Than or Equal To	*e1* `>=` *e2*	none
Equality	*e1* `==` *e2*	none
Inequality	*e1* `!=` *e2*	none
Logical AND	*e1* `&&` *e2*	left
Logical OR	*e1* `||` *e2*	left
Logical Implication	*e1* `->` *e2*	none

15.4. Derivations

The most important built-in function is `derivation`, which is used to describe a single derivation (a build action). It takes as input a set, the attributes of which specify the inputs of the build.

- There must be an attribute named `system` whose value must be a string specifying a Nix platform identifier, such as `"i686-linux"` or `"x86_64-darwin"`[7] The build can only be performed on a machine and operating system matching the platform identifier. (Nix can automatically forward builds for other platforms by forwarding them to other machines; see Chapter 16, *Remote Builds*.)

- There must be an attribute named `name` whose value must be a string. This is used as a symbolic name for the package by **nix-env**, and it is appended to the output paths of the derivation.
- There must be an attribute named `builder` that identifies the program that is executed to perform the build. It can be either a derivation or a source (a local file reference, e.g., `./builder.sh`).
- Every attribute is passed as an environment variable to the builder. Attribute values are translated to environment variables as follows:
  - Strings and numbers are just passed verbatim.
  - A *path* (e.g., `../foo/sources.tar`) causes the referenced file to be copied to the store; its location in the store is put in the environment variable. The idea is that all sources should reside in the Nix store, since all inputs to a derivation should reside in the Nix store.
  - A *derivation* causes that derivation to be built prior to the present derivation; its default output path is put in the environment variable.
  - Lists of the previous types are also allowed. They are simply concatenated, separated by spaces.
  - `true` is passed as the string `1`, `false` and `null` are passed as an empty string.
- The optional attribute `args` specifies command-line arguments to be passed to the builder. It should be a list.
- The optional attribute `outputs` specifies a list of symbolic outputs of the derivation. By default, a derivation produces a single output path, denoted as `out`. However, derivations can produce multiple output paths. This is useful because it allows outputs to be downloaded or garbage-collected separately. For instance, imagine a library package that provides a dynamic library, header files, and documentation. A program that links against the library doesn't need the header files and documentation at runtime, and it doesn't need the documentation at build time. Thus, the library package could specify:

  ```
  outputs = [ "lib" "headers" "doc" ];
  ```

  This will cause Nix to pass environment variables `lib`, `headers` and `doc` to the builder containing the intended store paths of each output. The builder would typically do something like

  ```
  ./configure --libdir=$lib/lib --includedir=$headers/include --docdir=$doc/share/doc
  ```

  for an Autoconf-style package. You can refer to each output of a derivation by selecting it as an attribute, e.g.

  ```
  1 buildInputs = [ pkg.lib pkg.headers ];
  ```

  The first element of `outputs` determines the *default output*. Thus, you could also write

  ```
  buildInputs = [ pkg pkg.headers ];
  ```

  since `pkg` is equivalent to `pkg.lib`.

The function `mkDerivation` in the Nixpkgs standard environment is a wrapper around `derivation` that adds a default value for `system` and always uses Bash as the builder, to which the supplied builder is passed as a command-line argument. See the Nixpkgs manual for details.

The builder is executed as follows:

- A temporary directory is created under the directory specified by `TMPDIR` (default `/tmp`) where the build will take place. The current directory is changed to this directory.
- The environment is cleared and set to the derivation attributes, as specified above.
- In addition, the following variables are set:
  - `NIX_BUILD_TOP` contains the path of the temporary directory for this build.
  - Also, `TMPDIR`, `TEMPDIR`, `TMP`, `TEMP` are set to point to the temporary directory. This is to prevent the builder from accidentally writing temporary files anywhere else. Doing so might cause interference by other processes.

  - `PATH` is set to `/path-not-set` to prevent shells from initialising it to their built-in default value.
  - `HOME` is set to `/homeless-shelter` to prevent programs from using `/etc/passwd` or the like to find the user's home directory, which could cause impurity. Usually, when `HOME` is set, it is used as the location of the home directory, even if it points to a non-existent path.
  - `NIX_STORE` is set to the path of the top-level Nix store directory (typically, `/nix/store`).
  - For each output declared in `outputs`, the corresponding environment variable is set to point to the intended path in the Nix store for that output. Each output path is a concatenation of the cryptographic hash of all build inputs, the `name` attribute and the output name. (The output name is omitted if it's `out`.)
- If an output path already exists, it is removed. Also, locks are acquired to prevent multiple Nix instances from performing the same build at the same time.
- A log of the combined standard output and error is written to `/nix/var/log/nix`.
- The builder is executed with the arguments specified by the attribute `args`. If it exits with exit code 0, it is considered to have succeeded.
- The temporary directory is removed (unless the `-K` option was specified).
- If the build was successful, Nix scans each output path for references to input paths by looking for the hash parts of the input paths. Since these are potential runtime dependencies, Nix registers them as dependencies of the output paths.
- After the build, Nix sets the last-modified timestamp on all files in the build result to 1 (00:00:01 1/1/1970 UTC), sets the group to the default group, and sets the mode of the file to 0444 or 0555 (i.e., read-only, with execute permission enabled if the file was originally executable). Note that possible `setuid` and `setgid` bits are cleared. Setuid and setgid programs are not currently supported by Nix. This is because the Nix archives used in deployment have no concept of ownership information, and because it makes the build result dependent on the user performing the build.

15.4.1. Advanced Attributes

Derivations can declare some infrequently used optional attributes.

`allowedReferences` The optional attribute `allowedReferences` specifies a list of legal references (dependencies) of the output of the builder. For example,

```
allowedReferences = [];
```

enforces that the output of a derivation cannot have any runtime dependencies on its inputs. To allow an output to have a runtime dependency on itself, use `"out"` as a list item. This is used in NixOS to check that generated files such as initial ramdisks for booting Linux don't have accidental dependencies on other paths in the Nix store.

`allowedRequisites` This attribute is similar to `allowedReferences`, but it specifies the legal requisites of the whole closure, so all the dependencies recursively. For example,

```
allowedRequisites = [ foobar ];
```

enforces that the output of a derivation cannot have any other runtime dependency than `foobar`, and in addition it enforces that `foobar` itself doesn't introduce any other dependency itself.

`disallowedReferences` The optional attribute `disallowedReferences` specifies a list of illegal references (dependencies) of the output of the builder. For example,

```
disallowedReferences = [ foo ];
```

enforces that the output of a derivation cannot have a direct runtime dependencies on the derivation `foo`.

`disallowedRequisites` This attribute is similar to `disallowedReferences`, but it specifies illegal requisites for the whole closure, so all the dependencies recursively. For example,

```
disallowedRequisites = [ foobar ];
```

enforces that the output of a derivation cannot have any runtime dependency on `foobar` or any other derivation depending recursively on `foobar`.

`exportReferencesGraph` This attribute allows builders access to the references graph of their inputs. The attribute is a list of inputs in the Nix store whose references graph the builder needs to know. The value of this attribute should be a list of pairs `[ name1` *`path1 name2 path2 ...`* `]`. The references graph of each *`pathN`* will be stored in a text file *`nameN`* in the temporary build directory. The text files have the format used by **nix-store --register-validity** (with the deriver fields left empty). For example, when the following derivation is built:

```
derivation {
  ...
  exportReferencesGraph = [ "libfoo-graph" libfoo ];
};
```

the references graph of `libfoo` is placed in the file `libfoo-graph` in the temporary build directory.

`exportReferencesGraph` is useful for builders that want to do something with the closure of a store path. Examples include the builders in NixOS that generate the initial ramdisk for booting Linux (a **cpio** archive containing the closure of the boot script) and the ISO-9660 image for the installation CD (which is populated with a Nix store containing the closure of a bootable NixOS configuration).

`impureEnvVars` This attribute allows you to specify a list of environment variables that should be passed from the environment of the calling user to the builder. Usually, the environment is cleared completely when the builder is executed, but with this attribute you can allow specific environment variables to be passed unmodified. For example, `fetchurl` in Nixpkgs has the line

```
impureEnvVars = [ "http_proxy" "https_proxy" ... ];
```

to make it use the proxy server configuration specified by the user in the environment variables `http_proxy` and friends.

This attribute is only allowed in fixed-output derivations, where impurities such as these are okay since (the hash of) the output is known in advance. It is ignored for all other derivations.

Warning

`impureEnvVars` implementation takes environment variables from the current builder process. When a daemon is building its environmental variables are used. Without the daemon, the environmental variables come from the environment of the **nix-build**.

`outputHash, outputHashAlgo, outputHashMode` These attributes declare that the derivation is a so-called *fixed-output derivation*, which means that a cryptographic hash of the output is already known in advance. When the build of a fixed-output derivation finishes, Nix computes the cryptographic hash of the output and compares it to the hash declared with these attributes. If there is a mismatch, the build fails.

The rationale for fixed-output derivations is derivations such as those produced by the `fetchurl` function. This function downloads a file from a given URL. To ensure that the downloaded file has not been modified, the caller must also specify a cryptographic hash of the file. For example,

```
fetchurl {
  url = http://ftp.gnu.org/pub/gnu/hello/hello-2.1.1.tar.gz;
  sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
}
```

It sometimes happens that the URL of the file changes, e.g., because servers are reorganised or no longer available. We then must update the call to `fetchurl`, e.g.,

```
fetchurl {
  url = ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
  sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
}
```

If a `fetchurl` derivation was treated like a normal derivation, the output paths of the derivation and *all derivations depending on it* would change. For instance, if we were to change the URL of the Glibc source distribution in Nixpkgs (a package on which almost all other packages depend) massive rebuilds would be needed. This is unfortunate for a change which we know cannot have a real effect as it propagates upwards through the dependency graph.

For fixed-output derivations, on the other hand, the name of the output path only depends on the `outputHash*` and `name` attributes, while all other attributes are ignored for the purpose of computing the output path. (The `name` attribute is included because it is part of the path.)

As an example, here is the (simplified) Nix expression for `fetchurl`:

```
{ stdenv, curl }: # The curl program is used for downloading.

{ url, sha256 }:

stdenv.mkDerivation {
  name = baseNameOf (toString url);
  builder = ./builder.sh;
  buildInputs = [ curl ];

  # This is a fixed-output derivation; the output must be a regular
  # file with SHA256 hash sha256.
  outputHashMode = "flat";
  outputHashAlgo = "sha256";
  outputHash = sha256;

  inherit url;
}
```

The `outputHashAlgo` attribute specifies the hash algorithm used to compute the hash. It can currently be `"sha1"`, `"sha256"` or `"sha512"`.

The `outputHashMode` attribute determines how the hash is computed. It must be one of the following two values:

`"flat"` The output must be a non-executable regular file. If it isn't, the build fails. The hash is simply computed over the contents of that file (so it's equal to what Unix commands like **sha256sum** or **sha1sum** produce).

This is the default.

`"recursive"` The hash is computed over the NAR archive dump of the output (i.e., the result of **nix-store --dump**). In this case, the output can be anything, including a directory tree.

The `outputHash` attribute, finally, must be a string containing the hash in either hexadecimal or base-32 notation. (See the **nix-hash** command for information about converting to and from base-32 notation.)

`passAsFile` A list of names of attributes that should be passed via files rather than environment variables. For example, if you have

```
passAsFile = ["big"];
big = "a very long string";
```

then when the builder runs, the environment variable `bigPath` will contain the absolute path to a temporary file containing `a very long string`. That is, for any attribute *x* listed in `passAsFile`, Nix will pass an environment variable `x`Path holding the path of the file containing the value of attribute *x*. This is useful when you need to pass large strings to a builder, since most operating systems impose a limit on the size of the environment (typically, a few hundred kilobyte).

`preferLocalBuild` If this attribute is set to `true` and distributed building is enabled, then, if possible, the derivaton will be built locally instead of forwarded to a remote machine. This is appropriate for trivial builders where the cost of doing a download or remote build would exceed the cost of building locally.

`allowSubstitutes` If this attribute is set to `false`, then Nix will always build this derivation; it will not try to substitute its outputs. This is useful for very trivial derivations (such as `writeText` in Nixpkgs) that are cheaper to build than to substitute from a binary cache.

15.5. Built-in Functions

This section lists the functions and constants built into the Nix expression evaluator. (The built-in function `derivation` is discussed above.) Some built-ins, such as `derivation`, are always in scope of every Nix expression; you can just access them right away. But to prevent polluting the namespace too much, most built-ins are not in scope. Instead, you can access them through the `builtins` built-in value, which is a set that contains all built-in functions and values. For instance, `derivation` is also available as `builtins.derivation`.

`abort` *`s`*, `builtins.abort` *`s`* Abort Nix expression evaluation, print error message *`s`*.

`builtins.add` *`e1`* *`e2`* Return the sum of the numbers *`e1`* and *`e2`*.

`builtins.all` *`pred`* *`list`* Return `true` if the function *`pred`* returns `true` for all elements of *`list`*, and `false` otherwise.

`builtins.any` *`pred`* *`list`* Return `true` if the function *`pred`* returns `true` for at least one element of *`list`*, and `false` otherwise.

`builtins.attrNames` *`set`* Return the names of the attributes in the set *`set`* in an alphabetically sorted list. For instance, `builtins.attrNames { y = 1; x = "foo"; }` evaluates to `[ "x" "y" ]`.

`builtins.attrValues` *`set`* Return the values of the attributes in the set *`set`* in the order corresponding to the sorted attribute names.

`baseNameOf` *`s`* Return the *base name* of the string *`s`*, that is, everything following the final slash in the string. This is similar to the GNU **basename** command.

`builtins.bitAnd` *`e1`* *`e2`* Return the bitwise AND of the integers *`e1`* and *`e2`*.

`builtins.bitOr` *`e1`* *`e2`* Return the bitwise OR of the integers *`e1`* and *`e2`*.

`builtins.bitXor` *`e1`* *`e2`* Return the bitwise XOR of the integers *`e1`* and *`e2`*.

`builtins` The set `builtins` contains all the built-in functions and values. You can use `builtins` to test for the availability of features in the Nix installation, e.g.,

```
if builtins ? getEnv then builtins.getEnv "PATH" else ""
```

This allows a Nix expression to fall back gracefully on older Nix installations that don't have the desired built-in function.

`builtins.compareVersions` *`s1`* *`s2`* Compare two strings representing versions and return `-1` if version *`s1`* is older than version *`s2`*, `0` if they are the same, and `1` if *`s1`* is newer than *`s2`*. The version comparison algorithm is the same as the one used by **nix-env -u**.

`builtins.concatLists` *`lists`* Concatenate a list of lists into a single list.

`builtins.concatStringsSep` *`separator`* *`list`* Concatenate a list of strings with a separator between each element, e.g. `concatStringsSep "/" ["usr" "local" "bin"] == "usr/local/bin"`

`builtins.currentSystem` The built-in value `currentSystem` evaluates to the Nix platform identifier for the Nix installation on which the expression is being evaluated, such as `"i686-linux"` or `"x86_64-darwin"`.

`builtins.deepSeq` *`e1`* *`e2`* This is like `seq e1` *`e2`*, except that *`e1`* is evaluated *deeply*: if it's a list or set, its elements or attributes are also evaluated recursively.

`derivation` *`attrs`*, `builtins.derivation` *`attrs`* `derivation` is described in Section 15.4, "Derivations".

`dirOf` *`s`*, `builtins.dirOf` *`s`* Return the directory part of the string *`s`*, that is, everything before the final slash in the string. This is similar to the GNU **dirname** command.

`builtins.div` *`e1`* *`e2`* Return the quotient of the numbers *`e1`* and *`e2`*.

builtins.elem *x xs* Return `true` if a value equal to *x* occurs in the list *xs*, and `false` otherwise.

builtins.elemAt *xs n* Return element *n* from the list *xs*. Elements are counted starting from 0. A fatal error occurs if the index is out of bounds.

builtins.fetchurl *url* Download the specified URL and return the path of the downloaded file. This function is not available if restricted evaluation mode is enabled.

fetchTarball *url*, **builtins.fetchTarball** *url* Download the specified URL, unpack it and return the path of the unpacked tree. The file must be a tape archive (`.tar`) compressed with `gzip`, `bzip2` or `xz`. The top-level path component of the files in the tarball is removed, so it is best if the tarball contains a single directory at top level. The typical use of the function is to obtain external Nix expression dependencies, such as a particular version of Nixpkgs, e.g.

```
with import (fetchTarball
    ↪ https://github.com/NixOS/nixpkgs-channels/archive/nixos-14.12.tar.gz) {};

stdenv.mkDerivation { ... }
```

The fetched tarball is cached for a certain amount of time (1 hour by default) in `~/.cache/nix/tarballs/`. You can change the cache timeout either on the command line with `--option tarball-ttl number of seconds` or in the Nix configuration file with this option: `tarball-ttl` *number of seconds to cache*.

Note that when obtaining the hash with `nix-prefetch-url` the option `--unpack` is required.

This function can also verify the contents against a hash. In that case, the function takes a set instead of a URL. The set requires the attribute `url` and the attribute `sha256`, e.g.

```
with import (fetchTarball {
  url = https://github.com/NixOS/nixpkgs-channels/archive/nixos-14.12.tar.gz;
  sha256 = "1jppksrfvbk5ypiqdz4cddxdl8z6zyzdb2srq8fcffr327ld5jj2";
}) {};

stdenv.mkDerivation { ... }
```

This function is not available if restricted evaluation mode is enabled.

builtins.fetchGit *args* Fetch a path from git. *args* can be a URL, in which case the HEAD of the repo at that URL is fetched. Otherwise, it can be an attribute with the following attributes (all except `url` optional):

url The URL of the repo.

name The name of the directory the repo should be exported to in the store. Defaults to the basename of the URL.

rev The git revision to fetch. Defaults to the tip of `ref`.

ref The git ref to look for the requested revision under. This is often a branch or tag name. Defaults to `HEAD`.

By default, the `ref` value is prefixed with `refs/heads/`. As of Nix 2.3.0 Nix will not prefix `refs/heads/` if `ref` starts with `refs/`.

Example 15.2. Fetching a private repository over SSH

```
builtins.fetchGit {
  url = "git@github.com:my-secret/repository.git";
  ref = "master";
  rev = "adab8b916a45068c044658c4158d81878f9ed1c3";
}
```

Example 15.3. Fetching an arbitrary ref

```
builtins.fetchGit {
  url = "https://github.com/NixOS/nix.git";
  ref = "refs/heads/0.5-release";
}
```

Example 15.4. Fetching a repository's specific commit on an arbitrary branch

If the revision you're looking for is in the default branch of the git repository you don't strictly need to specify the branch name in the `ref` attribute.

However, if the revision you're looking for is in a future branch for the non-default branch you will need to specify the the `ref` attribute as well.

```
builtins.fetchGit {
  url = "https://github.com/nixos/nix.git";
  rev = "841fcbd04755c7a2865c51c1e2d3b045976b7452";
  ref = "1.11-maintenance";
}
```

Note

It is nice to always specify the branch which a revision belongs to. Without the branch being specified, the fetcher might fail if the default branch changes. Additionally, it can be confusing to try a commit from a non-default branch and see the fetch fail. If the branch is specified the fault is much more obvious.

Example 15.5. Fetching a repository's specific commit on the default branch

If the revision you're looking for is in the default branch of the git repository you may omit the `ref` attribute.

```
builtins.fetchGit {
  url = "https://github.com/nixos/nix.git";
  rev = "841fcbd04755c7a2865c51c1e2d3b045976b7452";
}
```

Example 15.6. Fetching a tag

```
builtins.fetchGit {
  url = "https://github.com/nixos/nix.git";
  ref = "refs/tags/1.9";
}
```

Example 15.7. Fetching the latest version of a remote branch

`builtins.fetchGit` can behave impurely fetch the latest version of a remote branch.

Note

Nix will refetch the branch in accordance to `tarball-ttl`.

Note

This behavior is disabled in *Pure evaluation mode*.

```
builtins.fetchGit {
  url = "ssh://git@github.com/nixos/nix.git";
  ref = "master";
}
```

builtins.filter *f xs* Return a list consisting of the elements of *xs* for which the function *f* returns `true`.

builtins.filterSource *e1 e2* This function allows you to copy sources into the Nix store while filtering certain files. For instance, suppose that you want to use the directory `source-dir` as an input to a Nix expression, e.g.

```
stdenv.mkDerivation {
  ...
  src = ./source-dir;
}
```

However, if `source-dir` is a Subversion working copy, then all those annoying `.svn` subdirectories will also be copied to the store. Worse, the contents of those directories may change a lot, causing lots of spurious rebuilds. With `filterSource` you can filter out the `.svn` directories:

```
  src = builtins.filterSource
    (path: type: type != "directory" || baseNameOf path != ".svn")
    ./source-dir;
```

Thus, the first argument *e1* must be a predicate function that is called for each regular file, directory or symlink in the source tree *e2*. If the function returns `true`, the file is copied to the Nix store, otherwise it is omitted. The function is called with two arguments. The first is the full path of the file. The second is a string that identifies the type of the file, which is either `"regular"`, `"directory"`, `"symlink"` or `"unknown"` (for other kinds of files such as device nodes or fifos -- but note that those cannot be copied to the Nix store, so if the predicate returns `true` for them, the copy will fail). If you exclude a directory, the entire corresponding subtree of *e2* will be excluded.

builtins.foldl' *op nul list* Reduce a list by applying a binary operator, from left to right, e.g. `foldl' ↪ op nul [x0 x1 x2 ...] = op (op (op nul x0)x1)x2)...`. The operator is applied strictly, i.e., its arguments are evaluated first. For example, `foldl' (x: y: x + y)0 [1 2 3]` evaluates to 6.

builtins.functionArgs *f* Return a set containing the names of the formal arguments expected by the function *f*. The value of each attribute is a Boolean denoting whether the corresponding argument has a default value. For instance, `functionArgs ({ x, y ? 123}: ...)= { x = false; y = true; }`.

"Formal argument" here refers to the attributes pattern-matched by the function. Plain lambdas are not included, e.g. `functionArgs (x: ...)= { }`.

builtins.fromJSON *e* Convert a JSON string to a Nix value. For example,

```
builtins.fromJSON ''{"x": [1, 2, 3], "y": null}''
```

returns the value `{ x = [ 1 2 3 ]; y = null; }`.

builtins.genList *generator length* Generate list of size *length*, with each element *i* equal to the value returned by *generator* `i`. For example,

```
builtins.genList (x: x * x) 5
```

returns the list `[ 0 1 4 9 16 ]`.

builtins.getAttr *s set* `getAttr` returns the attribute named *s* from *set*. Evaluation aborts if the attribute doesn't exist. This is a dynamic version of the `.` operator, since *s* is an expression rather than an identifier.

builtins.getEnv *s* `getEnv` returns the value of the environment variable *s*, or an empty string if the variable doesn't exist. This function should be used with care, as it can introduce all sorts of nasty environment dependencies in your Nix expression.

`getEnv` is used in Nix Packages to locate the file `~/.nixpkgs/config.nix`, which contains user-local settings for Nix Packages. (That is, it does a `getEnv "HOME"` to locate the user's home directory.)

builtins.hasAttr *s set* `hasAttr` returns `true` if *set* has an attribute named *s*, and `false` otherwise. This is a dynamic version of the `?` operator, since *s* is an expression rather than an identifier.

builtins.hashString *type s* Return a base-16 representation of the cryptographic hash of string *s*. The hash algorithm specified by *type* must be one of `"md5"`, `"sha1"`, `"sha256"` or `"sha512"`.

builtins.hashFile *type p* Return a base-16 representation of the cryptographic hash of the file at path *p*. The hash algorithm specified by *type* must be one of `"md5"`, `"sha1"`, `"sha256"` or `"sha512"`.

builtins.head *list* Return the first element of a list; abort evaluation if the argument isn't a list or is an empty list. You can test whether a list is empty by comparing it with `[]`.

import *path*, **builtins.import** *path* Load, parse and return the Nix expression in the file *path*. If *path* is a directory, the file `default.nix` in that directory is loaded. Evaluation aborts if the file doesn't exist or contains an incorrect Nix expression. `import` implements Nix's module system: you can put any Nix expression (such as a set or a function) in a separate file, and use it from Nix expressions in other files.

Note

Unlike some languages, `import` is a regular function in Nix. Paths using the angle bracket syntax (e.g., `import <foo>`) are normal path values (see Section 15.1, "Values").

A Nix expression loaded by `import` must not contain any *free variables* (identifiers that are not defined in the Nix expression itself and are not built-in). Therefore, it cannot refer to variables that are in scope at the call site. For instance, if you have a calling expression

```
rec {
  x = 123;
  y = import ./foo.nix;
}
```

then the following `foo.nix` will give an error:

```
x + 456
```

since `x` is not in scope in `foo.nix`. If you want `x` to be available in `foo.nix`, you should pass it as a function argument:

```
rec {
  x = 123;
  y = import ./foo.nix x;
}
```

and

```
x: x + 456
```

(The function argument doesn't have to be called `x` in `foo.nix`; any name would work.)

builtins.intersectAttrs *e1 e2* Return a set consisting of the attributes in the set *e2* that also exist in the set *e1*.

builtins.isAttrs *e* Return `true` if *e* evaluates to a set, and `false` otherwise.

builtins.isList *e* Return `true` if *e* evaluates to a list, and `false` otherwise.

builtins.isFunction *e* Return `true` if *e* evaluates to a function, and `false` otherwise.

builtins.isString *e* Return `true` if *e* evaluates to a string, and `false` otherwise.

builtins.isInt *e* Return `true` if *e* evaluates to an int, and `false` otherwise.

builtins.isFloat *e* Return `true` if *e* evaluates to a float, and `false` otherwise.

builtins.isBool *e* Return `true` if *e* evaluates to a bool, and `false` otherwise.

builtins.isPath *e* Return `true` if *e* evaluates to a path, and `false` otherwise.

isNull *e*, builtins.isNull *e* Return `true` if *e* evaluates to `null`, and `false` otherwise.

Warning

This function is *deprecated*; just write `e == null` instead.

builtins.length *e* Return the length of the list *e*.

builtins.lessThan *e1 e2* Return `true` if the number *e1* is less than the number *e2*, and `false` otherwise. Evaluation aborts if either *e1* or *e2* does not evaluate to a number.

builtins.listToAttrs *e* Construct a set from a list specifying the names and values of each attribute. Each element of the list should be a set consisting of a string-valued attribute `name` specifying the name of the attribute, and an attribute `value` specifying its value. Example:

```
builtins.listToAttrs
  [ { name = "foo"; value = 123; }
    { name = "bar"; value = 456; }
  ]
```

evaluates to

```
{ foo = 123; bar = 456; }
```

map *f list*, builtins.map *f list* Apply the function *f* to each element in the list *list*. For example,

```
map (x: "foo" + x) [ "bar" "bla" "abc" ]
```

evaluates to `[ "foobar" "foobla" "fooabc" ]`.

builtins.match *regex str* Returns a list if the extended POSIX regular expression *regex* matches *str* precisely, otherwise returns `null`. Each item in the list is a regex group.

```
builtins.match "ab" "abc"
```

Evaluates to `null`.

```
builtins.match "abc" "abc"
```

Evaluates to `[ ]`.

```
builtins.match "a(b)(c)" "abc"
```

Evaluates to `[ "b" "c" ]`.

```
builtins.match "[[:space:]]+([[:upper:]]+)[[:space:]]+" "  FOO   "
```

Evaluates to `[ "foo" ]`.

builtins.mul *e1 e2* Return the product of the numbers *e1* and *e2*.

builtins.parseDrvName *s* Split the string *s* into a package name and version. The package name is everything up to but not including the first dash followed by a digit, and the version is everything following that dash. The result is returned in a set `{ name, version }`. Thus, `builtins.parseDrvName "nix-0.12pre12876"` returns `{ name = "nix"; version = "0.12pre12876"; }`.

builtins.path *args* An enrichment of the built-in path type, based on the attributes present in *args*. All are optional except `path`:

path The underlying path.

name The name of the path when added to the store. This can used to reference paths that have nix-illegal characters in their names, like `@`.

filter A function of the type expected by builtins.filterSource, with the same semantics.

recursive When `false`, when `path` is added to the store it is with a flat hash, rather than a hash of the NAR serialization of the file. Thus, `path` must refer to a regular file, not a directory. This allows similar behavior to `fetchurl`. Defaults to `true`.

sha256 When provided, this is the expected hash of the file at the path. Evaluation will fail if the hash is incorrect, and providing a hash allows `builtins.path` to be used even when the `pure-eval` nix config option is on.

builtins.pathExists *path* Return `true` if the path *path* exists at evaluation time, and `false` otherwise.

builtins.placeholder *output* Return a placeholder string for the specified *output* that will be substituted by the corresponding output path at build time. Typical outputs would be `"out"`, `"bin"` or `"dev"`.

builtins.readDir *path* Return the contents of the directory *path* as a set mapping directory entries to the corresponding file type. For instance, if directory `A` contains a regular file `B` and another directory `C`, then `builtins.readDir ./A` will return the set

```
{ B = "regular"; C = "directory"; }
```

The possible values for the file type are `"regular"`, `"directory"`, `"symlink"` and `"unknown"`.

builtins.readFile *path* Return the contents of the file *path* as a string.

removeAttrs *set list*, builtins.removeAttrs *set list* Remove the attributes listed in *list* from *set*. The attributes don't have to exist in *set*. For instance,

```
removeAttrs { x = 1; y = 2; z = 3; } [ "a" "x" "z" ]
```

evaluates to `{ y = 2; }`.

builtins.replaceStrings *from to s* Given string *s*, replace every occurrence of the strings in *from* with the corresponding string in *to*. For example,

```
builtins.replaceStrings ["oo" "a"] ["a" "i"] "foobar"
```

evaluates to `"fabir"`.

builtins.seq *e1 e2* Evaluate *e1*, then evaluate and return *e2*. This ensures that a computation is strict in the value of *e1*.

builtins.sort *comparator list* Return *list* in sorted order. It repeatedly calls the function *comparator* with two elements. The comparator should return `true` if the first element is less than the second, and `false` otherwise. For example,

```
builtins.sort builtins.lessThan [ 483 249 526 147 42 77 ]
```

produces the list `[ 42 77 147 249 483 526 ]`.

This is a stable sort: it preserves the relative order of elements deemed equal by the comparator.

builtins.split *regex str* Returns a list composed of non matched strings interleaved with the lists of the extended POSIX regular expression *regex* matches of *str*. Each item in the lists of matched sequences is a regex group.

```
builtins.split "(a)b" "abc"
```

Evaluates to `[ "" [ "a" ] "c" ]`.

```
builtins.split "([ac])" "abc"
```

Evaluates to `[ "" [ "a" ] "b" [ "c" ] "" ]`.

```
builtins.split "(a)|(c)" "abc"
```

Evaluates to `[ "" [ "a" null ] "b" [ null "c" ] "" ]`.

```
builtins.split "([[:upper:]]+)" "  FOO   "
```

Evaluates to `[ " " [ "FOO" ] " " ]`.

builtins.splitVersion *s* Split a string representing a version into its components, by the same version splitting logic underlying the version comparison in **nix-env -u**.

builtins.stringLength *e* Return the length of the string *e*. If *e* is not a string, evaluation is aborted.

builtins.sub *e1 e2* Return the difference between the numbers *e1* and *e2*.

builtins.substring *start len s* Return the substring of *s* from character position *start* (zero-based) up to but not including *start + len*. If *start* is greater than the length of the string, an empty string is returned, and if *start + len* lies beyond the end of the string, only the substring up to the end of the string is returned. *start* must be non-negative. For example,

```
builtins.substring 0 3 "nixos"
```

evaluates to `"nix"`.

builtins.tail *list* Return the second to last elements of a list; abort evaluation if the argument isn't a list or is an empty list.

throw *s*, **builtins.throw** *s* Throw an error message *s*. This usually aborts Nix expression evaluation, but in **nix-env -qa** and other commands that try to evaluate a set of derivations to get information about those derivations, a derivation that throws an error is silently skipped (which is not the case for `abort`).

builtins.toFile *name s* Store the string *s* in a file in the Nix store and return its path. The file has suffix *name*. This file can be used as an input to derivations. One application is to write builders "inline". For instance, the following Nix expression combines Example 14.1, "Nix expression for GNU Hello (`default.nix`)" and Example 14.2, "Build script for GNU Hello (`builder.sh`)" into one file:

```
{ stdenv, fetchurl, perl }:

stdenv.mkDerivation {
  name = "hello-2.1.1";

  builder = builtins.toFile "builder.sh" "
    source $stdenv/setup

    PATH=$perl/bin:$PATH

    tar xvfz $src
    cd hello-*
    ./configure --prefix=$out
    make
    make install
  ";

  src = fetchurl {
    url = http://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
    sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
  };
  inherit perl;
}
```

It is even possible for one file to refer to another, e.g.,

```
builder = let
  configFile = builtins.toFile "foo.conf" "
    # This is some dummy configuration file.
    ...
  ";
in builtins.toFile "builder.sh" "
  source $stdenv/setup
  ...
  cp ${configFile} $out/etc/foo.conf
";
```

Note that `${configFile}` is an antiquotation (see Section 15.1, "Values"), so the result of the expression `configFile` (i.e., a path like `/nix/store/m7p7jfny445k...-foo.conf`) will be spliced into the resulting string.

It is however *not* allowed to have files mutually referring to each other, like so:

```
let
  foo = builtins.toFile "foo" "...${bar}...";
  bar = builtins.toFile "bar" "...${foo}...";
in foo
```

This is not allowed because it would cause a cyclic dependency in the computation of the cryptographic hashes for `foo` and `bar`.

It is also not possible to reference the result of a derivation. If you are using Nixpkgs, the `writeTextFile` function is able to do that.

`builtins.toJSON` ***e*** Return a string containing a JSON representation of *e*. Strings, integers, floats, booleans, nulls and lists are mapped to their JSON equivalents. Sets (except derivations) are represented as objects. Derivations are translated to a JSON string containing the derivation's output path. Paths are copied to the store and represented as a JSON string of the resulting store path.

`builtins.toPath` ***s*** DEPRECATED. Use `/. + "/path"` to convert a string into an absolute path. For relative paths, use `./. + "/path"`.

`toString` ***e*****, `builtins.toString`** ***e*** Convert the expression *e* to a string. *e* can be:

- A string (in which case the string is returned unmodified).
- A path (e.g., `toString /foo/bar` yields `"/foo/bar"`.
- A set containing `{ __toString = self: ...; }`.
- An integer.
- A list, in which case the string representations of its elements are joined with spaces.
- A Boolean (`false` yields `""`, `true` yields `"1"`).
- `null`, which yields the empty string.

`builtins.toXML` ***e*** Return a string containing an XML representation of *e*. The main application for `toXML` is to communicate information with the builder in a more structured format than plain environment variables.

Example 15.8, "Passing information to a builder using `toXML`" shows an example where this is the case. The builder is supposed to generate the configuration file for a Jetty servlet container. A servlet container contains a number of servlets (`*.war` files) each exported under a specific URI prefix. So the servlet configuration is a list of sets containing the `path` and `war` of the servlet (❸). This kind of information is difficult to communicate with the normal method of passing information through an environment variable, which just concatenates everything together into a string (which might just work in this case, but wouldn't work if fields are optional or contain lists themselves). Instead the Nix expression is converted to an XML representation with `toXML`, which is unambiguous and can easily be processed with the appropriate tools. For instance, in the example an XSLT stylesheet (❷) is applied to it (❶) to generate the XML configuration file for the Jetty server. The

XML representation produced from ❸ by `toXML` is shown in Example 15.9, "XML representation produced by `toXML`".

Note that Example 15.8, "Passing information to a builder using `toXML`" uses the `toFile` built-in to write the builder and the stylesheet "inline" in the Nix expression. The path of the stylesheet is spliced into the builder at `xsltproc ${stylesheet} ...`.

Example 15.8. Passing information to a builder using `toXML`

```
{ stdenv, fetchurl, libxslt, jira, uberwiki }:

stdenv.mkDerivation (rec {
  name = "web-server";

  buildInputs = [ libxslt ];

  builder = builtins.toFile "builder.sh" "
    source $stdenv/setup
    mkdir $out
    echo "$servlets" | xsltproc ${stylesheet} - > $out/server-conf.xml
  ";

  stylesheet = builtins.toFile "stylesheet.xsl"
   "<?xml version='1.0' encoding='UTF-8'?>
    <xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>
      <xsl:template match='/'>
        <Configure>
          <xsl:for-each select='/expr/list/attrs'>
            <Call name='addWebApplication'>
              <Arg><xsl:value-of select=\"attr[@name = 'path']/string/@value\"
                 ↪ /></Arg>
              <Arg><xsl:value-of select=\"attr[@name = 'war']/path/@value\"
                 ↪ /></Arg>
            </Call>
          </xsl:for-each>
        </Configure>
      </xsl:template>
    </xsl:stylesheet>
  ";

  servlets = builtins.toXML [
    { path = "/bugtracker"; war = jira + "/lib/atlassian-jira.war"; }
    { path = "/wiki"; war = uberwiki + "/uberwiki.war"; }
  ];
})
```

Example 15.9. XML representation produced by `toXML`

```
<?xml version='1.0' encoding='utf-8'?>
<expr>
  <list>
    <attrs>
      <attr name="path">
        <string value="/bugtracker" />
      </attr>
      <attr name="war">
```

```
        <path value="/nix/store/d1jh9pasa7k2...-jira/lib/atlassian-jira.war" />
      </attr>
    </attrs>
    <attrs>
      <attr name="path">
        <string value="/wiki" />
      </attr>
      <attr name="war">
        <path value="/nix/store/y6423b1yi4sx...-uberwiki/uberwiki.war" />
      </attr>
    </attrs>
  </list>
</expr>
```

builtins.trace *e1 e2* Evaluate *e1* and print its abstract syntax representation on standard error. Then return *e2*. This function is useful for debugging.

builtins.tryEval *e* Try to shallowly evaluate *e*. Return a set containing the attributes `success` (`true` if *e* evaluated successfully, `false` if an error was thrown) and `value`, equalling *e* if successful and `false` otherwise. Note that this doesn't evaluate *e* deeply, so `let e = { x = throw ""; }; in (builtins.tryEval e).success` will be `true`. Using `builtins.deepSeq` one can get the expected result: `let e = { x = throw ""; }; in (builtins.tryEval (builtins.deepSeq e e)).success` will be `false`.

builtins.typeOf *e* Return a string representing the type of the value *e*, namely `"int"`, `"bool"`, `"string"`, `"path"`, `"null"`, `"set"`, `"list"`, `"lambda"` or `"float"`.

---

[5] It's parsed as an expression that selects the attribute `sh` from the variable `builder`.

[6] Actually, Nix detects infinite recursion in this case and aborts ("infinite recursion encountered").

[7] To figure out your platform identifier, look at the line "Checking for the canonical Nix system name" in the output of Nix's `configure` script.

Part V. Advanced Topics

Chapter 16. Remote Builds

Nix supports remote builds, where a local Nix installation can forward Nix builds to other machines. This allows multiple builds to be performed in parallel and allows Nix to perform multi-platform builds in a semi-transparent way. For instance, if you perform a build for a `x86_64-darwin` on an `i686-linux` machine, Nix can automatically forward the build to a `x86_64-darwin` machine, if available.

To forward a build to a remote machine, it's required that the remote machine is accessible via SSH and that it has Nix installed. You can test whether connecting to the remote Nix instance works, e.g.

```
$ nix ping-store --store ssh://mac
```

will try to connect to the machine named `mac`. It is possible to specify an SSH identity file as part of the remote store URI, e.g.

```
$ nix ping-store --store ssh://mac?ssh-key=/home/alice/my-key
```

Since builds should be non-interactive, the key should not have a passphrase. Alternatively, you can load identities ahead of time into **ssh-agent** or **gpg-agent**.

If you get the error

```
bash: nix-store: command not found
error: cannot connect to 'mac'
```

then you need to ensure that the `PATH` of non-interactive login shells contains Nix.

Warning

If you are building via the Nix daemon, it is the Nix daemon user account (that is, `root`) that should have SSH access to the remote machine. If you can't or don't want to configure `root` to be able to access to remote machine, you can use a private Nix store instead by passing e.g. `--store ~/my-nix`.

The list of remote machines can be specified on the command line or in the Nix configuration file. The former is convenient for testing. For example, the following command allows you to build a derivation for `x86_64-darwin` on a Linux machine:

```
$ uname
Linux

$ nix build \
  '(with import <nixpkgs> { system = "x86_64-darwin"; }; runCommand "foo" {}
     ↪ "uname > $out")' \
  --builders 'ssh://mac x86_64-darwin'
[1/0/1 built, 0.0 MiB DL] building foo on ssh://mac

$ cat ./result
Darwin
```

It is possible to specify multiple builders separated by a semicolon or a newline, e.g.

```
  --builders 'ssh://mac x86_64-darwin ; ssh://beastie x86_64-freebsd'
```

Each machine specification consists of the following elements, separated by spaces. Only the first element is required. To leave a field at its default, set it to `-`.

1. The URI of the remote store in the format `ssh://[username@]`*hostname*, e.g. `ssh://nix@mac` or `ssh://mac`. For backward compatibility, `ssh://` may be omitted. The hostname may be an alias defined in your `~/.ssh/config`.
2. A comma-separated list of Nix platform type identifiers, such as `x86_64-darwin`. It is possible for a machine to support multiple platform types, e.g., `i686-linux,x86_64-linux`. If omitted, this defaults to the local platform type.
3. The SSH identity file to be used to log in to the remote machine. If omitted, SSH will use its regular identities.
4. The maximum number of builds that Nix will execute in parallel on the machine. Typically this should be equal to the number of CPU cores. For instance, the machine `itchy` in the example will execute up to 8 builds in parallel.
5. The "speed factor", indicating the relative speed of the machine. If there are multiple machines of the right type, Nix will prefer the fastest, taking load into account.
6. A comma-separated list of *supported features*. If a derivation has the `requiredSystemFeatures` attribute, then Nix will only perform the derivation on a machine that has the specified features. For instance, the attribute

   ```
   requiredSystemFeatures = [ "kvm" ];
   ```

   will cause the build to be performed on a machine that has the `kvm` feature.

7. A comma-separated list of *mandatory features*. A machine will only be used to build a derivation if all of the machine's mandatory features appear in the derivation's `requiredSystemFeatures` attribute..

For example, the machine specification

```
nix@scratchy.labs.cs.uu.nl  i686-linux      /home/nix/.ssh/id_scratchy_auto
    ↪ 8 1 kvm
nix@itchy.labs.cs.uu.nl     i686-linux      /home/nix/.ssh/id_scratchy_auto
    ↪ 8 2
nix@poochie.labs.cs.uu.nl   i686-linux      /home/nix/.ssh/id_scratchy_auto
    ↪ 1 2 kvm benchmark
```

specifies several machines that can perform `i686-linux` builds. However, `poochie` will only do builds that have the attribute

```
requiredSystemFeatures = [ "benchmark" ];
```

or

```
requiredSystemFeatures = [ "benchmark" "kvm" ];
```

`itchy` cannot do builds that require `kvm`, but `scratchy` does support such builds. For regular builds, `itchy` will be preferred over `scratchy` because it has a higher speed factor.

Remote builders can also be configured in `nix.conf`, e.g.

```
builders = ssh://mac x86_64-darwin ; ssh://beastie x86_64-freebsd
```

Finally, remote builders can be configured in a separate configuration file included in `builders` via the syntax `@file`. For example,

```
builders = @/etc/nix/machines
```

causes the list of machines in `/etc/nix/machines` to be included. (This is the default.)

If you want the builders to use caches, you likely want to set the option `builders-use-substitutes` in your local `nix.conf`.

To build only on remote builders and disable building on the local machine, you can use the option `--max-jobs 0`.

Chapter 17. Tuning Cores and Jobs

Nix has two relevant settings with regards to how your CPU cores will be utilized: `cores` and `max-jobs`. This chapter will talk about what they are, how they interact, and their configuration trade-offs.

`max-jobs` Dictates how many separate derivations will be built at the same time. If you set this to zero, the local machine will do no builds. Nix will still substitute from binary caches, and build remotely if remote builders are configured.

`cores` Suggests how many cores each derivation should use. Similar to **make -j**.

The `cores` setting determines the value of `NIX_BUILD_CORES`. `NIX_BUILD_CORES` is equal to `cores`, unless `cores` equals `0`, in which case `NIX_BUILD_CORES` will be the total number of cores in the system.

The total number of consumed cores is a simple multiplication, `cores` * `NIX_BUILD_CORES`.

The balance on how to set these two independent variables depends upon each builder's workload and hardware. Here are a few example scenarios on a machine with 24 cores:

Table 17.1. Balancing 24 Build Cores

max-jobs	cores	NIX_BUILD_CORES	Maximum Processes	Result
1	24	24	24	One derivation will be built at a time, each one can use 24 cor
4	6	6	24	Four derivations will be built at once, each given access to six
12	6	6	72	12 derivations will be built at once, each given access to six co
24	1	1	24	24 derivations can build at the same time, each using a single
24	0	24	576	24 derivations can build at the same time, each using all the a

It is up to the derivations' build script to respect host's requested cores-per-build by following the value of the `NIX_BUILD_CORES` environment variable.

Chapter 18. Verifying Build Reproducibility with `diff-hook`

Check build reproducibility by running builds multiple times and comparing their results.

Specify a program with Nix's `diff-hook` to compare build results when two builds produce different results. Note: this hook is only executed if the results are not the same, this hook is not used for determining if the results are the same.

For purposes of demonstration, we'll use the following Nix file, `deterministic.nix` for testing:

```
let
  inherit (import <nixpkgs> {}) runCommand;
in {
  stable = runCommand "stable" {} ''
    touch $out
  '';

  unstable = runCommand "unstable" {} ''
    echo $RANDOM > $out
  '';
}
```

Additionally, `nix.conf` contains:

```
diff-hook = /etc/nix/my-diff-hook
run-diff-hook = true
```

where `/etc/nix/my-diff-hook` is an executable file containing:

```
#!/bin/sh
exec >&2
echo "For derivation $3:"
/run/current-system/sw/bin/diff -r "$1" "$2"
```

The diff hook is executed by the same user and group who ran the build. However, the diff hook does not have write access to the store path just built.

18.1. Spot-Checking Build Determinism

Verify a path which already exists in the Nix store by passing `--check` to the build command.

If the build passes and is deterministic, Nix will exit with a status code of 0:

```
$ nix-build ./deterministic.nix -A stable
these derivations will be built:
  /nix/store/z98fasz2jqy9gs0xbvdj939p27jwda38-stable.drv
```

```
building '/nix/store/z98fasz2jqy9gs0xbvdj939p27jwda38-stable.drv'...
/nix/store/yyxlzw3vqaas7wfp04g0b1xg51f2czgq-stable

$ nix-build ./deterministic.nix -A stable --check
checking outputs of '/nix/store/z98fasz2jqy9gs0xbvdj939p27jwda38-stable.drv'...
/nix/store/yyxlzw3vqaas7wfp04g0b1xg51f2czgq-stable
```

If the build is not deterministic, Nix will exit with a status code of 1:

```
$ nix-build ./deterministic.nix -A unstable
these derivations will be built:
  /nix/store/cgl13lbj1w368r5z8gywipl1ifli7dhk-unstable.drv
building '/nix/store/cgl13lbj1w368r5z8gywipl1ifli7dhk-unstable.drv'...
/nix/store/krpqk0l9ib0ibi1d2w52z293zw455cap-unstable

$ nix-build ./deterministic.nix -A unstable --check
checking outputs of '/nix/store/cgl13lbj1w368r5z8gywipl1ifli7dhk-unstable.drv'...
error: derivation '/nix/store/cgl13lbj1w368r5z8gywipl1ifli7dhk-unstable.drv' may
   ↪ not be deterministic: output
   ↪ '/nix/store/krpqk0l9ib0ibi1d2w52z293zw455cap-unstable' differs
```

In the Nix daemon's log, we will now see:

```
For derivation /nix/store/cgl13lbj1w368r5z8gywipl1ifli7dhk-unstable.drv:
1c1
< 8108
---
> 30204
```

Using `--check` with `--keep-failed` will cause Nix to keep the second build's output in a special, `.check` path:

```
$ nix-build ./deterministic.nix -A unstable --check --keep-failed
checking outputs of '/nix/store/cgl13lbj1w368r5z8gywipl1ifli7dhk-unstable.drv'...
note: keeping build directory '/tmp/nix-build-unstable.drv-0'
error: derivation '/nix/store/cgl13lbj1w368r5z8gywipl1ifli7dhk-unstable.drv' may
   ↪ not be deterministic: output
   ↪ '/nix/store/krpqk0l9ib0ibi1d2w52z293zw455cap-unstable' differs from
   ↪ '/nix/store/krpqk0l9ib0ibi1d2w52z293zw455cap-unstable.check'
```

In particular, notice the `/nix/store/krpqk0l9ib0ibi1d2w52z293zw455cap-unstable.check` output. Nix has copied the build results to that directory where you can examine it.

`.check` paths are not registered store paths

Check paths are not protected against garbage collection, and this path will be deleted on the next garbage collection.

The path is guaranteed to be alive for the duration of `diff-hook`'s execution, but may be deleted any time after.

If the comparison is performed as part of automated tooling, please use the diff-hook or author your tooling to handle the case where the build was not deterministic and also a check path does not exist.

`--check` is only usable if the derivation has been built on the system already. If the derivation has not been built Nix will fail with the error:

```
error: some outputs of '/nix/store/hzi1h60z2qf0nb85iwnpvrai3j2w7rr6-unstable.drv'
   ↪ are not valid, so checking is not possible
```

Run the build without `--check`, and then try with `--check` again.

18.2. Automatic and Optionally Enforced Determinism Verification

Automatically verify every build at build time by executing the build multiple times.

Setting `repeat` and `enforce-determinism` in your `nix.conf` permits the automated verification of every build Nix performs.

The following configuration will run each build three times, and will require the build to be deterministic:

```
enforce-determinism = true
repeat = 2
```

Setting `enforce-determinism` to false as in the following configuration will run the build multiple times, execute the build hook, but will allow the build to succeed even if it does not build reproducibly:

```
enforce-determinism = false
repeat = 1
```

An example output of this configuration:

```
$ nix-build ./test.nix -A unstable
these derivations will be built:
  /nix/store/ch6llwpr2h8c3jmnf3f2ghkhx59aa97f-unstable.drv
building '/nix/store/ch6llwpr2h8c3jmnf3f2ghkhx59aa97f-unstable.drv' (round 1/2)...
building '/nix/store/ch6llwpr2h8c3jmnf3f2ghkhx59aa97f-unstable.drv' (round 2/2)...
output '/nix/store/6xg356v9gl03hpbbg8gws77n19qanh02-unstable' of
   ↪ '/nix/store/ch6llwpr2h8c3jmnf3f2ghkhx59aa97f-unstable.drv' differs from
   ↪ '/nix/store/6xg356v9gl03hpbbg8gws77n19qanh02-unstable.check' from previous
   ↪ round
/nix/store/6xg356v9gl03hpbbg8gws77n19qanh02-unstable
```

Chapter 19. Using the `post-build-hook`

Uploading to an S3-compatible binary cache after each build

19.1. Implementation Caveats

Here we use the post-build hook to upload to a binary cache. This is a simple and working example, but it is not suitable for all use cases.

The post build hook program runs after each executed build, and blocks the build loop. The build loop exits if the hook program fails.

Concretely, this implementation will make Nix slow or unusable when the internet is slow or unreliable.

A more advanced implementation might pass the store paths to a user-supplied daemon or queue for processing the store paths outside of the build loop.

19.2. Prerequisites

This tutorial assumes you have configured an S3-compatible binary cache according to the instructions at Section 13.4.3, "Authenticated Writes to your S3-compatible binary cache", and that the `root` user's default AWS profile can upload to the bucket.

19.3. Set up a Signing Key

Use **nix-store --generate-binary-cache-key** to create our public and private signing keys. We will sign paths with the private key, and distribute the public key for verifying the authenticity of the paths.

```
# nix-store --generate-binary-cache-key example-nix-cache-1 /etc/nix/key.private
   ↪ /etc/nix/key.public
# cat /etc/nix/key.public
```

```
example-nix-cache-1:1/cKDz3QCCOmwcztD2eV6Coggp6rqc9DGjWv7C0G+rM=
```

Then, add the public key and the cache URL to your `nix.conf`'s `trusted-public-keys` and `substituters` like:

```
substituters = https://cache.nixos.org/ s3://example-nix-cache
trusted-public-keys =
    ↪ cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY=
    ↪ example-nix-cache-1:1/cKDz3QCCOmwcztD2eV6Coggp6rqc9DGjWv7C0G+rM=
```

we will restart the Nix daemon a later step.

19.4. Implementing the build hook

Write the following script to `/etc/nix/upload-to-cache.sh`:

```
#!/bin/sh

set -eu
set -f # disable globbing
export IFS=' '

echo "Signing paths" $OUT_PATHS
nix sign-paths --key-file /etc/nix/key.private $OUT_PATHS
echo "Uploading paths" $OUT_PATHS
exec nix copy --to 's3://example-nix-cache' $OUT_PATHS
```

Should `$OUT_PATHS` be quoted?

The `$OUT_PATHS` variable is a space-separated list of Nix store paths. In this case, we expect and want the shell to perform word splitting to make each output path its own argument to **nix sign-paths**. Nix guarantees the paths will not contain any spaces, however a store path might contain glob characters. The **set -f** disables globbing in the shell.

Then make sure the hook program is executable by the `root` user:

```
# chmod +x /etc/nix/upload-to-cache.sh
```

19.5. Updating Nix Configuration

Edit `/etc/nix/nix.conf` to run our hook, by adding the following configuration snippet at the end:

```
post-build-hook = /etc/nix/upload-to-cache.sh
```

Then, restart the **nix-daemon**.

19.6. Testing

Build any derivation, for example:

```
$ nix-build -E '(import <nixpkgs> {}).writeText "example" (builtins.toString
    ↪ builtins.currentTime)'
these derivations will be built:
  /nix/store/s4pnfbkalzy5qz57qs6yybna8wylkig6-example.drv
building '/nix/store/s4pnfbkalzy5qz57qs6yybna8wylkig6-example.drv'...
running post-build-hook
    ↪ '/home/grahamc/projects/github.com/NixOS/nix/post-hook.sh'...
post-build-hook: Signing paths /nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
post-build-hook: Uploading paths
    ↪ /nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
/nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
```

Then delete the path from the store, and try substituting it from the binary cache:

```
$ rm ./result
$ nix-store --delete /nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
```

Now, copy the path back from the cache:

```
$ nix store --realize /nix/store/ibcyipq5gf91838ldx40mjsp0b8w9n18-example
copying path '/nix/store/m8bmqwrch6l3h8s0k3d673xpmipcdpsa-example from
    ↪ 's3://example-nix-cache'...
warning: you did not specify '--add-root'; the result might be removed by the
    ↪ garbage collector
/nix/store/m8bmqwrch6l3h8s0k3d673xpmipcdpsa-example
```

19.7. Conclusion

We now have a Nix installation configured to automatically sign and upload every local build to a remote binary cache.

Before deploying this to production, be sure to consider the implementation caveats in Section 19.1, "Implementation Caveats".

Part VI. Command Reference

This section lists commands and options that you can use when you work with Nix.

Chapter 20. Common Options

Most Nix commands accept the following command-line options:

`--help` Prints out a summary of the command syntax and exits.

`--version` Prints out the Nix version number on standard output and exits.

`--verbose` / `-v` Increases the level of verbosity of diagnostic messages printed on standard error. For each Nix operation, the information printed on standard output is well-defined; any diagnostic information is printed on standard error, never on standard output.

This option may be specified repeatedly. Currently, the following verbosity levels exist:

0 "Errors only": only print messages explaining why the Nix invocation failed.

1 "Informational": print *useful* messages about what Nix is doing. This is the default.

2 "Talkative": print more informational messages.

3 "Chatty": print even more informational messages.

4 "Debug": print debug information.

5 "Vomit": print vast amounts of debug information.

`--quiet` Decreases the level of verbosity of diagnostic messages printed on standard error. This is the inverse option to `-v` / `--verbose`.

This option may be specified repeatedly. See the previous verbosity levels list.

`--no-build-output` / `-Q` By default, output written by builders to standard output and standard error is echoed to the Nix command's standard error. This option suppresses this behaviour. Note that the builder's standard output and error are always written to a log file in `prefix`/nix/var/log/nix.

`--max-jobs` / `-j` *number* Sets the maximum number of build jobs that Nix will perform in parallel to the specified number. Specify `auto` to use the number of CPUs in the system. The default is specified by the `max-jobs` configuration setting, which itself defaults to `1`. A higher value is useful on SMP systems or to exploit I/O latency.

Setting it to `0` disallows building on the local machine, which is useful when you want builds to happen only on remote builders.

`--cores` Sets the value of the `NIX_BUILD_CORES` environment variable in the invocation of builders. Builders can use this variable at their discretion to control the maximum amount of parallelism. For instance, in Nixpkgs, if the derivation attribute `enableParallelBuilding` is set to `true`, the builder passes the `-jN` flag to GNU Make. It defaults to the value of the `cores` configuration setting, if set, or `1` otherwise. The value `0` means that the builder should use all available CPU cores in the system.

`--max-silent-time` Sets the maximum number of seconds that a builder can go without producing any data on standard output or standard error. The default is specified by the `max-silent-time` configuration setting. `0` means no time-out.

`--timeout` Sets the maximum number of seconds that a builder can run. The default is specified by the `timeout` configuration setting. `0` means no timeout.

`--keep-going` / `-k` Keep going in case of failed builds, to the greatest extent possible. That is, if building an input of some derivation fails, Nix will still build the other inputs, but not the derivation itself. Without this option, Nix stops if any build fails (except for builds of substitutes), possibly killing builds in progress (in case of parallel or distributed builds).

`--keep-failed` / `-K` Specifies that in case of a build failure, the temporary directory (usually in `/tmp`) in which the build takes place should not be deleted. The path of the build directory is printed as an informational message.

`--fallback` Whenever Nix attempts to build a derivation for which substitutes are known for each output path, but realising the output paths through the substitutes fails, fall back on building the derivation.

The most common scenario in which this is useful is when we have registered substitutes in order to perform binary distribution from, say, a network repository. If the repository is down, the realisation of the derivation will fail. When this option is specified, Nix will build the derivation instead. Thus, installation from binaries falls back on installation from source. This option is not the default since it is generally not desirable for a transient failure in obtaining the substitutes to lead to a full build from source (with the related consumption of resources).

`--no-build-hook` Disables the build hook mechanism. This allows to ignore remote builders if they are setup on the machine.

It's useful in cases where the bandwidth between the client and the remote builder is too low. In that case it can take more time to upload the sources to the remote builder and fetch back the result than to do the computation locally.

`--readonly-mode` When this option is used, no attempt is made to open the Nix database. Most Nix operations do need database access, so those operations will fail.

`--arg` *name value* This option is accepted by **nix-env**, **nix-instantiate** and **nix-build**. When evaluating Nix expressions, the expression evaluator will automatically try to call functions that it encounters. It can automatically call functions for which every argument has a default value (e.g., `{ argName ?` *defaultValue* `}: ...`). With `--arg`, you can also call functions that have arguments without a default value (or override a default value). That is, if the evaluator encounters a function with an argument named *name*, it will call it with value *value*.

For instance, the top-level `default.nix` in Nixpkgs is actually a function:

```
{ # The system (e.g., `i686-linux') for which to build the packages.
  system ? builtins.currentSystem
  ...
}: ...
```

So if you call this Nix expression (e.g., when you do `nix-env -i pkgname`), the function will be called automatically using the value `builtins.currentSystem` for the `system` argument. You can override this using `--arg`, e.g., `nix-env -i pkgname` --arg system \"i686-freebsd\". (Note that since the argument is a Nix string literal, you have to escape the quotes.)

`--argstr` ***name value*** This option is like `--arg`, only the value is not a Nix expression but a string. So instead of `--arg system \"i686-linux\"` (the outer quotes are to keep the shell happy) you can say `--argstr system i686-linux`.

`--attr` / `-A` ***attrPath*** Select an attribute from the top-level Nix expression being evaluated. (**nix-env**, **nix-instantiate**, **nix-build** and **nix-shell** only.) The *attribute path* ***attrPath*** is a sequence of attribute names separated by dots. For instance, given a top-level Nix expression ***e***, the attribute path `xorg.xorgserver` would cause the expression `e`.xorg.xorgserver to be used. See **nix-env --install** for some concrete examples.

In addition to attribute names, you can also specify array indices. For instance, the attribute path `foo.3.bar` selects the `bar` attribute of the fourth element of the array in the `foo` attribute of the top-level expression.

`--expr` / `-E` Interpret the command line arguments as a list of Nix expressions to be parsed and evaluated, rather than as a list of file names of Nix expressions. (**nix-instantiate**, **nix-build** and **nix-shell** only.)

`-I` ***path*** Add a path to the Nix expression search path. This option may be given multiple times. See the `NIX_PATH` environment variable for information on the semantics of the Nix search path. Paths added through `-I` take precedence over `NIX_PATH`.

`--option` ***name value*** Set the Nix configuration option ***name*** to ***value***. This overrides settings in the Nix configuration file (see nix.conf(5)).

`--repair` Fix corrupted or missing store paths by redownloading or rebuilding them. Note that this is slow because it requires computing a cryptographic hash of the contents of every path in the closure of the build. Also note the warning under **nix-store --repair-path**.

Chapter 21. Common Environment Variables

Most Nix commands interpret the following environment variables:

`IN_NIX_SHELL` Indicator that tells if the current environment was set up by **nix-shell**. Since Nix 2.0 the values are `"pure"` and `"impure"`

`NIX_PATH` A colon-separated list of directories used to look up Nix expressions enclosed in angle brackets (i.e., `<path>`). For instance, the value

```
/home/eelco/Dev:/etc/nixos
```

will cause Nix to look for paths relative to `/home/eelco/Dev` and `/etc/nixos`, in that order. It is also possible to match paths against a prefix. For example, the value

```
nixpkgs=/home/eelco/Dev/nixpkgs-branch:/etc/nixos
```

will cause Nix to search for `<nixpkgs/path>` in `/home/eelco/Dev/nixpkgs-branch/path` and `/etc/nixos/nixpkgs/path`.

If a path in the Nix search path starts with `http://` or `https://`, it is interpreted as the URL of a tarball that will be downloaded and unpacked to a temporary location. The tarball must consist of a single top-level directory. For example, setting `NIX_PATH` to

```
nixpkgs=https://github.com/NixOS/nixpkgs-channels/archive/nixos-15.09.tar.gz
```

tells Nix to download the latest revision in the Nixpkgs/NixOS 15.09 channel.

A following shorthand can be used to refer to the official channels:

```
1 nixpkgs=channel:nixos-15.09
```

The search path can be extended using the `-I` option, which takes precedence over `NIX_PATH`.

`NIX_IGNORE_SYMLINK_STORE` Normally, the Nix store directory (typically `/nix/store`) is not allowed to contain any symlink components. This is to prevent "impure" builds. Builders sometimes "canonicalise" paths by resolving all symlink components. Thus, builds on different machines (with `/nix/store` resolving to different locations) could yield different results. This is generally not a problem, except when builds are deployed to machines where `/nix/store` resolves differently. If you are sure that you're not going to do that, you can set `NIX_IGNORE_SYMLINK_STORE` to `1`.

Note that if you're symlinking the Nix store so that you can put it on another file system than the root file system, on Linux you're better off using `bind` mount points, e.g.,

```
$ mkdir /nix
$ mount -o bind /mnt/otherdisk/nix /nix
```

Consult the mount(8) manual page for details.

`NIX_STORE_DIR` Overrides the location of the Nix store (default `prefix`/store).

`NIX_DATA_DIR` Overrides the location of the Nix static data directory (default `prefix`/share).

`NIX_LOG_DIR` Overrides the location of the Nix log directory (default `prefix`/var/log/nix).

`NIX_STATE_DIR` Overrides the location of the Nix state directory (default `prefix`/var/nix).

`NIX_CONF_DIR` Overrides the location of the Nix configuration directory (default `prefix`/etc/nix).

`TMPDIR` Use the specified directory to store temporary files. In particular, this includes temporary build directories; these can take up substantial amounts of disk space. The default is `/tmp`.

`NIX_REMOTE` This variable should be set to `daemon` if you want to use the Nix daemon to execute Nix operations. This is necessary in multi-user Nix installations. If the Nix daemon's Unix socket is at some non-standard path, this variable should be set to `unix://path/to/socket`. Otherwise, it should be left unset.

`NIX_SHOW_STATS` If set to `1`, Nix will print some evaluation statistics, such as the number of values allocated.

`NIX_COUNT_CALLS` If set to `1`, Nix will print how often functions were called during Nix expression evaluation. This is useful for profiling your Nix expressions.

`GC_INITIAL_HEAP_SIZE` If Nix has been configured to use the Boehm garbage collector, this variable sets the initial size of the heap in bytes. It defaults to 384 MiB. Setting it to a low value reduces memory consumption, but will increase runtime due to the overhead of garbage collection.

Chapter 22. Main Commands

This section lists commands and options that you can use when you work with Nix.

nix-env

Name

nix-env -- manipulate or query Nix user environments

Synopsis

`nix-env` [`--help`] [`--version`] [{ `--verbose` | `-v` } ...] [`--quiet`] [`--no-build-output` | `-Q`] [{ `--max-jobs` | `-j` } *number*] [`--cores` *number*] [`--max-silent-time` *number*] [`--timeout` *number*] [`--keep-going` | `-k`] [`--keep-failed` | `-K`] [`--fallback`] [`--readonly-mode`] [`-I` *path*] [`--option` *name value*] [`--arg` *name value*] [`--argstr` *name value*] [{ `--file` | `-f` } *path*] [{ `--profile` | `-p` } *path*] [`--system-filter` *system*] [`--dry-run`] *operation* [*options*...] [*arguments*...]

Description

The command **nix-env** is used to manipulate Nix user environments. User environments are sets of software packages available to a user at some point in time. In other words, they are a synthesised view of the programs available in the Nix store. There may be many user environments: different users can have different environments, and individual users can switch between different environments.

nix-env takes exactly one *operation* flag which indicates the subcommand to be performed. These are documented below.

Selectors

Several commands, such as **nix-env -q** and **nix-env -i**, take a list of arguments that specify the packages on which to operate. These are extended regular expressions that must match the entire name of the package. (For details on regular expressions, see regex(7).) The match is case-sensitive. The regular expression can optionally be followed by a dash and a version number; if omitted, any version of the package will match. Here are some examples:

`firefox` Matches the package name `firefox` and any version.

`firefox-32.0` Matches the package name `firefox` and version `32.0`.

`gtk\\+` Matches the package name `gtk+`. The `+` character must be escaped using a backslash to prevent it from being interpreted as a quantifier, and the backslash must be escaped in turn with another backslash to ensure that the shell passes it on.

`.\*` Matches any package name. This is the default for most commands.

`'.*zip.*'` Matches any package name containing the string `zip`. Note the dots: `'*zip*'` does not work, because in a regular expression, the character `*` is interpreted as a quantifier.

`'.*(firefox|chromium).*'` Matches any package name containing the strings `firefox` or `chromium`.

Common options

This section lists the options that are common to all operations. These options are allowed for every subcommand, though they may not always have an effect. See also Chapter 20, *Common Options*.

`--file` / `-f` *`path`* Specifies the Nix expression (designated below as the *active Nix expression*) used by the `--install`, `--upgrade`, and `--query --available` operations to obtain derivations. The default is `~/.nix-defexpr`.

If the argument starts with `http://` or `https://`, it is interpreted as the URL of a tarball that will be downloaded and unpacked to a temporary location. The tarball must include a single top-level directory containing at least a file named `default.nix`.

`--profile` / `-p` *`path`* Specifies the profile to be used by those operations that operate on a profile (designated below as the *active profile*). A profile is a sequence of user environments called *generations*, one of which is the *current generation*.

`--dry-run` For the `--install`, `--upgrade`, `--uninstall`, `--switch-generation`, `--delete-generations` and `--rollback` operations, this flag will cause **nix-env** to print what *would* be done if this flag had not been specified, without actually doing it.

`--dry-run` also prints out which paths will be substituted (i.e., downloaded) and which paths will be built from source (because no substitute is available).

`--system-filter` *`system`* By default, operations such as `--query --available` show derivations matching any platform. This option allows you to use derivations for the specified platform *`system`*.

Files

`~/.nix-defexpr` The source for the default Nix expressions used by the `--install`, `--upgrade`, and `--query --available` operations to obtain derivations. The `--file` option may be used to override this default.

If `~/.nix-defexpr` is a file, it is loaded as a Nix expression. If the expression is a set, it is used as the default Nix expression. If the expression is a function, an empty set is passed as argument and the return value is used as the default Nix expression.

If `~/.nix-defexpr` is a directory containing a `default.nix` file, that file is loaded as in the above paragraph.

If `~/.nix-defexpr` is a directory without a `default.nix` file, then its contents (both files and subdirectories) are loaded as Nix expressions. The expressions are combined into a single set, each expression under an attribute with the same name as the original file or subdirectory.

For example, if `~/.nix-defexpr` contains two files, `foo.nix` and `bar.nix`, then the default Nix expression will essentially be

```
{
  foo = import ~/.nix-defexpr/foo.nix;
  bar = import ~/.nix-defexpr/bar.nix;
}
```

The file `manifest.nix` is always ignored. Subdirectories without a `default.nix` file are traversed recursively in search of more Nix expressions, but the names of these intermediate directories are not added to the attribute paths of the default Nix expression.

The command **nix-channel** places symlinks to the downloaded Nix expressions from each subscribed channel in this directory.

`~/.nix-profile` A symbolic link to the user's current profile. By default, this symlink points to `prefix`/var/nix/profiles/default. The `PATH` environment variable should include `~/.nix-profile/bin` for the user environment to be visible to the user.

Operation `--install`

Synopsis

`nix-env` { `--install` | `-i` } [{ `--prebuilt-only` | `-b` }] [{ `--attr` | `-A` }] [`--from-expression`] [`-E`] [`--from-profile` *path*] [`--preserve-installed` | `-P`] [`--remove-all` | `-r`] *args*...

Description

The install operation creates a new user environment, based on the current generation of the active profile, to which a set of store paths described by *args* is added. The arguments *args* map to store paths in a number of possible ways:

- By default, *args* is a set of derivation names denoting derivations in the active Nix expression. These are realised, and the resulting output paths are installed. Currently installed derivations with a name equal to the name of a derivation being added are removed unless the option `--preserve-installed` is specified.

  If there are multiple derivations matching a name in *args* that have the same name (e.g., `gcc-3.3.6` and `gcc-4.1.1`), then the derivation with the highest *priority* is used. A derivation can define a priority by declaring the `meta.priority` attribute. This attribute should be a number, with a higher value denoting a lower priority. The default priority is `0`.

  If there are multiple matching derivations with the same priority, then the derivation with the highest version will be installed.

  You can force the installation of multiple derivations with the same name by being specific about the versions. For instance, `nix-env -i gcc-3.3.6 gcc-4.1.1` will install both version of GCC (and will probably cause a user environment conflict!).

- If `--attr` (`-A`) is specified, the arguments are *attribute paths* that select attributes from the top-level Nix expression. This is faster than using derivation names and unambiguous. To find out the attribute paths of available packages, use `nix-env -qaP`.

- If `--from-profile` *path* is given, *args* is a set of names denoting installed store paths in the profile *path*. This is an easy way to copy user environment elements from one profile to another.

- If `--from-expression` is given, *args* are Nix functions that are called with the active Nix expression as their single argument. The derivations returned by those function calls are installed. This allows derivations to be specified in an unambiguous way, which is necessary if there are multiple derivations with the same name.

- If *args* are store derivations, then these are realised, and the resulting output paths are installed.

- If *args* are store paths that are not store derivations, then these are realised and installed.

- By default all outputs are installed for each derivation. That can be reduced by setting `meta.outputsToInstall`.

Flags

`--prebuilt-only` / `-b` Use only derivations for which a substitute is registered, i.e., there is a pre-built binary available that can be downloaded in lieu of building the derivation. Thus, no packages will be built from source.

`--preserve-installed, -P` Do not remove derivations with a name matching one of the derivations being installed. Usually, trying to have two versions of the same package installed in the same generation of a profile will lead to an error in building the generation, due to file name clashes between the two versions. However, this is not the case for all packages.

`--remove-all, -r` Remove all previously installed packages first. This is equivalent to running `nix-env -e '.*'` first, except that everything happens in a single transaction.

Examples

To install a specific version of **gcc** from the active Nix expression:

```
$ nix-env --install gcc-3.3.2
installing `gcc-3.3.2'
uninstalling `gcc-3.1'
```

Note the previously installed version is removed, since `--preserve-installed` was not specified.

To install an arbitrary version:

```
$ nix-env --install gcc
installing `gcc-3.3.2'
```

To install using a specific attribute:

```
$ nix-env -i -A gcc40mips
$ nix-env -i -A xorg.xorgserver
```

To install all derivations in the Nix expression `foo.nix`:

```
$ nix-env -f ~/foo.nix -i '.*'
```

To copy the store path with symbolic name `gcc` from another profile:

```
$ nix-env -i --from-profile /nix/var/nix/profiles/foo gcc
```

To install a specific store derivation (typically created by **nix-instantiate**):

```
$ nix-env -i /nix/store/fibjb1bfbpm5mrsxc4mh2d8n37sxh91i-gcc-3.4.3.drv
```

To install a specific output path:

```
$ nix-env -i /nix/store/y3cgx0xj1p4iv9x0pnnmdhr8iyg741vk-gcc-3.4.3
```

To install from a Nix expression specified on the command-line:

```
$ nix-env -f ./foo.nix -i -E \
    'f: (f {system = "i686-linux";}).subversionWithJava'
```

I.e., this evaluates to `(f: (f {system = "i686-linux";}).subversionWithJava)(import ./foo.nix)`, thus selecting the `subversionWithJava` attribute from the set returned by calling the function defined in `./foo.nix`.

A dry-run tells you which paths will be downloaded or built from source:

```
$ nix-env -f '<nixpkgs>' -iA hello --dry-run
(dry run; not doing anything)
installing ‘hello-2.10'
these paths will be fetched (0.04 MiB download, 0.19 MiB unpacked):
  /nix/store/wkhdf9jinag5750mqlax6z2zbwhqb76n-hello-2.10
  ...
```

To install Firefox from the latest revision in the Nixpkgs/NixOS 14.12 channel:

```
$ nix-env -f https://github.com/NixOS/nixpkgs-channels/archive/nixos-14.12.tar.gz
   ↪ -iA firefox
```

(The GitHub repository `nixpkgs-channels` is updated automatically from the main `nixpkgs` repository after certain tests have succeeded and binaries have been built and uploaded to the binary cache at `cache.nixos.org`.)

Operation `--upgrade`

Synopsis

`nix-env { --upgrade | -u } [ { --prebuilt-only | -b } ] [ { --attr | -A } ] [--from-expression] [-E] [--from-profile` *`path`*`] [ --lt | --leq | --eq | --always ]` *`args`*`...`

Description

The upgrade operation creates a new user environment, based on the current generation of the active profile, in which all store paths are replaced for which there are newer versions in the set of paths described by *`args`*. Paths for which there are no newer versions are left untouched; this is not an error. It is also not an error if an element of *`args`* matches no installed derivations.

For a description of how *`args`* is mapped to a set of store paths, see `--install`. If *`args`* describes multiple store paths with the same symbolic name, only the one with the highest version is installed.

Flags

`--lt` Only upgrade a derivation to newer versions. This is the default.

`--leq` In addition to upgrading to newer versions, also "upgrade" to derivations that have the same version. Version are not a unique identification of a derivation, so there may be many derivations that have the same version. This flag may be useful to force "synchronisation" between the installed and available derivations.

`--eq` *Only* "upgrade" to derivations that have the same version. This may not seem very useful, but it actually is, e.g., when there is a new release of Nixpkgs and you want to replace installed applications with the same versions built against newer dependencies (to reduce the number of dependencies floating around on your system).

`--always` In addition to upgrading to newer versions, also "upgrade" to derivations that have the same or a lower version. I.e., derivations may actually be downgraded depending on what is available in the active Nix expression.

For the other flags, see `--install`.

Examples

```
$ nix-env --upgrade gcc
upgrading `gcc-3.3.1' to `gcc-3.4'

$ nix-env -u gcc-3.3.2 --always (switch to a specific version)
upgrading `gcc-3.4' to `gcc-3.3.2'

$ nix-env --upgrade pan
(no upgrades available, so nothing happens)

$ nix-env -u (try to upgrade everything)
upgrading `hello-2.1.2' to `hello-2.1.3'
upgrading `mozilla-1.2' to `mozilla-1.4'
```

Versions

The upgrade operation determines whether a derivation `y` is an upgrade of a derivation `x` by looking at their respective `name` attributes. The names (e.g., `gcc-3.3.1` are split into two parts: the package name (`gcc`), and the

version (`3.3.1`). The version part starts after the first dash not followed by a letter. `x` is considered an upgrade of `y` if their package names match, and the version of `y` is higher that that of `x`.

The versions are compared by splitting them into contiguous components of numbers and letters. E.g., `3.3.1pre5` is split into `[3, 3, 1, "pre", 5]`. These lists are then compared lexicographically (from left to right). Corresponding components `a` and `b` are compared as follows. If they are both numbers, integer comparison is used. If `a` is an empty string and `b` is a number, `a` is considered less than `b`. The special string component `pre` (for *pre-release*) is considered to be less than other components. String components are considered less than number components. Otherwise, they are compared lexicographically (i.e., using case-sensitive string comparison).

This is illustrated by the following examples:

```
1.0 < 2.3
2.1 < 2.3
2.3 = 2.3
2.5 > 2.3
3.1 > 2.3
2.3.1 > 2.3
2.3.1 > 2.3a
2.3pre1 < 2.3
2.3pre3 < 2.3pre12
2.3a < 2.3c
2.3pre1 < 2.3c
2.3pre1 < 2.3q
```

Operation `--uninstall`

Synopsis

`nix-env` { `--uninstall` | `-e` } *drvnames...*

Description

The uninstall operation creates a new user environment, based on the current generation of the active profile, from which the store paths designated by the symbolic names *names* are removed.

Examples

```
$ nix-env --uninstall gcc
$ nix-env -e '.*' (remove everything)
```

Operation `--set`

Synopsis

`nix-env --set` *drvname*

Description

The `--set` operation modifies the current generation of a profile so that it contains exactly the specified derivation, and nothing else.

Examples

The following updates a profile such that its current generation will contain just Firefox:

```
$ nix-env -p /nix/var/nix/profiles/browser --set firefox
```

Operation `--set-flag`

Synopsis

`nix-env --set-flag` *name value drvnames...*

Description

The `--set-flag` operation allows meta attributes of installed packages to be modified. There are several attributes that can be usefully modified, because they affect the behaviour of **nix-env** or the user environment build script:

- `priority` can be changed to resolve filename clashes. The user environment build script uses the `meta.priority` attribute of derivations to resolve filename collisions between packages. Lower priority values denote a higher priority. For instance, the GCC wrapper package and the Binutils package in Nixpkgs both have a file `bin/ld`, so previously if you tried to install both you would get a collision. Now, on the other hand, the GCC wrapper declares a higher priority than Binutils, so the former's `bin/ld` is symlinked in the user environment.
- `keep` can be set to `true` to prevent the package from being upgraded or replaced. This is useful if you want to hang on to an older version of a package.
- `active` can be set to `false` to "disable" the package. That is, no symlinks will be generated to the files of the package, but it remains part of the profile (so it won't be garbage-collected). It can be set back to `true` to re-enable the package.

Examples

To prevent the currently installed Firefox from being upgraded:

```
$ nix-env --set-flag keep true firefox
```

After this, **nix-env -u** will ignore Firefox.

To disable the currently installed Firefox, then install a new Firefox while the old remains part of the profile:

```
$ nix-env -q
firefox-2.0.0.9 (the current one)

$ nix-env --preserve-installed -i firefox-2.0.0.11
installing `firefox-2.0.0.11'
building path(s) `/nix/store/myy0y59q3ig70dgq37jqwg1j0rsapzsl-user-environment'
collision between `/nix/store/...-firefox-2.0.0.11/bin/firefox'
  and `/nix/store/...-firefox-2.0.0.9/bin/firefox'.
(i.e., can't have two active at the same time)

$ nix-env --set-flag active false firefox
setting flag on `firefox-2.0.0.9'

$ nix-env --preserve-installed -i firefox-2.0.0.11
installing `firefox-2.0.0.11'

$ nix-env -q
firefox-2.0.0.11 (the enabled one)
firefox-2.0.0.9 (the disabled one)
```

To make files from `binutils` take precedence over files from `gcc`:

```
$ nix-env --set-flag priority 5 binutils
$ nix-env --set-flag priority 10 gcc
```

Operation `--query`

Synopsis

`nix-env { --query | -q } [ --installed | --available | -a ]`
`[ { --status | -s } ] [ { --attr-path | -P } ] [--no-name] [ { --compare-versions | -c } ] [--system] [--drv-path]`
`[--out-path] [--description] [--meta]`

[--xml] [--json] [{ --prebuilt-only | -b }] [{ --attr | -A } *attribute-path*]
names...

Description

The query operation displays information about either the store paths that are installed in the current generation of the active profile (`--installed`), or the derivations that are available for installation in the active Nix expression (`--available`). It only prints information about derivations whose symbolic name matches one of *names*.

The derivations are sorted by their `name` attributes.

Source selection

The following flags specify the set of things on which the query operates.

`--installed` The query operates on the store paths that are installed in the current generation of the active profile. This is the default.

`--available, -a` The query operates on the derivations that are available in the active Nix expression.

Queries

The following flags specify what information to display about the selected derivations. Multiple flags may be specified, in which case the information is shown in the order given here. Note that the name of the derivation is shown unless `--no-name` is specified.

`--xml` Print the result in an XML representation suitable for automatic processing by other tools. The root element is called `items`, which contains a `item` element for each available or installed derivation. The fields discussed below are all stored in attributes of the `item` elements.

`--json` Print the result in a JSON representation suitable for automatic processing by other tools.

`--prebuilt-only / -b` Show only derivations for which a substitute is registered, i.e., there is a pre-built binary available that can be downloaded in lieu of building the derivation. Thus, this shows all packages that probably can be installed quickly.

`--status, -s` Print the *status* of the derivation. The status consists of three characters. The first is `I` or `-`, indicating whether the derivation is currently installed in the current generation of the active profile. This is by definition the case for `--installed`, but not for `--available`. The second is `P` or `-`, indicating whether the derivation is present on the system. This indicates whether installation of an available derivation will require the derivation to be built. The third is `S` or `-`, indicating whether a substitute is available for the derivation.

`--attr-path, -P` Print the *attribute path* of the derivation, which can be used to unambiguously select it using the `--attr` option available in commands that install derivations like `nix-env --install`.

`--no-name` Suppress printing of the `name` attribute of each derivation.

`--compare-versions / -c` Compare installed versions to available versions, or vice versa (if `--available` is given). This is useful for quickly seeing whether upgrades for installed packages are available in a Nix expression. A column is added with the following meaning:

`< version` A newer version of the package is available or installed.

`= version` At most the same version of the package is available or installed.

`> version` Only older versions of the package are available or installed.

`- ?` No version of the package is available or installed.

`--system` Print the `system` attribute of the derivation.

`--drv-path` Print the path of the store derivation.

`--out-path` Print the output path of the derivation.

`--description` Print a short (one-line) description of the derivation, if available. The description is taken from the `meta.description` attribute of the derivation.

`--meta` Print all of the meta-attributes of the derivation. This option is only available with `--xml` or `--json`.

Examples

To show installed packages:

```
$ nix-env -q
bison-1.875c
docbook-xml-4.2
firefox-1.0.4
MPlayer-1.0pre7
ORBit2-2.8.3
...
```

To show available packages:

```
$ nix-env -qa
firefox-1.0.7
GConf-2.4.0.1
MPlayer-1.0pre7
ORBit2-2.8.3
...
```

To show the status of available packages:

```
$ nix-env -qas
-P- firefox-1.0.7   (not installed but present)
--S GConf-2.4.0.1   (not present, but there is a substitute for fast installation)
--S MPlayer-1.0pre3 (i.e., this is not the installed MPlayer, even though the
   ↪ version is the same!)
IP- ORBit2-2.8.3    (installed and by definition present)
...
```

To show available packages in the Nix expression `foo.nix`:

```
$ nix-env -f ./foo.nix -qa
foo-1.2.3
```

To compare installed versions to what's available:

```
$ nix-env -qc
...
acrobat-reader-7.0 - ?      (package is not available at all)
autoconf-2.59      = 2.59   (same version)
firefox-1.0.4      < 1.0.7  (a more recent version is available)
...
```

To show all packages with "`zip`" in the name:

```
$ nix-env -qa '.*zip.*'
bzip2-1.0.6
gzip-1.6
zip-3.0
...
```

To show all packages with "`firefox`" or "`chromium`" in the name:

```
$ nix-env -qa '.*(firefox|chromium).*'
chromium-37.0.2062.94
chromium-beta-38.0.2125.24
firefox-32.0.3
firefox-with-plugins-13.0.1
...
```

To show all packages in the latest revision of the Nixpkgs repository:

```
$ nix-env -f https://github.com/NixOS/nixpkgs/archive/master.tar.gz -qa
```

Operation `--switch-profile`

Synopsis

`nix-env` { `--switch-profile` | `-S` } {*path*}

Description

This operation makes *path* the current profile for the user. That is, the symlink `~/.nix-profile` is made to point to *path*.

Examples

```
$ nix-env -S ~/my-profile
```

Operation `--list-generations`

Synopsis

`nix-env --list-generations`

Description

This operation print a list of all the currently existing generations for the active profile. These may be switched to using the `--switch-generation` operation. It also prints the creation date of the generation, and indicates the current generation.

Examples

```
$ nix-env --list-generations
  95   2004-02-06 11:48:24
  96   2004-02-06 11:49:01
  97   2004-02-06 16:22:45
  98   2004-02-06 16:24:33   (current)
```

Operation `--delete-generations`

Synopsis

`nix-env --delete-generations` *generations...*

Description

This operation deletes the specified generations of the current profile. The generations can be a list of generation numbers, the special value `old` to delete all non-current generations, a value such as `30d` to delete all generations older than the specified number of days (except for the generation that was active at that point in time), or a value such as `+5` to keep the last `5` generations ignoring any newer than current, e.g., if `30` is the current generation `+5` will delete generation `25` and all older generations. Periodically deleting old generations is important to make garbage collection effective.

Examples

```
$ nix-env --delete-generations 3 4 8

$ nix-env --delete-generations +5

$ nix-env --delete-generations 30d

$ nix-env -p other_profile --delete-generations old
```

Operation `--switch-generation`

Synopsis

`nix-env` { `--switch-generation` | `-G` } {*`generation`*}

Description

This operation makes generation number *`generation`* the current generation of the active profile. That is, if the `profile` is the path to the active profile, then the symlink `profile` is made to point to `profile-`*`generation`*`-link`, which is in turn a symlink to the actual user environment in the Nix store.

Switching will fail if the specified generation does not exist.

Examples

```
$ nix-env -G 42
switching from generation 50 to 42
```

Operation `--rollback`

Synopsis

`nix-env --rollback`

Description

This operation switches to the "previous" generation of the active profile, that is, the highest numbered generation lower than the current generation, if it exists. It is just a convenience wrapper around `--list-generations` and `--switch-generation`.

Examples

```
$ nix-env --rollback
switching from generation 92 to 91

$ nix-env --rollback
error: no generation older than the current (91) exists
```

---

nix-build

Name

nix-build -- build a Nix expression

Synopsis

`nix-build` [`--help`] [`--version`] [{ `--verbose` | `-v` } ...] [`--quiet`] [`--no-build-output` | `-Q`] [{ `--max-jobs` | `-j` } *`number`*] [`--cores` *`number`*] [`--max-silent-time` *`number`*] [`--timeout` *`number`*] [`--keep-going` | `-k`] [`--keep-failed` | `-K`] [`--fallback`] [`--readonly-mode`] [`-I` *`path`*] [`--option` *`name value`*] [`--arg` *`name value`*] [`--argstr` *`name value`*] [{ `--attr` | `-A` } *`attrPath`*] [`--no-out-link`] [`--dry-run`] [{ `--out-link` | `-o` } *`outlink`*] *`paths`*...

Description

The **nix-build** command builds the derivations described by the Nix expressions in *`paths`*. If the build succeeds, it places a symlink to the result in the current directory. The symlink is called `result`. If there are multiple Nix expressions, or the Nix expressions evaluate to multiple derivations, multiple sequentially numbered symlinks are created (`result`, `result-2`, and so on).

If no *`paths`* are specified, then **nix-build** will use `default.nix` in the current directory, if it exists.

If an element of *paths* starts with `http://` or `https://`, it is interpreted as the URL of a tarball that will be downloaded and unpacked to a temporary location. The tarball must include a single top-level directory containing at least a file named `default.nix`.

nix-build is essentially a wrapper around **nix-instantiate** (to translate a high-level Nix expression to a low-level store derivation) and **nix-store --realise** (to build the store derivation).

Warning

The result of the build is automatically registered as a root of the Nix garbage collector. This root disappears automatically when the `result` symlink is deleted or renamed. So don't rename the symlink.

Options

All options not listed here are passed to **nix-store --realise**, except for `--arg` and `--attr` / `-A` which are passed to **nix-instantiate**. See also Chapter 20, *Common Options*.

`--no-out-link` Do not create a symlink to the output path. Note that as a result the output does not become a root of the garbage collector, and so might be deleted by **nix-store --gc**.

`--dry-run` Show what store paths would be built or downloaded

`--out-link` / `-o` *outlink* Change the name of the symlink to the output path created from `result` to *outlink*.

The following common options are supported:

Examples

```
$ nix-build '<nixpkgs>' -A firefox
store derivation is /nix/store/qybprl8sz2lc...-firefox-1.5.0.7.drv
/nix/store/d18hyl92g30l...-firefox-1.5.0.7

$ ls -l result
lrwxrwxrwx  ...  result -> /nix/store/d18hyl92g30l...-firefox-1.5.0.7

$ ls ./result/bin/
firefox  firefox-config
```

If a derivation has multiple outputs, **nix-build** will build the default (first) output. You can also build all outputs:

```
$ nix-build '<nixpkgs>' -A openssl.all
```

This will create a symlink for each output named `result-outputname`. The suffix is omitted if the output name is `out`. So if `openssl` has outputs `out`, `bin` and `man`, **nix-build** will create symlinks `result`, `result-bin` and `result-man`. It's also possible to build a specific output:

```
$ nix-build '<nixpkgs>' -A openssl.man
```

This will create a symlink `result-man`.

Build a Nix expression given on the command line:

```
$ nix-build -E 'with import <nixpkgs> { }; runCommand "foo" { } "echo bar > $out"'
$ cat ./result
bar
```

Build the GNU Hello package from the latest revision of the master branch of Nixpkgs:

```
$ nix-build https://github.com/NixOS/nixpkgs/archive/master.tar.gz -A hello
```

---

nix-shell

Name

nix-shell -- start an interactive shell based on a Nix expression

Synopsis

`nix-shell` [`--arg` *name value*] [`--argstr` *name value*] [{ `--attr` | `-A` } *attrPath*] [`--command` *cmd*] [`--run` *cmd*] [`--exclude` *regexp*] [`--pure`] [`--keep` *name*] { { `--packages` | `-p` } *packages...* | [*path*]}

Description

The command **nix-shell** will build the dependencies of the specified derivation, but not the derivation itself. It will then start an interactive shell in which all environment variables defined by the derivation *path* have been set to their corresponding values, and the script `$stdenv/setup` has been sourced. This is useful for reproducing the environment of a derivation for development.

If *path* is not given, **nix-shell** defaults to `shell.nix` if it exists, and `default.nix` otherwise.

If *path* starts with `http://` or `https://`, it is interpreted as the URL of a tarball that will be downloaded and unpacked to a temporary location. The tarball must include a single top-level directory containing at least a file named `default.nix`.

If the derivation defines the variable `shellHook`, it will be evaluated after `$stdenv/setup` has been sourced. Since this hook is not executed by regular Nix builds, it allows you to perform initialisation specific to **nix-shell**. For example, the derivation attribute

```
shellHook =
  ''
    echo "Hello shell"
  '';
```

will cause **nix-shell** to print `Hello shell`.

Options

All options not listed here are passed to **nix-store --realise**, except for `--arg` and `--attr` / `-A` which are passed to **nix-instantiate**. See also Chapter 20, *Common Options*.

`--command` *cmd* In the environment of the derivation, run the shell command *cmd*. This command is executed in an interactive shell. (Use `--run` to use a non-interactive shell instead.) However, a call to `exit` is implicitly added to the command, so the shell will exit after running the command. To prevent this, add `return` at the end; e.g. `--command "echo Hello; return"` will print `Hello` and then drop you into the interactive shell. This can be useful for doing any additional initialisation.

`--run` *cmd* Like `--command`, but executes the command in a non-interactive shell. This means (among other things) that if you hit Ctrl-C while the command is running, the shell exits.

`--exclude` *regexp* Do not build any dependencies whose store path matches the regular expression *regexp*. This option may be specified multiple times.

`--pure` If this flag is specified, the environment is almost entirely cleared before the interactive shell is started, so you get an environment that more closely corresponds to the "real" Nix build. A few variables, in particular `HOME`, `USER` and `DISPLAY`, are retained. Note that `~/.bashrc` and (depending on your Bash installation) `/etc/bashrc` are still sourced, so any variables set there will affect the interactive shell.

`--packages` / `-p` *packages...* Set up an environment in which the specified packages are present. The command line arguments are interpreted as attribute names inside the Nix Packages collection. Thus, `nix-shell -p ↪ libjpeg openjdk` will start a shell in which the packages denoted by the attribute names `libjpeg` and `openjdk` are present.

`-i` *interpreter* The chained script interpreter to be invoked by **nix-shell**. Only applicable in `#!`-scripts (described below).

`--keep` *name* When a `--pure` shell is started, keep the listed environment variables.

The following common options are supported:

Environment variables

`NIX_BUILD_SHELL` Shell used to start the interactive environment. Defaults to the **bash** found in `PATH`.

Examples

To build the dependencies of the package Pan, and start an interactive shell in which to build it:

```
$ nix-shell '<nixpkgs>' -A pan
[nix-shell]$ unpackPhase
[nix-shell]$ cd pan-*
[nix-shell]$ configurePhase
[nix-shell]$ buildPhase
[nix-shell]$ ./pan/gui/pan
```

To clear the environment first, and do some additional automatic initialisation of the interactive shell:

```
$ nix-shell '<nixpkgs>' -A pan --pure \
    --command 'export NIX_DEBUG=1; export NIX_CORES=8; return'
```

Nix expressions can also be given on the command line. For instance, the following starts a shell containing the packages `sqlite` and `libX11`:

```
$ nix-shell -E 'with import <nixpkgs> { }; runCommand "dummy" { buildInputs = [
   ↪ sqlite xorg.libX11 ]; } ""'
```

A shorter way to do the same is:

```
$ nix-shell -p sqlite xorg.libX11
[nix-shell]$ echo $NIX_LDFLAGS
... -L/nix/store/j1zg5v...-sqlite-3.8.0.2/lib
   ↪ -L/nix/store/0gmcz9...-libX11-1.6.1/lib ...
```

The **-p** flag looks up Nixpkgs in the Nix search path. You can override it by passing `-I` or setting `NIX_PATH`. For example, the following gives you a shell containing the Pan package from a specific revision of Nixpkgs:

```
$ nix-shell -p pan -I
   ↪ nixpkgs=https://github.com/NixOS/nixpkgs-channels/archive/8a3eea054838b55aca962c3fbde9c

[nix-shell:~]$ pan --version
Pan 0.139
```

Use as a `#!`-interpreter

You can use **nix-shell** as a script interpreter to allow scripts written in arbitrary languages to obtain their own dependencies via Nix. This is done by starting the script with the following lines:

```
#! /usr/bin/env nix-shell
#! nix-shell -i real-interpreter -p packages
```

where *`real-interpreter`* is the "real" script interpreter that will be invoked by **nix-shell** after it has obtained the dependencies and initialised the environment, and *`packages`* are the attribute names of the dependencies in Nixpkgs.

The lines starting with `#! nix-shell` specify **nix-shell** options (see above). Note that you cannot write `#!` ↪ `/usr/bin/env nix-shell -i ...` because many operating systems only allow one argument in `#!` lines.

For example, here is a Python script that depends on Python and the `prettytable` package:

```
#! /usr/bin/env nix-shell
#! nix-shell -i python -p python pythonPackages.prettytable

import prettytable

# Print a simple table.
t = prettytable.PrettyTable(["N", "N^2"])
for n in range(1, 10): t.add_row([n, n * n])
print t
```

Similarly, the following is a Perl script that specifies that it requires Perl and the `HTML::TokeParser::Simple` and `LWP` packages:

```
#! /usr/bin/env nix-shell
#! nix-shell -i perl -p perl perlPackages.HTMLTokeParserSimple perlPackages.LWP

use HTML::TokeParser::Simple;

# Fetch nixos.org and print all hrefs.
my $p = HTML::TokeParser::Simple->new(url => 'http://nixos.org/');

while (my $token = $p->get_tag("a")) {
    my $href = $token->get_attr("href");
    print "$href\n" if $href;
}
```

Sometimes you need to pass a simple Nix expression to customize a package like Terraform:

```
#! /usr/bin/env nix-shell
#! nix-shell -i bash -p "terraform.withPlugins (plugins: [ plugins.openstack ])"

terraform apply
```

Note

You must use double quotes (") when passing a simple Nix expression in a nix-shell shebang.

Finally, using the merging of multiple nix-shell shebangs the following Haskell script uses a specific branch of Nixpkgs/NixOS (the 18.03 stable branch):

```
#! /usr/bin/env nix-shell
#! nix-shell -i runghc -p "haskellPackages.ghcWithPackages (ps: [ps.HTTP
   ↪ ps.tagsoup])"
#! nix-shell -I
   ↪ nixpkgs=https://github.com/NixOS/nixpkgs-channels/archive/nixos-18.03.tar.gz

import Network.HTTP
import Text.HTML.TagSoup

-- Fetch nixos.org and print all hrefs.
main = do
  resp <- Network.HTTP.simpleHTTP (getRequest "http://nixos.org/")
  body <- getResponseBody resp
  let tags = filter (isTagOpenName "a") $ parseTags body
  let tags' = map (fromAttrib "href") tags
  mapM_ putStrLn $ filter (/= "") tags'
```

If you want to be even more precise, you can specify a specific revision of Nixpkgs:

```
#! nix-shell -I
    ↪ nixpkgs=https://github.com/NixOS/nixpkgs-channels/archive/0672315759b3e15e2121365f067c1
```

The examples above all used `-p` to get dependencies from Nixpkgs. You can also use a Nix expression to build your own dependencies. For example, the Python example could have been written as:

```
#! /usr/bin/env nix-shell
#! nix-shell deps.nix -i python
```

where the file `deps.nix` in the same directory as the `#!`-script contains:

```
with import <nixpkgs> {};

runCommand "dummy" { buildInputs = [ python pythonPackages.prettytable ]; } ""
```

---

nix-store

Name

nix-store -- manipulate or query the Nix store

Synopsis

`nix-store` [`--help`] [`--version`] [{ `--verbose` | `-v` } ...] [`--quiet`] [`--no-build-output` | `-Q`] [{ `--max-jobs` | `-j` } *`number`*] [`--cores` *`number`*] [`--max-silent-time` *`number`*] [`--timeout` *`number`*] [`--keep-going` | `-k`] [`--keep-failed` | `-K`] [`--fallback`] [`--readonly-mode`] [`-I` *`path`*] [`--option` *`name value`*] [`--add-root` *`path`*] [`--indirect`] *`operation`* [*`options`*...] [*`arguments`*...]

Description

The command **nix-store** performs primitive operations on the Nix store. You generally do not need to run this command manually.

nix-store takes exactly one *operation* flag which indicates the subcommand to be performed. These are documented below.

Common options

This section lists the options that are common to all operations. These options are allowed for every subcommand, though they may not always have an effect. See also Chapter 20, *Common Options* for a list of common options.

`--add-root` *`path`* Causes the result of a realisation (`--realise` and `--force-realise`) to be registered as a root of the garbage collector (see Section 11.1, "Garbage Collector Roots"). The root is stored in *`path`*, which must be inside a directory that is scanned for roots by the garbage collector (i.e., typically in a subdirectory of `/nix/var/nix/gcroots/`) *unless* the `--indirect` flag is used.

If there are multiple results, then multiple symlinks will be created by sequentially numbering symlinks beyond the first one (e.g., `foo`, `foo-2`, `foo-3`, and so on).

`--indirect` In conjunction with `--add-root`, this option allows roots to be stored *outside* of the GC roots directory. This is useful for commands such as **nix-build** that place a symlink to the build result in the current directory; such a build result should not be garbage-collected unless the symlink is removed.

The `--indirect` flag causes a uniquely named symlink to *`path`* to be stored in `/nix/var/nix/gcroots/auto/`. For instance,

```
$ nix-store --add-root /home/eelco/bla/result --indirect -r ...

$ ls -l /nix/var/nix/gcroots/auto
lrwxrwxrwx    1 ... 2005-03-13 21:10 dn54lcypm8f8... -> /home/eelco/bla/result

$ ls -l /home/eelco/bla/result
```

```
lrwxrwxrwx    1 ... 2005-03-13 21:10 /home/eelco/bla/result ->
   ↪ /nix/store/1r11343n6qd4...-f-spot-0.0.10
```

Thus, when `/home/eelco/bla/result` is removed, the GC root in the `auto` directory becomes a dangling symlink and will be ignored by the collector.

Warning

Note that it is not possible to move or rename indirect GC roots, since the symlink in the `auto` directory will still point to the old location.

Operation `--realise`

Synopsis

`nix-store` { `--realise` | `-r` } *paths...* [`--dry-run`]

Description

The operation `--realise` essentially "builds" the specified store paths. Realisation is a somewhat overloaded term:

- If the store path is a *derivation*, realisation ensures that the output paths of the derivation are valid (i.e., the output path and its closure exist in the file system). This can be done in several ways. First, it is possible that the outputs are already valid, in which case we are done immediately. Otherwise, there may be substitutes that produce the outputs (e.g., by downloading them). Finally, the outputs can be produced by performing the build action described by the derivation.
- If the store path is not a derivation, realisation ensures that the specified path is valid (i.e., it and its closure exist in the file system). If the path is already valid, we are done immediately. Otherwise, the path and any missing paths in its closure may be produced through substitutes. If there are no (successful) subsitutes, realisation fails.

The output path of each derivation is printed on standard output. (For non-derivations argument, the argument itself is printed.)

The following flags are available:

`--dry-run` Print on standard error a description of what packages would be built or downloaded, without actually performing the operation.

`--ignore-unknown` If a non-derivation path does not have a substitute, then silently ignore it.

`--check` This option allows you to check whether a derivation is deterministic. It rebuilds the specified derivation and checks whether the result is bitwise-identical with the existing outputs, printing an error if that's not the case. The outputs of the specified derivation must already exist. When used with `-K`, if an output path is not identical to the corresponding output from the previous build, the new output path is left in `/nix/store/name`.check.

See also the `build-repeat` configuration option, which repeats a derivation a number of times and prevents its outputs from being registered as "valid" in the Nix store unless they are identical.

Special exit codes:

`100` Generic build failure, the builder process returned with a non-zero exit code.

`101` Build timeout, the build was aborted because it did not complete within the specified `timeout`.

`102` Hash mismatch, the build output was rejected because it does not match the specified `outputHash`.

`104` Not deterministic, the build succeeded in check mode but the resulting output is not binary reproducable.

With the `--keep-going` flag it's possible for multiple failures to occur, in this case the 1xx status codes are or combined using binary or.

```
1100100
   ^^^^
   |||`- timeout
```

```
   ||`-- output hash mismatch
   |`--- build failure
   `---- not deterministic
```

Examples

This operation is typically used to build store derivations produced by **nix-instantiate**:

```
$ nix-store -r $(nix-instantiate ./test.nix)
/nix/store/31axcgrlbfsxzmfff1gyj1bf62hvkby2-aterm-2.3.1
```

This is essentially what **nix-build** does.

To test whether a previously-built derivation is deterministic:

```
$ nix-build '<nixpkgs>' -A hello --check -K
```

Operation `--serve`

Synopsis

`nix-store --serve` [`--write`]

Description

The operation `--serve` provides access to the Nix store over stdin and stdout, and is intended to be used as a means of providing Nix store access to a restricted ssh user.

The following flags are available:

`--write` Allow the connected client to request the realization of derivations. In effect, this can be used to make the host act as a remote builder.

Examples

To turn a host into a build server, the `authorized_keys` file can be used to provide build access to a given SSH public key:

```
$ cat <<EOF >>/root/.ssh/authorized_keys
command="nice -n20 nix-store --serve --write" ssh-rsa AAAAB3NzaC1yc2EAAAA...
EOF
```

Operation `--gc`

Synopsis

`nix-store --gc` [`--print-roots` | `--print-live` | `--print-dead` | `--delete`] [`--max-freed` *bytes*]

Description

Without additional flags, the operation `--gc` performs a garbage collection on the Nix store. That is, all paths in the Nix store not reachable via file system references from a set of "roots", are deleted.

The following suboperations may be specified:

`--print-roots` This operation prints on standard output the set of roots used by the garbage collector. What constitutes a root is described in Section 11.1, "Garbage Collector Roots".

`--print-live` This operation prints on standard output the set of "live" store paths, which are all the store paths reachable from the roots. Live paths should never be deleted, since that would break consistency -- it would become possible that applications are installed that reference things that are no longer present in the store.

`--print-dead` This operation prints out on standard output the set of "dead" store paths, which is just the opposite of the set of live paths: any path in the store that is not live (with respect to the roots) is dead.

`--delete` This operation performs an actual garbage collection. All dead paths are removed from the store. This is the default.

By default, all unreachable paths are deleted. The following options control what gets deleted and in what order:

`--max-freed` *bytes* Keep deleting paths until at least *bytes* bytes have been deleted, then stop. The argument *bytes* can be followed by the multiplicative suffix `K`, `M`, `G` or `T`, denoting KiB, MiB, GiB or TiB units.

The behaviour of the collector is also influenced by the `keep-outputs` and `keep-derivations` variables in the Nix configuration file.

With `--delete`, the collector prints the total number of freed bytes when it finishes (or when it is interrupted). With `--print-dead`, it prints the number of bytes that would be freed.

Examples

To delete all unreachable paths, just do:

```
$ nix-store --gc
deleting `/nix/store/kq82idx6g0nyzsp2s14gfsc38npai7lf-cairo-1.0.4.tar.gz.drv'
...
8825586 bytes freed (8.42 MiB)
```

To delete at least 100 MiBs of unreachable paths:

```
$ nix-store --gc --max-freed $((100 * 1024 * 1024))
```

Operation `--delete`

Synopsis

`nix-store --delete` [`--ignore-liveness`] *paths*...

Description

The operation `--delete` deletes the store paths *paths* from the Nix store, but only if it is safe to do so; that is, when the path is not reachable from a root of the garbage collector. This means that you can only delete paths that would also be deleted by `nix-store --gc`. Thus, `--delete` is a more targeted version of `--gc`.

With the option `--ignore-liveness`, reachability from the roots is ignored. However, the path still won't be deleted if there are other paths in the store that refer to it (i.e., depend on it).

Example

```
$ nix-store --delete /nix/store/zq0h41l75vlb4z45kzgjjmsjxvcv1qk7-mesa-6.4
0 bytes freed (0.00 MiB)
error: cannot delete path `/nix/store/zq0h41l75vlb4z45kzgjjmsjxvcv1qk7-mesa-6.4'
    ↪ since it is still alive
```

Operation `--query`

Synopsis

`nix-store` { `--query` | `-q` } { `--outputs` | `--requisites` | `-R` | `--references` | `--referrers` | `--referrers-closure` | `--deriver` | `-d` | `--graph` | `--tree` | `--binding` *name* | `-b` *name* | `--hash` | `--size` | `--roots` } [`--use-output`] [`-u`] [`--force-realise`] [`-f`] *paths*...

Description

The operation `--query` displays various bits of information about the store paths . The queries are described below. At most one query can be specified. The default query is `--outputs`.

The paths *paths* may also be symlinks from outside of the Nix store, to the Nix store. In that case, the query is applied to the target of the symlink.

Common options

`--use-output, -u` For each argument to the query that is a store derivation, apply the query to the output path of the derivation instead.

`--force-realise, -f` Realise each argument to the query first (see **nix-store --realise**).

Queries

`--outputs` Prints out the output paths of the store derivations *paths*. These are the paths that will be produced when the derivation is built.

`--requisites, -R` Prints out the closure of the store path *paths*.

This query has one option:

`--include-outputs` Also include the output path of store derivations, and their closures.

This query can be used to implement various kinds of deployment. A *source deployment* is obtained by distributing the closure of a store derivation. A *binary deployment* is obtained by distributing the closure of an output path. A *cache deployment* (combined source/binary deployment, including binaries of build-time-only dependencies) is obtained by distributing the closure of a store derivation and specifying the option `--include-outputs`.

`--references` Prints the set of references of the store paths *paths*, that is, their immediate dependencies. (For *all* dependencies, use `--requisites`.)

`--referrers` Prints the set of *referrers* of the store paths *paths*, that is, the store paths currently existing in the Nix store that refer to one of *paths*. Note that contrary to the references, the set of referrers is not constant; it can change as store paths are added or removed.

`--referrers-closure` Prints the closure of the set of store paths *paths* under the referrers relation; that is, all store paths that directly or indirectly refer to one of *paths*. These are all the path currently in the Nix store that are dependent on *paths*.

`--deriver, -d` Prints the deriver of the store paths *paths*. If the path has no deriver (e.g., if it is a source file), or if the deriver is not known (e.g., in the case of a binary-only deployment), the string `unknown-deriver` is printed.

`--graph` Prints the references graph of the store paths *paths* in the format of the **dot** tool of AT&T's Graphviz package. This can be used to visualise dependency graphs. To obtain a build-time dependency graph, apply this to a store derivation. To obtain a runtime dependency graph, apply it to an output path.

`--tree` Prints the references graph of the store paths *paths* as a nested ASCII tree. References are ordered by descending closure size; this tends to flatten the tree, making it more readable. The query only recurses into a store path when it is first encountered; this prevents a blowup of the tree representation of the graph.

`--graphml` Prints the references graph of the store paths *paths* in the GraphML file format. This can be used to visualise dependency graphs. To obtain a build-time dependency graph, apply this to a store derivation. To obtain a runtime dependency graph, apply it to an output path.

`--binding` *name*, `-b` *name* Prints the value of the attribute *name* (i.e., environment variable) of the store derivations *paths*. It is an error for a derivation to not have the specified attribute.

`--hash` Prints the SHA-256 hash of the contents of the store paths *paths* (that is, the hash of the output of **nix-store --dump** on the given paths). Since the hash is stored in the Nix database, this is a fast operation.

`--size` Prints the size in bytes of the contents of the store paths *paths* -- to be precise, the size of the output of **nix-store --dump** on the given paths. Note that the actual disk space required by the store paths may be higher, especially on filesystems with large cluster sizes.

`--roots` Prints the garbage collector roots that point, directly or indirectly, at the store paths *paths*.

Examples

Print the closure (runtime dependencies) of the **svn** program in the current user environment:

```
$ nix-store -qR $(which svn)
/nix/store/5mbglq5ldqld8sj57273aljwkfvj22mc-subversion-1.1.4
/nix/store/9lz9yc6zgmc0vlqmn2ipcpkjlmbi51vv-glibc-2.3.4
...
```

Print the build-time dependencies of **svn**:

```
$ nix-store -qR $(nix-store -qd $(which svn))
/nix/store/02iizgn86m42q905rddvg4ja975bk2i4-grep-2.5.1.tar.bz2.drv
/nix/store/07a2bzxmzwz5hp58nf03pahrv2ygwgs3-gcc-wrapper.sh
/nix/store/0ma7c9wsbaxahww104gbw3fcd806ski4-glibc-2.3.4.drv
... lots of other paths ...
```

The difference with the previous example is that we ask the closure of the derivation (`-qd`), not the closure of the output path that contains **svn**.

Show the build-time dependencies as a tree:

```
$ nix-store -q --tree $(nix-store -qd $(which svn))
/nix/store/7i5082kfb6yjbqdbiwdhhza0am2xvh6c-subversion-1.1.4.drv
+---/nix/store/d8afh10z72n8l1cr5w42366abiblgn54-builder.sh
+---/nix/store/fmzxmpjx2lh849ph0l36snfj9zdibw67-bash-3.0.drv
|   +---/nix/store/570hmhmx3v57605cqg9yfvvyh0nnb8k8-bash
|   +---/nix/store/p3srsbd8dx44v2pg6nbnszab5mcwx03v-builder.sh
...
```

Show all paths that depend on the same OpenSSL library as **svn**:

```
$ nix-store -q --referrers $(nix-store -q --binding openssl $(nix-store -qd
    ↪ $(which svn)))
/nix/store/23ny9l9wixx21632y2wi4p585qhva1q8-sylpheed-1.0.0
/nix/store/5mbglq5ldqld8sj57273aljwkfvj22mc-subversion-1.1.4
/nix/store/dpmvp969yhdqs7lm2r1a3gng7pyq6vy4-subversion-1.1.3
/nix/store/l51240xqsgg8a7yrbqdx1rfzyv6l26fx-lynx-2.8.5
```

Show all paths that directly or indirectly depend on the Glibc (C library) used by **svn**:

```
$ nix-store -q --referrers-closure $(ldd $(which svn) | grep /libc.so | awk
    ↪ '{print $3}')
/nix/store/034a6h4vpz9kds5r6kzb9lhh81mscw43-libgnomeprintui-2.8.2
/nix/store/15l3yi0d45prm7a82pcrknxdh6nzmxza-gawk-3.1.4
...
```

Note that **ldd** is a command that prints out the dynamic libraries used by an ELF executable.

Make a picture of the runtime dependency graph of the current user environment:

```
$ nix-store -q --graph ~/.nix-profile | dot -Tps > graph.ps
$ gv graph.ps
```

Show every garbage collector root that points to a store path that depends on **svn**:

```
$ nix-store -q --roots $(which svn)
/nix/var/nix/profiles/default-81-link
/nix/var/nix/profiles/default-82-link
/nix/var/nix/profiles/per-user/eelco/profile-97-link
```

Operation `--add`

Synopsis

`nix-store --add` *paths*...

Description

The operation `--add` adds the specified paths to the Nix store. It prints the resulting paths in the Nix store on standard output.

Example

```
$ nix-store --add ./foo.c
/nix/store/m7lrha58ph6rcnv109yzx1nk1cj7k7zf-foo.c
```

Operation `--add-fixed`

Synopsis

`nix-store` [`--recursive`] `--add-fixed` *algorithm paths...*

Description

The operation `--add-fixed` adds the specified paths to the Nix store. Unlike `--add` paths are registered using the specified hashing algorithm, resulting in the same output path as a fixed output derivation. This can be used for sources that are not available from a public url or broke since the download expression was written.

This operation has the following options:

`--recursive` Use recursive instead of flat hashing mode, used when adding directories to the store.

Example

```
$ nix-store --add-fixed sha256 ./hello-2.10.tar.gz
/nix/store/3x7dwzq014bblazs7kq20p9hyzz0qh8g-hello-2.10.tar.gz
```

Operation `--verify`

Synopsis

`nix-store --verify` [`--check-contents`] [`--repair`]

Description

The operation `--verify` verifies the internal consistency of the Nix database, and the consistency between the Nix database and the Nix store. Any inconsistencies encountered are automatically repaired. Inconsistencies are generally the result of the Nix store or database being modified by non-Nix tools, or of bugs in Nix itself.

This operation has the following options:

`--check-contents` Checks that the contents of every valid store path has not been altered by computing a SHA-256 hash of the contents and comparing it with the hash stored in the Nix database at build time. Paths that have been modified are printed out. For large stores, `--check-contents` is obviously quite slow.

`--repair` If any valid path is missing from the store, or (if `--check-contents` is given) the contents of a valid path has been modified, then try to repair the path by redownloading it. See **nix-store --repair-path** for details.

Operation `--verify-path`

Synopsis

`nix-store --verify-path` *paths...*

Description

The operation `--verify-path` compares the contents of the given store paths to their cryptographic hashes stored in Nix's database. For every changed path, it prints a warning message. The exit status is 0 if no path has changed, and 1 otherwise.

Example

To verify the integrity of the **svn** command and all its dependencies:

```
$ nix-store --verify-path $(nix-store -qR $(which svn))
```

Operation `--repair-path`

Synopsis

`nix-store --repair-path` *paths*...

Description

The operation `--repair-path` attempts to "repair" the specified paths by redownloading them using the available substituters. If no substitutes are available, then repair is not possible.

Warning

During repair, there is a very small time window during which the old path (if it exists) is moved out of the way and replaced with the new path. If repair is interrupted in between, then the system may be left in a broken state (e.g., if the path contains a critical system component like the GNU C Library).

Example

```
$ nix-store --verify-path /nix/store/dj7a81wsm1ijwwpkks3725661h3263p5-glibc-2.13
path `/nix/store/dj7a81wsm1ijwwpkks3725661h3263p5-glibc-2.13' was modified!
  expected hash `2db57715ae90b7e31ff1f2ecb8c12ec1cc43da920efcbe3b22763f36a1861588',
  got `481c5aa5483ebc97c20457bb8bca24deea56550d3985cda0027f67fe54b808e4'

$ nix-store --repair-path /nix/store/dj7a81wsm1ijwwpkks3725661h3263p5-glibc-2.13
fetching path `/nix/store/d7a81wsm1ijwwpkks3725661h3263p5-glibc-2.13'...
...
```

Operation `--dump`

Synopsis

`nix-store --dump` *path*

Description

The operation `--dump` produces a NAR (Nix ARchive) file containing the contents of the file system tree rooted at *path*. The archive is written to standard output.

A NAR archive is like a TAR or Zip archive, but it contains only the information that Nix considers important. For instance, timestamps are elided because all files in the Nix store have their timestamp set to 0 anyway. Likewise, all permissions are left out except for the execute bit, because all files in the Nix store have 644 or 755 permission.

Also, a NAR archive is *canonical*, meaning that "equal" paths always produce the same NAR archive. For instance, directory entries are always sorted so that the actual on-disk order doesn't influence the result. This means that the cryptographic hash of a NAR dump of a path is usable as a fingerprint of the contents of the path. Indeed, the hashes of store paths stored in Nix's database (see `nix-store -q --hash`) are SHA-256 hashes of the NAR dump of each store path.

NAR archives support filenames of unlimited length and 64-bit file sizes. They can contain regular files, directories, and symbolic links, but not other types of files (such as device nodes).

A Nix archive can be unpacked using `nix-store --restore`.

Operation `--restore`

Synopsis

`nix-store --restore` *path*

Description

The operation `--restore` unpacks a NAR archive to *path*, which must not already exist. The archive is read from standard input.

Operation `--export`

Synopsis

`nix-store --export` *paths...*

Description

The operation `--export` writes a serialisation of the specified store paths to standard output in a format that can be imported into another Nix store with **nix-store --import**. This is like **nix-store --dump**, except that the NAR archive produced by that command doesn't contain the necessary meta-information to allow it to be imported into another Nix store (namely, the set of references of the path).

This command does not produce a *closure* of the specified paths, so if a store path references other store paths that are missing in the target Nix store, the import will fail. To copy a whole closure, do something like:

```
$ nix-store --export $(nix-store -qR paths) > out
```

To import the whole closure again, run:

```
$ nix-store --import < out
```

Operation `--import`

Synopsis

`nix-store --import`

Description

The operation `--import` reads a serialisation of a set of store paths produced by **nix-store --export** from standard input and adds those store paths to the Nix store. Paths that already exist in the Nix store are ignored. If a path refers to another path that doesn't exist in the Nix store, the import fails.

Operation `--optimise`

Synopsis

`nix-store --optimise`

Description

The operation `--optimise` reduces Nix store disk space usage by finding identical files in the store and hard-linking them to each other. It typically reduces the size of the store by something like 25-35%. Only regular files and symlinks are hard-linked in this manner. Files are considered identical when they have the same NAR archive serialisation: that is, regular files must have the same contents and permission (executable or non-executable), and symlinks must have the same contents.

After completion, or when the command is interrupted, a report on the achieved savings is printed on standard error.

Use `-vv` or `-vvv` to get some progress indication.

Example

```
$ nix-store --optimise
hashing files in `/nix/store/qhqx7l2f1kmwihc9bnxs7rc159hsxnf3-gcc-4.1.1'
...
541838819 bytes (516.74 MiB) freed by hard-linking 54143 files;
there are 114486 files with equal contents out of 215894 files in total
```

Operation `--read-log`

Synopsis

`nix-store` { `--read-log` | `-l` } *paths*...

Description

The operation `--read-log` prints the build log of the specified store paths on standard output. The build log is whatever the builder of a derivation wrote to standard output and standard error. If a store path is not a derivation, the deriver of the store path is used.

Build logs are kept in `/nix/var/log/nix/drvs`. However, there is no guarantee that a build log is available for any particular store path. For instance, if the path was downloaded as a pre-built binary through a substitute, then the log is unavailable.

Example

```
$ nix-store -l $(which ktorrent)
building /nix/store/dhc73pvzpnzxhdgpimsd9sw39di66ph1-ktorrent-2.2.1
unpacking sources
unpacking source archive
    ↪ /nix/store/p8n1jpqs27mgkjw07pb5269717nzf5f8-ktorrent-2.2.1.tar.gz
ktorrent-2.2.1/
ktorrent-2.2.1/NEWS
...
```

Operation `--dump-db`

Synopsis

`nix-store --dump-db` [*paths*...]

Description

The operation `--dump-db` writes a dump of the Nix database to standard output. It can be loaded into an empty Nix store using `--load-db`. This is useful for making backups and when migrating to different database schemas.

By default, `--dump-db` will dump the entire Nix database. When one or more store paths is passed, only the subset of the Nix database for those store paths is dumped. As with `--export`, the user is responsible for passing all the store paths for a closure. See `--export` for an example.

Operation `--load-db`

Synopsis

`nix-store --load-db`

Description

The operation `--load-db` reads a dump of the Nix database created by `--dump-db` from standard input and loads it into the Nix database.

Operation `--print-env`

Synopsis

`nix-store --print-env` *drvpath*

Description

The operation `--print-env` prints out the environment of a derivation in a format that can be evaluated by a shell. The command line arguments of the builder are placed in the variable `_args`.

Example

```
$ nix-store --print-env $(nix-instantiate '<nixpkgs>' -A firefox)
...
export src;
   ↪ src='/nix/store/plpj7qrwcz94z2psh6fchsi7s8yihc7k-firefox-12.0.source.tar.bz2'
export stdenv; stdenv='/nix/store/7c8asx3yfrg5dg1gzhzyq2236zfgibnn-stdenv'
export system; system='x86_64-linux'
export _args; _args='-e
   ↪ /nix/store/9krlzvny65gdc8s7kpb6lkx8cd02c25c-default-builder.sh'
```

Operation `--generate-binary-cache-key`

Synopsis

`nix-store --generate-binary-cache-key key-name secret-key-file public-key-file`

Description

This command generates an Ed25519 key pair that can be used to create a signed binary cache. It takes three mandatory parameters:

1. A key name, such as `cache.example.org-1`, that is used to look up keys on the client when it verifies signatures. It can be anything, but it's suggested to use the host name of your cache (e.g. `cache.example.org`) with a suffix denoting the number of the key (to be incremented every time you need to revoke a key).
2. The file name where the secret key is to be stored.
3. The file name where the public key is to be stored.

Chapter 23. Utilities

This section lists utilities that you can use when you work with Nix.

nix-channel

Name

nix-channel -- manage Nix channels

Synopsis

`nix-channel` { `--add` *`url`* [*`name`*] | `--remove` *`name`* | `--list` | `--update` [*`names...`*] | `--rollback` [*`generation`*] }

Description

A Nix channel is a mechanism that allows you to automatically stay up-to-date with a set of pre-built Nix expressions. A Nix channel is just a URL that points to a place containing a set of Nix expressions. See also Chapter 12, *Channels*.

This command has the following operations:

`--add` *`url`* [*`name`*] Adds a channel named *`name`* with URL *`url`* to the list of subscribed channels. If *`name`* is omitted, it defaults to the last component of *`url`*, with the suffixes `-stable` or `-unstable` removed.

`--remove` *`name`* Removes the channel named *`name`* from the list of subscribed channels.

`--list` Prints the names and URLs of all subscribed channels on standard output.

`--update` [*`names...`*] Downloads the Nix expressions of all subscribed channels (or only those included in *`names`* if specified) and makes them the default for **nix-env** operations (by symlinking them from the directory `~/.nix-defexpr`).

`--rollback` [*`generation`*] Reverts the previous call to **nix-channel --update**. Optionally, you can specify a specific channel generation number to restore.

Note that `--add` does not automatically perform an update.

The list of subscribed channels is stored in `~/.nix-channels`.

Examples

To subscribe to the Nixpkgs channel and install the GNU Hello package:

```
$ nix-channel --add https://nixos.org/channels/nixpkgs-unstable
$ nix-channel --update
$ nix-env -iA nixpkgs.hello
```

You can revert channel updates using `--rollback`:

```
$ nix-instantiate --eval -E '(import <nixpkgs> {}).lib.nixpkgsVersion'
"14.04.527.0e935f1"

$ nix-channel --rollback
switching from generation 483 to 482

$ nix-instantiate --eval -E '(import <nixpkgs> {}).lib.nixpkgsVersion'
"14.04.526.dbadfad"
```

Files

/nix/var/nix/profiles/per-user/username/channels nix-channel uses a **nix-env** profile to keep track of previous versions of the subscribed channels. Every time you run **nix-channel --update**, a new channel generation (that is, a symlink to the channel Nix expressions in the Nix store) is created. This enables **nix-channel --rollback** to revert to previous versions.

~/.nix-defexpr/channels This is a symlink to `/nix/var/nix/profiles/per-user/username`/channels. It ensures that **nix-env** can find your channels. In a multi-user installation, you may also have `~/.nix-defexpr/channels_root`, which links to the channels of the root user.

Channel format

A channel URL should point to a directory containing the following files:

nixexprs.tar.xz A tarball containing Nix expressions and files referenced by them (such as build scripts and patches). At the top level, the tarball should contain a single directory. That directory must contain a file `default.nix` that serves as the channel's "entry point".

---

nix-collect-garbage

Name

nix-collect-garbage -- delete unreachable store paths

Synopsis

`nix-collect-garbage` [`--delete-old`] [`-d`] [`--delete-older-than` *period*] [`--max-freed` *bytes*] [`--dry-run`]

Description

The command **nix-collect-garbage** is mostly an alias of **nix-store --gc**, that is, it deletes all unreachable paths in the Nix store to clean up your system. However, it provides two additional options: `-d` (`--delete-old`), which deletes all old generations of all profiles in `/nix/var/nix/profiles` by invoking `nix-env --delete-generations old` on all profiles (of course, this makes rollbacks to previous configurations impossible); and `--delete-older-than` *period*, where period is a value such as `30d`, which deletes all generations older than the specified number of days in all profiles in `/nix/var/nix/profiles` (except for the generations that were active at that point in time).

Example

To delete from the Nix store everything that is not used by the current generations of each profile, do

```
$ nix-collect-garbage -d
```

---

nix-copy-closure

Name

nix-copy-closure -- copy a closure to or from a remote machine via SSH

Synopsis

`nix-copy-closure` [`--to` | `--from`] [`--gzip`] [`--include-outputs`] [`--use-substitutes` | `-s`] [`-v`] *`user@machine`* *`paths`*

Description

nix-copy-closure gives you an easy and efficient way to exchange software between machines. Given one or more Nix store *`paths`* on the local machine, **nix-copy-closure** computes the closure of those paths (i.e. all their dependencies in the Nix store), and copies all paths in the closure to the remote machine via the **ssh** (Secure Shell) command. With the `--from`, the direction is reversed: the closure of *`paths`* on a remote machine is copied to the Nix store on the local machine.

This command is efficient because it only sends the store paths that are missing on the target machine.

Since **nix-copy-closure** calls **ssh**, you may be asked to type in the appropriate password or passphrase. In fact, you may be asked *twice* because **nix-copy-closure** currently connects twice to the remote machine, first to get the set of paths missing on the target machine, and second to send the dump of those paths. If this bothers you, use **ssh-agent**.

Options

`--to` Copy the closure of *`paths`* from the local Nix store to the Nix store on *`machine`*. This is the default.

`--from` Copy the closure of *`paths`* from the Nix store on *`machine`* to the local Nix store.

`--gzip` Enable compression of the SSH connection.

`--include-outputs` Also copy the outputs of store derivations included in the closure.

`--use-substitutes` / `-s` Attempt to download missing paths on the target machine using Nix's substitute mechanism. Any paths that cannot be substituted on the target are still copied normally from the source. This is useful, for instance, if the connection between the source and target machine is slow, but the connection between the target machine and `nixos.org` (the default binary cache server) is fast.

`-v` Show verbose output.

Environment variables

`NIX_SSHOPTS` Additional options to be passed to **ssh** on the command line.

Examples

Copy Firefox with all its dependencies to a remote machine:

```
$ nix-copy-closure --to alice@itchy.labs $(type -tP firefox)
```

Copy Subversion from a remote machine and then install it into a user environment:

```
1 $ nix-copy-closure --from alice@itchy.labs \
    /nix/store/0dj0503hjxy5mbwlafv1rsbdiyx1gkdy-subversion-1.4.4
3 $ nix-env -i /nix/store/0dj0503hjxy5mbwlafv1rsbdiyx1gkdy-subversion-1.4.4
```

---

nix-daemon

Name

nix-daemon -- Nix multi-user support daemon

Synopsis

`nix-daemon`

Description

The Nix daemon is necessary in multi-user Nix installations. It performs build actions and other operations on the Nix store on behalf of unprivileged users.

---

nix-hash

Name

nix-hash -- compute the cryptographic hash of a path

Synopsis

`nix-hash` [`--flat`] [`--base32`] [`--truncate`] [`--type` *hashAlgo*] *path*...

`nix-hash --to-base16` *hash*...

`nix-hash --to-base32` *hash*...

Description

The command **nix-hash** computes the cryptographic hash of the contents of each *path* and prints it on standard output. By default, it computes an MD5 hash, but other hash algorithms are available as well. The hash is printed in hexadecimal. To generate the same hash as **nix-prefetch-url** you have to specify multiple arguments, see below for an example.

The hash is computed over a *serialisation* of each path: a dump of the file system tree rooted at the path. This allows directories and symlinks to be hashed as well as regular files. The dump is in the *NAR format* produced by **nix-store** `--dump`. Thus, `nix-hash path` yields the same cryptographic hash as `nix-store --dump path` | md5sum.

Options

`--flat` Print the cryptographic hash of the contents of each regular file *path*. That is, do not compute the hash over the dump of *path*. The result is identical to that produced by the GNU commands **md5sum** and **sha1sum**.

`--base32` Print the hash in a base-32 representation rather than hexadecimal. This base-32 representation is more compact and can be used in Nix expressions (such as in calls to `fetchurl`).

`--truncate` Truncate hashes longer than 160 bits (such as SHA-256) to 160 bits.

`--type` *hashAlgo* Use the specified cryptographic hash algorithm, which can be one of `md5`, `sha1`, and `sha256`.

`--to-base16` Don't hash anything, but convert the base-32 hash representation *hash* to hexadecimal.

`--to-base32` Don't hash anything, but convert the hexadecimal hash representation *hash* to base-32.

Examples

Computing the same hash as **nix-prefetch-url**:

```
$ nix-prefetch-url file://<(echo test)
1lkgqb6fclns49861dwk9rzb6xnfkxbpws74mxnx01z9qyv1pjpj
$ nix-hash --type sha256 --flat --base32 <(echo test)
1lkgqb6fclns49861dwk9rzb6xnfkxbpws74mxnx01z9qyv1pjpj
```

Computing hashes:

```
$ mkdir test
$ echo "hello" > test/world

$ nix-hash test/ (MD5 hash; default)
8179d3caeff1869b5ba1744e5a245c04

$ nix-store --dump test/ | md5sum (for comparison)
8179d3caeff1869b5ba1744e5a245c04  -

$ nix-hash --type sha1 test/
e4fd8ba5f7bbeaea5ace89fe10255536cd60dab6

$ nix-hash --type sha1 --base32 test/
nvd61k9nalji1zl9rrdfmsmvyyjqpzg4

$ nix-hash --type sha256 --flat test/
error: reading file `test/': Is a directory

$ nix-hash --type sha256 --flat test/world
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03
```

Converting between hexadecimal and base-32:

```
$ nix-hash --type sha1 --to-base32 e4fd8ba5f7bbeaea5ace89fe10255536cd60dab6
nvd61k9nalji1zl9rrdfmsmvyyjqpzg4

$ nix-hash --type sha1 --to-base16 nvd61k9nalji1zl9rrdfmsmvyyjqpzg4
e4fd8ba5f7bbeaea5ace89fe10255536cd60dab6
```

---

nix-instantiate

Name

nix-instantiate -- instantiate store derivations from Nix expressions

Synopsis

`nix-instantiate` [`--parse` | `--eval` [`--strict`] [`--json`] [`--xml`]] [`--read-write-mode`] [`--arg` *name value*] [{ `--attr` | `-A` } *attrPath*] [`--add-root` *path*] [`--indirect`] [`--expr` | `-E`] *files*...

`nix-instantiate` `--find-file` *files*...

Description

The command **nix-instantiate** generates store derivations from (high-level) Nix expressions. It evaluates the Nix expressions in each of *files* (which defaults to *./default.nix*). Each top-level expression should evaluate to a derivation, a list of derivations, or a set of derivations. The paths of the resulting store derivations are printed on standard output.

If *files* is the character `-`, then a Nix expression will be read from standard input.

See also Chapter 20, *Common Options* for a list of common options.

Options

`--add-root` *path*, `--indirect` See the corresponding options in **nix-store**.

`--parse` Just parse the input files, and print their abstract syntax trees on standard output in ATerm format.

`--eval` Just parse and evaluate the input files, and print the resulting values on standard output. No instantiation of store derivations takes place.

`--find-file` Look up the given files in Nix's search path (as specified by the `NIX_PATH` environment variable). If found, print the corresponding absolute paths on standard output. For instance, if `NIX_PATH` is `nixpkgs=/home/alice/nixpkgs`, then `nix-instantiate --find-file nixpkgs/default.nix` will print `/home/alice/nixpkgs/default.nix`.

`--strict` When used with `--eval`, recursively evaluate list elements and attributes. Normally, such sub-expressions are left unevaluated (since the Nix expression language is lazy).

Warning

This option can cause non-termination, because lazy data structures can be infinitely large.

`--json` When used with `--eval`, print the resulting value as an JSON representation of the abstract syntax tree rather than as an ATerm.

`--xml` When used with `--eval`, print the resulting value as an XML representation of the abstract syntax tree rather than as an ATerm. The schema is the same as that used by the `toXML` built-in.

`--read-write-mode` When used with `--eval`, perform evaluation in read/write mode so nix language features that require it will still work (at the cost of needing to do instantiation of every evaluated derivation). If this option is not enabled, there may be uninstantiated store paths in the final output.

Examples

Instantiating store derivations from a Nix expression, and building them using **nix-store**:

```
$ nix-instantiate test.nix (instantiate)
/nix/store/cigxbmvy6dzix98dxxh9b6shg7ar5bvs-perl-BerkeleyDB-0.26.drv

$ nix-store -r $(nix-instantiate test.nix) (build)
...
/nix/store/qhqk4n8ci095g3sdp93x7rgwyh9rdvgk-perl-BerkeleyDB-0.26 (output path)

$ ls -l /nix/store/qhqk4n8ci095g3sdp93x7rgwyh9rdvgk-perl-BerkeleyDB-0.26
dr-xr-xr-x    2 eelco    users        4096 1970-01-01 01:00 lib
...
```

You can also give a Nix expression on the command line:

```
$ nix-instantiate -E 'with import <nixpkgs> { }; hello'
/nix/store/j8s4zyv75a724q38cb0r87rlczaiag4y-hello-2.8.drv
```

This is equivalent to:

```
$ nix-instantiate '<nixpkgs>' -A hello
```

Parsing and evaluating Nix expressions:

```
$ nix-instantiate --parse -E '1 + 2'
1 + 2

$ nix-instantiate --eval -E '1 + 2'
3

$ nix-instantiate --eval --xml -E '1 + 2'
<?xml version='1.0' encoding='utf-8'?>
<expr>
  <int value="3" />
</expr>
```

The difference between non-strict and strict evaluation:

```
$ nix-instantiate --eval --xml -E 'rec { x = "foo"; y = x; }'
...
  <attr name="x">
    <string value="foo" />
  </attr>
  <attr name="y">
    <unevaluated />
  </attr>
...
```

Note that `y` is left unevaluated (the XML representation doesn't attempt to show non-normal forms).

```
$ nix-instantiate --eval --xml --strict -E 'rec { x = "foo"; y = x; }'
...
  <attr name="x">
    <string value="foo" />
  </attr>
  <attr name="y">
    <string value="foo" />
  </attr>
...
```

---

nix-prefetch-url

Name

nix-prefetch-url -- copy a file from a URL into the store and print its hash

Synopsis

`nix-prefetch-url` [`--version`] [`--type` *`hashAlgo`*] [`--print-path`] [`--unpack`] [`--name` *`name`*] *`url`* [*`hash`*]

Description

The command **nix-prefetch-url** downloads the file referenced by the URL *`url`*, prints its cryptographic hash, and copies it into the Nix store. The file name in the store is `hash-`*`baseName`*, where *`baseName`* is everything following the final slash in *`url`*.

This command is just a convenience for Nix expression writers. Often a Nix expression fetches some source distribution from the network using the `fetchurl` expression contained in Nixpkgs. However, `fetchurl` requires a cryptographic hash. If you don't know the hash, you would have to download the file first, and then `fetchurl` would download it again when you build your Nix expression. Since `fetchurl` uses the same name for the downloaded file as **nix-prefetch-url**, the redundant download can be avoided.

If *`hash`* is specified, then a download is not performed if the Nix store already contains a file with the same hash and base name. Otherwise, the file is downloaded, and an error is signaled if the actual hash of the file does not match the specified hash.

This command prints the hash on standard output. Additionally, if the option `--print-path` is used, the path of the downloaded file in the Nix store is also printed.

Options

`--type` *`hashAlgo`* Use the specified cryptographic hash algorithm, which can be one of `md5`, `sha1`, and `sha256`.

`--print-path` Print the store path of the downloaded file on standard output.

`--unpack` Unpack the archive (which must be a tarball or zip file) and add the result to the Nix store. The resulting hash can be used with functions such as Nixpkgs's `fetchzip` or `fetchFromGitHub`.

`--name` *`name`* Override the name of the file in the Nix store. By default, this is `hash-`*`basename`*, where *`basename`* is the last component of *`url`*. Overriding the name is necessary when *`basename`* contains characters that are not allowed in Nix store paths.

Examples

```
$ nix-prefetch-url ftp://ftp.gnu.org/pub/gnu/hello/hello-2.10.tar.gz
0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i

$ nix-prefetch-url --print-path mirror://gnu/hello/hello-2.10.tar.gz
0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i
/nix/store/3x7dwzq014bblazs7kq20p9hyzz0qh8g-hello-2.10.tar.gz

$ nix-prefetch-url --unpack --print-path
    ↪ https://github.com/NixOS/patchelf/archive/0.8.tar.gz
079agjlv0hrv7fxnx9ngipx14gyncbkllxrp9cccnh3a50fxcmy7
/nix/store/19zrmhm3m40xxaw81c8cqm6aljgrnwj2-0.8.tar.gz
```

Chapter 24. Files

This section lists configuration files that you can use when you work with Nix.

nix.conf

Name

nix.conf -- Nix configuration file

Description

Nix reads settings from two configuration files:

- The system-wide configuration file `sysconfdir`/nix/nix.conf (i.e. `/etc/nix/nix.conf` on most systems), or `$NIX_CONF_DIR/nix.conf` if `NIX_CONF_DIR` is set.
- The user configuration file `$XDG_CONFIG_HOME/nix/nix.conf`, or `~/.config/nix/nix.conf` if `XDG_CONFIG_HOME` is not set.

The configuration files consist of `name` = *`value`* pairs, one per line. Other files can be included with a line like `include path`, where *`path`* is interpreted relative to the current conf file and a missing file is an error unless `!include` is used instead. Comments start with a `#` character. Here is an example configuration file:

```
keep-outputs = true       # Nice for developers
keep-derivations = true   # Idem
```

You can override settings on the command line using the `--option` flag, e.g. `--option keep-outputs false`.

The following settings are currently available:

`allowed-uris` A list of URI prefixes to which access is allowed in restricted evaluation mode. For example, when set to `https://github.com/NixOS`, builtin functions such as `fetchGit` are allowed to access `https://github.com/NixOS/patchelf.git`.

`allow-import-from-derivation` By default, Nix allows you to `import` from a derivation, allowing building at evaluation time. With this option set to false, Nix will throw an error when evaluating an expression that uses this feature, allowing users to ensure their evaluation will not require any builds to take place.

`allow-new-privileges` (Linux-specific.) By default, builders on Linux cannot acquire new privileges by calling setuid/setgid programs or programs that have file capabilities. For example, programs such as **sudo** or **ping** will fail. (Note that in sandbox builds, no such programs are available unless you bind-mount them into the sandbox via the `sandbox-paths` option.) You can allow the use of such programs by enabling this option. This is impure and usually undesirable, but may be useful in certain scenarios (e.g. to spin up containers or set up userspace network interfaces in tests).

`allowed-users` A list of names of users (separated by whitespace) that are allowed to connect to the Nix daemon. As with the `trusted-users` option, you can specify groups by prefixing them with `@`. Also, you can allow all users by specifying `*`. The default is `*`.

Note that trusted users are always allowed to connect.

`auto-optimise-store` If set to `true`, Nix automatically detects files in the store that have identical contents, and replaces them with hard links to a single copy. This saves disk space. If set to `false` (the default), you can still run **nix-store --optimise** to get rid of duplicate files.

`builders` A list of machines on which to perform builds. See Chapter 16, *Remote Builds* for details.

`builders-use-substitutes` If set to `true`, Nix will instruct remote build machines to use their own binary substitutes if available. In practical terms, this means that remote hosts will fetch as many build dependencies as possible from their own substitutes (e.g, from `cache.nixos.org`), instead of waiting for this host to upload them all. This can drastically reduce build times if the network connection between this computer and the remote build host is slow. Defaults to `false`.

`build-users-group` This options specifies the Unix group containing the Nix build user accounts. In multi-user Nix installations, builds should not be performed by the Nix account since that would allow users to arbitrarily modify the Nix store and database by supplying specially crafted builders; and they cannot be performed by the calling user since that would allow him/her to influence the build result.

Therefore, if this option is non-empty and specifies a valid group, builds will be performed under the user accounts that are a member of the group specified here (as listed in `/etc/group`). Those user accounts should not be used for any other purpose!

Nix will never run two builds under the same user account at the same time. This is to prevent an obvious security hole: a malicious user writing a Nix expression that modifies the build result of a legitimate Nix expression being built by another user. Therefore it is good to have as many Nix build user accounts as you can spare. (Remember: uids are cheap.)

The build users should have permission to create files in the Nix store, but not delete them. Therefore, `/nix/store` should be owned by the Nix account, its group should be the group specified here, and its mode should be `1775`.

If the build users group is empty, builds will be performed under the uid of the Nix process (that is, the uid of the caller if `NIX_REMOTE` is empty, the uid under which the Nix daemon runs if `NIX_REMOTE` is `daemon`). Obviously, this should not be used in multi-user settings with untrusted users.

`compress-build-log` If set to `true` (the default), build logs written to `/nix/var/log/nix/drvs` will be compressed on the fly using bzip2. Otherwise, they will not be compressed.

`connect-timeout` The timeout (in seconds) for establishing connections in the binary cache substituter. It corresponds to **curl**'s `--connect-timeout` option.

`cores` Sets the value of the `NIX_BUILD_CORES` environment variable in the invocation of builders. Builders can use this variable at their discretion to control the maximum amount of parallelism. For instance, in Nixpkgs, if the derivation attribute `enableParallelBuilding` is set to `true`, the builder passes the `-jN` flag to GNU Make. It can be overridden using the `--cores` command line switch and defaults to `1`. The value `0` means that the builder should use all available CPU cores in the system.

See also Chapter 17, *Tuning Cores and Jobs*.

`diff-hook` Absolute path to an executable capable of diffing build results. The hook executes if `run-diff-hook` is true, and the output of a build is known to not be the same. This program is not executed to determine if two results are the same.

The diff hook is executed by the same user and group who ran the build. However, the diff hook does not have write access to the store path just built.

The diff hook program receives three parameters:

1. A path to the previous build's results

2. A path to the current build's results
3. The path to the build's derivation
4. The path to the build's scratch directory. This directory will exist only if the build was run with `--keep-failed`.

The stderr and stdout output from the diff hook will not be displayed to the user. Instead, it will print to the nix-daemon's log.

When using the Nix daemon, `diff-hook` must be set in the `nix.conf` configuration file, and cannot be passed at the command line.

`enforce-determinism` See `repeat`.

`extra-sandbox-paths` A list of additional paths appended to `sandbox-paths`. Useful if you want to extend its default value.

`extra-platforms` Platforms other than the native one which this machine is capable of building for. This can be useful for supporting additional architectures on compatible machines: i686-linux can be built on x86_64-linux machines (and the default for this setting reflects this); armv7 is backwards-compatible with armv6 and armv5tel; some aarch64 machines can also natively run 32-bit ARM code; and qemu-user may be used to support non-native platforms (though this may be slow and buggy). Most values for this are not enabled by default because build systems will often misdetect the target platform and generate incompatible code, so you may wish to cross-check the results of using this option against proper natively-built versions of your derivations.

`extra-substituters` Additional binary caches appended to those specified in `substituters`. When used by unprivileged users, untrusted substituters (i.e. those not listed in `trusted-substituters`) are silently ignored.

`fallback` If set to `true`, Nix will fall back to building from source if a binary substitute fails. This is equivalent to the `--fallback` flag. The default is `false`.

`fsync-metadata` If set to `true`, changes to the Nix store metadata (in `/nix/var/nix/db`) are synchronously flushed to disk. This improves robustness in case of system crashes, but reduces performance. The default is `true`.

`hashed-mirrors` A list of web servers used by `builtins.fetchurl` to obtain files by hash. The default is `http://tarballs.nixos.org/`. Given a hash type *`ht`* and a base-16 hash *`h`*, Nix will try to download the file from `hashed-mirror/ht/`*`h`*. This allows files to be downloaded even if they have disappeared from their original URI. For example, given the default mirror `http://tarballs.nixos.org/`, when building the derivation

```
builtins.fetchurl {
  url = https://example.org/foo-1.2.3.tar.xz;
  sha256 = "2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae";
}
```

Nix will attempt to download this file from `http://tarballs.nixos.org/sha256/2c26b46b68ffc68ff99b453c1d30413413` first. If it is not available there, if will try the original URI.

`http-connections` The maximum number of parallel TCP connections used to fetch files from binary caches and by other downloads. It defaults to 25. 0 means no limit.

`keep-build-log` If set to `true` (the default), Nix will write the build log of a derivation (i.e. the standard output and error of its builder) to the directory `/nix/var/log/nix/drvs`. The build log can be retrieved using the command **nix-store -l** *`path`*.

`keep-derivations` If `true` (default), the garbage collector will keep the derivations from which non-garbage store paths were built. If `false`, they will be deleted unless explicitly registered as a root (or reachable from other roots).

Keeping derivation around is useful for querying and traceability (e.g., it allows you to ask with what dependencies or options a store path was built), so by default this option is on. Turn it off to save a bit of disk space (or a lot if `keep-outputs` is also turned on).

`keep-env-derivations` If `false` (default), derivations are not stored in Nix user environments. That is, the derivations of any build-time-only dependencies may be garbage-collected.

If `true`, when you add a Nix derivation to a user environment, the path of the derivation is stored in the user environment. Thus, the derivation will not be garbage-collected until the user environment generation is deleted (**nix-env --delete-generations**). To prevent build-time-only dependencies from being collected, you should also turn on `keep-outputs`.

The difference between this option and `keep-derivations` is that this one is "sticky": it applies to any user environment created while this option was enabled, while `keep-derivations` only applies at the moment the garbage collector is run.

`keep-outputs` If `true`, the garbage collector will keep the outputs of non-garbage derivations. If `false` (default), outputs will be deleted unless they are GC roots themselves (or reachable from other roots).

In general, outputs must be registered as roots separately. However, even if the output of a derivation is registered as a root, the collector will still delete store paths that are used only at build time (e.g., the C compiler, or source tarballs downloaded from the network). To prevent it from doing so, set this option to `true`.

`max-build-log-size` This option defines the maximum number of bytes that a builder can write to its stdout/stderr. If the builder exceeds this limit, it's killed. A value of `0` (the default) means that there is no limit.

`max-free` When a garbage collection is triggered by the `min-free` option, it stops as soon as `max-free` bytes are available. The default is infinity (i.e. delete all garbage).

`max-jobs` This option defines the maximum number of jobs that Nix will try to build in parallel. The default is `1`. The special value `auto` causes Nix to use the number of CPUs in your system. `0` is useful when using remote builders to prevent any local builds (except for `preferLocalBuild` derivation attribute which executes locally regardless). It can be overridden using the `--max-jobs` (`-j`) command line switch.

See also Chapter 17, *Tuning Cores and Jobs*.

`max-silent-time` This option defines the maximum number of seconds that a builder can go without producing any data on standard output or standard error. This is useful (for instance in an automated build system) to catch builds that are stuck in an infinite loop, or to catch remote builds that are hanging due to network problems. It can be overridden using the `--max-silent-time` command line switch.

The value `0` means that there is no timeout. This is also the default.

`min-free` When free disk space in `/nix/store` drops below `min-free` during a build, Nix performs a garbage-collection until `max-free` bytes are available or there is no more garbage. A value of `0` (the default) disables this feature.

`narinfo-cache-negative-ttl` The TTL in seconds for negative lookups. If a store path is queried from a substituter but was not found, there will be a negative lookup cached in the local disk cache database for the specified duration.

`narinfo-cache-positive-ttl` The TTL in seconds for positive lookups. If a store path is queried from a substituter, the result of the query will be cached in the local disk cache database including some of the NAR metadata. The default TTL is a month, setting a shorter TTL for positive lookups can be useful for binary caches that have frequent garbage collection, in which case having a more frequent cache invalidation would prevent trying to pull the path again and failing with a hash mismatch if the build isn't reproducible.

`netrc-file` If set to an absolute path to a `netrc` file, Nix will use the HTTP authentication credentials in this file when trying to download from a remote host through HTTP or HTTPS. Defaults to `$NIX_CONF_DIR/netrc`.

The `netrc` file consists of a list of accounts in the following format:

```
machine my-machine
login my-username
password my-password
```

For the exact syntax, see the `curl` documentation.

Note

This must be an absolute path, and `~` is not resolved. For example, `~/.netrc` won't resolve to your home directory's `.netrc`.

plugin-files A list of plugin files to be loaded by Nix. Each of these files will be dlopened by Nix, allowing them to affect execution through static initialization. In particular, these plugins may construct static instances of RegisterPrimOp to add new primops or constants to the expression language, RegisterStoreImplementation to add new store implementations, RegisterCommand to add new subcommands to the `nix` command, and RegisterSetting to add new nix config settings. See the constructors for those types for more details.

Since these files are loaded into the same address space as Nix itself, they must be DSOs compatible with the instance of Nix running at the time (i.e. compiled against the same headers, not linked to any incompatible libraries). They should not be linked to any Nix libs directly, as those will be available already at load time.

If an entry in the list is a directory, all files in the directory are loaded as plugins (non-recursively).

pre-build-hook If set, the path to a program that can set extra derivation-specific settings for this system. This is used for settings that can't be captured by the derivation model itself and are too variable between different versions of the same system to be hard-coded into nix.

The hook is passed the derivation path and, if sandboxes are enabled, the sandbox directory. It can then modify the sandbox and send a series of commands to modify various settings to stdout. The currently recognized commands are:

extra-sandbox-paths Pass a list of files and directories to be included in the sandbox for this build. One entry per line, terminated by an empty line. Entries have the same format as `sandbox-paths`.

post-build-hook Optional. The path to a program to execute after each build.

This option is only settable in the global `nix.conf`, or on the command line by trusted users.

When using the nix-daemon, the daemon executes the hook as `root`. If the nix-daemon is not involved, the hook runs as the user executing the nix-build.

- The hook executes after an evaluation-time build.
- The hook does not execute on substituted paths.
- The hook's output always goes to the user's terminal.
- If the hook fails, the build succeeds but no further builds execute.
- The hook executes synchronously, and blocks other builds from progressing while it runs.

The program executes with no arguments. The program's environment contains the following environment variables:

DRV_PATH The derivation for the built paths.

Example: `/nix/store/5nihn1a7pa8b25l9zafqaqibznlvvp3f-bash-4.4-p23.drv`

OUT_PATHS Output paths of the built derivation, separated by a space character.

Example: `/nix/store/zf5lbh336mnzf1nlswdn11g4n2m8zh3g-bash-4.4-p23-dev /nix/store/rjxwxwv1fpn9wa2x5`
`↪ /nix/store/6bqvbzjkcp9695dq0dpl5y43nvy37pq1-bash-4.4-p23-info /nix/store/r7fng3kk3vlpdlh2idnrbn`
`↪ /nix/store/xfghy8ixrhz3kyy6p724iv3cxji088dx-bash-4.4-p23`.

See Chapter 19, *Using the `post-build-hook`* for an example implementation.

repeat How many times to repeat builds to check whether they are deterministic. The default value is 0. If the value is non-zero, every build is repeated the specified number of times. If the contents of any of the runs differs from the previous ones and `enforce-determinism` is true, the build is rejected and the resulting store paths are not registered as "valid" in Nix's database.

require-sigs If set to `true` (the default), any non-content-addressed path added or copied to the Nix store (e.g. when substituting from a binary cache) must have a valid signature, that is, be signed using one of the keys listed in `trusted-public-keys` or `secret-key-files`. Set to `false` to disable signature checking.

`restrict-eval` If set to `true`, the Nix evaluator will not allow access to any files outside of the Nix search path (as set via the `NIX_PATH` environment variable or the `-I` option), or to URIs outside of `allowed-uri`. The default is `false`.

`run-diff-hook` If true, enable the execution of `diff-hook`.

When using the Nix daemon, `run-diff-hook` must be set in the `nix.conf` configuration file, and cannot be passed at the command line.

`sandbox` If set to `true`, builds will be performed in a *sandboxed environment*, i.e., they're isolated from the normal file system hierarchy and will only see their dependencies in the Nix store, the temporary build directory, private versions of `/proc`, `/dev`, `/dev/shm` and `/dev/pts` (on Linux), and the paths configured with the `sandbox-paths` option. This is useful to prevent undeclared dependencies on files in directories such as `/usr/bin`. In addition, on Linux, builds run in private PID, mount, network, IPC and UTS namespaces to isolate them from other processes in the system (except that fixed-output derivations do not run in private network namespace to ensure they can access the network).

Currently, sandboxing only work on Linux and macOS. The use of a sandbox requires that Nix is run as root (so you should use the "build users" feature to perform the actual builds under different users than root).

If this option is set to `relaxed`, then fixed-output derivations and derivations that have the `__noChroot` attribute set to `true` do not run in sandboxes.

The default is `true` on Linux and `false` on all other platforms.

`sandbox-dev-shm-size` This option determines the maximum size of the `tmpfs` filesystem mounted on `/dev/shm` in Linux sandboxes. For the format, see the description of the `size` option of `tmpfs` in mount(8). The default is `50%`.

`sandbox-paths` A list of paths bind-mounted into Nix sandbox environments. You can use the syntax `target=`*`source`* to mount a path in a different location in the sandbox; for instance, `/bin=/nix-bin` will mount the path `/nix-bin` as `/bin` inside the sandbox. If *`source`* is followed by `?`, then it is not an error if *`source`* does not exist; for example, `/dev/nvidiactl?` specifies that `/dev/nvidiactl` will only be mounted in the sandbox if it exists in the host filesystem.

Depending on how Nix was built, the default value for this option may be empty or provide `/bin/sh` as a bind-mount of **bash**.

`secret-key-files` A whitespace-separated list of files containing secret (private) keys. These are used to sign locally-built paths. They can be generated using **nix-store --generate-binary-cache-key**. The corresponding public key can be distributed to other users, who can add it to `trusted-public-keys` in their `nix.conf`.

`show-trace` Causes Nix to print out a stack trace in case of Nix expression evaluation errors.

`substitute` If set to `true` (default), Nix will use binary substitutes if available. This option can be disabled to force building from source.

`stalled-download-timeout` The timeout (in seconds) for receiving data from servers during download. Nix cancels idle downloads after this timeout's duration.

`substituters` A list of URLs of substituters, separated by whitespace. The default is `https://cache.nixos.org`.

`system` This option specifies the canonical Nix system name of the current installation, such as `i686-linux` or `x86_64-darwin`. Nix can only build derivations whose `system` attribute equals the value specified here. In general, it never makes sense to modify this value from its default, since you can use it to 'lie' about the platform you are building on (e.g., perform a Mac OS build on a Linux machine; the result would obviously be wrong). It only makes sense if the Nix binaries can run on multiple platforms, e.g., 'universal binaries' that run on `x86_64-linux` and `i686-linux`.

It defaults to the canonical Nix system name detected by `configure` at build time.

`system-features` A set of system "features" supported by this machine, e.g. `kvm`. Derivations can express a dependency on such features through the derivation attribute `requiredSystemFeatures`. For example, the attribute

```
requiredSystemFeatures = [ "kvm" ];
```

ensures that the derivation can only be built on a machine with the `kvm` feature.

This setting by default includes `kvm` if `/dev/kvm` is accessible, and the pseudo-features `nixos-test`, `benchmark` and `big-parallel` that are used in Nixpkgs to route builds to specific machines.

`tarball-ttl` Default: `3600` seconds.

The number of seconds a downloaded tarball is considered fresh. If the cached tarball is stale, Nix will check whether it is still up to date using the ETag header. Nix will download a new version if the ETag header is unsupported, or the cached ETag doesn't match.

Setting the TTL to `0` forces Nix to always check if the tarball is up to date.

Nix caches tarballs in `$XDG_CACHE_HOME/nix/tarballs`.

Files fetched via `NIX_PATH`, `fetchGit`, `fetchMercurial`, `fetchTarball`, and `fetchurl` respect this TTL.

`timeout` This option defines the maximum number of seconds that a builder can run. This is useful (for instance in an automated build system) to catch builds that are stuck in an infinite loop but keep writing to their standard output or standard error. It can be overridden using the `--timeout` command line switch.

The value `0` means that there is no timeout. This is also the default.

`trace-function-calls` Default: `false`.

If set to `true`, the Nix evaluator will trace every function call. Nix will print a log message at the "vomit" level for every function entrance and function exit.

```
function-trace entered undefined position at 1565795816999559622
function-trace exited undefined position at 1565795816999581277
function-trace entered /nix/store/.../example.nix:226:41 at 1565795253249935150
function-trace exited /nix/store/.../example.nix:226:41 at 1565795253249941684
```

The `undefined position` means the function call is a builtin.

Use the `contrib/stack-collapse.py` script distributed with the Nix source code to convert the trace logs in to a format suitable for **flamegraph.pl**.

`trusted-public-keys` A whitespace-separated list of public keys. When paths are copied from another Nix store (such as a binary cache), they must be signed with one of these keys. For example: `cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY= hydra.nixos.org-1:CNHJZBh9K4tP3EKF6Fkk`

`trusted-substituters` A list of URLs of substituters, separated by whitespace. These are not used by default, but can be enabled by users of the Nix daemon by specifying `--option substituters urls` on the command line. Unprivileged users are only allowed to pass a subset of the URLs listed in `substituters` and `trusted-substituters`.

`trusted-users` A list of names of users (separated by whitespace) that have additional rights when connecting to the Nix daemon, such as the ability to specify additional binary caches, or to import unsigned NARs. You can also specify groups by prefixing them with `@`; for instance, `@wheel` means all users in the `wheel` group. The default is `root`.

Warning

Adding a user to `trusted-users` is essentially equivalent to giving that user root access to the system. For example, the user can set `sandbox-paths` and thereby obtain read access to directories that are otherwise inacessible to them.

Deprecated Settings

`binary-caches` *Deprecated:* `binary-caches` is now an alias to `substituters`.

`binary-cache-public-keys` *Deprecated:* `binary-cache-public-keys` is now an alias to `trusted-public-keys`.

`build-compress-log` *Deprecated:* `build-compress-log` is now an alias to `compress-build-log`.

`build-cores` *Deprecated:* `build-cores` is now an alias to `cores`.

`build-extra-chroot-dirs` *Deprecated:* `build-extra-chroot-dirs` is now an alias to `extra-sandbox-paths`.

`build-extra-sandbox-paths` *Deprecated:* `build-extra-sandbox-paths` is now an alias to `extra-sandbox-paths`.

`build-fallback` *Deprecated:* `build-fallback` is now an alias to `fallback`.

`build-max-jobs` *Deprecated:* `build-max-jobs` is now an alias to `max-jobs`.

`build-max-log-size` *Deprecated:* `build-max-log-size` is now an alias to `max-build-log-size`.

`build-max-silent-time` *Deprecated:* `build-max-silent-time` is now an alias to `max-silent-time`.

`build-repeat` *Deprecated:* `build-repeat` is now an alias to `repeat`.

`build-timeout` *Deprecated:* `build-timeout` is now an alias to `timeout`.

`build-use-chroot` *Deprecated:* `build-use-chroot` is now an alias to `sandbox`.

`build-use-sandbox` *Deprecated:* `build-use-sandbox` is now an alias to `sandbox`.

`build-use-substitutes` *Deprecated:* `build-use-substitutes` is now an alias to `substitute`.

`gc-keep-derivations` *Deprecated:* `gc-keep-derivations` is now an alias to `keep-derivations`.

`gc-keep-outputs` *Deprecated:* `gc-keep-outputs` is now an alias to `keep-outputs`.

`env-keep-derivations` *Deprecated:* `env-keep-derivations` is now an alias to `keep-env-derivations`.

`extra-binary-caches` *Deprecated:* `extra-binary-caches` is now an alias to `extra-substituters`.

`trusted-binary-caches` *Deprecated:* `trusted-binary-caches` is now an alias to `trusted-substituters`.

Appendix A. Glossary

derivation A description of a build action. The result of a derivation is a store object. Derivations are typically specified in Nix expressions using the `derivation` primitive. These are translated into low-level *store derivations* (implicitly by **nix-env** and **nix-build**, or explicitly by **nix-instantiate**).

store The location in the file system where store objects live. Typically `/nix/store`.

store path The location in the file system of a store object, i.e., an immediate child of the Nix store directory.

store object A file that is an immediate child of the Nix store directory. These can be regular files, but also entire directory trees. Store objects can be sources (objects copied from outside of the store), derivation outputs (objects produced by running a build action), or derivations (files describing a build action).

substitute A substitute is a command invocation stored in the Nix database that describes how to build a store object, bypassing the normal build mechanism (i.e., derivations). Typically, the substitute builds the store object by downloading a pre-built version of the store object from some server.

purity The assumption that equal Nix derivations when run always produce the same output. This cannot be guaranteed in general (e.g., a builder can rely on external inputs such as the network or the system time) but the Nix model assumes it.

Nix expression A high-level description of software packages and compositions thereof. Deploying software using Nix entails writing Nix expressions for your packages. Nix expressions are translated to derivations that are stored in the Nix store. These derivations can then be built.

reference A store path `P` is said to have a reference to a store path `Q` if the store object at `P` contains the path `Q` somewhere. The *references* of a store path are the set of store paths to which it has a reference.

A derivation can reference other derivations and sources (but not output paths), whereas an output path only references other output paths.

reachable A store path `Q` is reachable from another store path `P` if `Q` is in the closure of the references relation.

closure The closure of a store path is the set of store paths that are directly or indirectly "reachable" from that store path; that is, it's the closure of the path under the references relation. For a package, the closure of its derivation is equivalent to the build-time dependencies, while the closure of its output path is equivalent to its runtime dependencies. For correct deployment it is necessary to deploy whole closures, since otherwise at runtime files could be missing. The command **nix-store -qR** prints out closures of store paths.

As an example, if the store object at path `P` contains a reference to path `Q`, then `Q` is in the closure of `P`. Further, if `Q` references `R` then `R` is also in the closure of `P`.

output path A store path produced by a derivation.

deriver The deriver of an output path is the store derivation that built it.

validity A store path is considered *valid* if it exists in the file system, is listed in the Nix database as being valid, and if all paths in its closure are also valid.

user environment An automatically generated store object that consists of a set of symlinks to "active" applications, i.e., other store paths. These are generated automatically by **nix-env**. See Chapter 10, *Profiles*.

profile A symlink to the current user environment of a user, e.g., `/nix/var/nix/profiles/default`.

NAR A *N*ix *AR*chive. This is a serialisation of a path in the Nix store. It can contain regular files, directories and symbolic links. NARs are generated and unpacked using **nix-store --dump** and **nix-store --restore**.

Appendix B. Hacking

This section provides some notes on how to hack on Nix. To get the latest version of Nix from GitHub:

```
$ git clone git://github.com/NixOS/nix.git
$ cd nix
```

To build it and its dependencies:

```
$ nix-build release.nix -A build.x86_64-linux
```

To build all dependencies and start a shell in which all environment variables are set up so that those dependencies can be found:

```
$ nix-shell
```

To build Nix itself in this shell:

```
[nix-shell]$ ./bootstrap.sh
[nix-shell]$ configurePhase
[nix-shell]$ make
```

To install it in `$(pwd)/inst` and test it:

```
[nix-shell]$ make install
[nix-shell]$ make installcheck
```

Appendix C. Nix Release Notes

C.1. Release 2.3 (2019-09-04)

This is primarily a bug fix release. However, it makes some incompatible changes:

- Nix now uses BSD file locks instead of POSIX file locks. Because of this, you should not use Nix 2.3 and previous releases at the same time on a Nix store.

It also has the following changes:

- `builtins.fetchGit`'s `ref` argument now allows specifying an absolute remote ref. Nix will automatically prefix `ref` with `refs/heads` only if `ref` doesn't already begin with `refs/`.
- The installer now enables sandboxing by default on Linux when the system has the necessary kernel support.
- The `max-jobs` setting now defaults to 1.
- New builtin functions: `builtins.isPath`, `builtins.hashFile`.
- The **nix** command has a new `--print-build-logs` (`-L`) flag to print build log output to stderr, rather than showing the last log line in the progress bar. To distinguish between concurrent builds, log lines are prefixed by the name of the package.
- Builds are now executed in a pseudo-terminal, and the `TERM` environment variable is set to `xterm-256color`. This allows many programs (e.g. **gcc**, **clang**, **cmake**) to print colorized log output.
- Add `--no-net` convenience flag. This flag disables substituters; sets the `tarball-ttl` setting to infinity (ensuring that any previously downloaded files are considered current); and disables retrying downloads and sets the connection timeout to the minimum. This flag is enabled automatically if there are no configured non-loopback network interfaces.
- Add a `post-build-hook` setting to run a program after a build has succeeded.
- Add a `trace-function-calls` setting to log the duration of Nix function calls to stderr.

C.2. Release 2.2 (2019-01-11)

This is primarily a bug fix release. It also has the following changes:

- In derivations that use structured attributes (i.e. that specify set the `__structuredAttrs` attribute to `true` to cause all attributes to be passed to the builder in JSON format), you can now specify closure checks per output, e.g.:

```
outputChecks."out" = {
  # The closure of 'out' must not be larger than 256 MiB.
  maxClosureSize = 256 * 1024 * 1024;

  # It must not refer to C compiler or to the 'dev' output.
  disallowedRequisites = [ stdenv.cc "dev" ];
};

outputChecks."dev" = {
  # The 'dev' output must not be larger than 128 KiB.
  maxSize = 128 * 1024;
};
```

- The derivation attribute `requiredSystemFeatures` is now enforced for local builds, and not just to route builds to remote builders. The supported features of a machine can be specified through the configuration setting `system-features`.

  By default, `system-features` includes `kvm` if `/dev/kvm` exists. For compatibility, it also includes the pseudo-features `nixos-test`, `benchmark` and `big-parallel` which are used by Nixpkgs to route builds to particular Hydra build machines.
- Sandbox builds are now enabled by default on Linux.
- The new command **nix doctor** shows potential issues with your Nix installation.
- The `fetchGit` builtin function now uses a caching scheme that puts different remote repositories in distinct local repositories, rather than a single shared repository. This may require more disk space but is faster.
- The `dirOf` builtin function now works on relative paths.
- Nix now supports SRI hashes, allowing the hash algorithm and hash to be specified in a single string. For example, you can write:

```
import <nix/fetchurl.nix> {
  url = https://nixos.org/releases/nix/nix-2.1.3/nix-2.1.3.tar.xz;
  hash = "sha256-XSLa0FjVyADWWhFfkZ2iKTjFDda6mMXjoYMXLRSYQKQ=";
};
```

instead of

```
import <nix/fetchurl.nix> {
  url = https://nixos.org/releases/nix/nix-2.1.3/nix-2.1.3.tar.xz;
  sha256 = "5d22dad058d5c800d65a115f919da22938c50dd6ba98c5e3a183172d149840a4";
};
```

In fixed-output derivations, the `outputHashAlgo` attribute is no longer mandatory if `outputHash` specifies the hash.

nix hash-file and **nix hash-path** now print hashes in SRI format by default. They also use SHA-256 by default instead of SHA-512 because that's what we use most of the time in Nixpkgs.

- Integers are now 64 bits on all platforms.
- The evaluator now prints profiling statistics (enabled via the `NIX_SHOW_STATS` and `NIX_COUNT_CALLS` environment variables) in JSON format.
- The option `--xml` in **nix-store --query** has been removed. Instead, there now is an option `--graphml` to output the dependency graph in GraphML format.
- All `nix-*` commands are now symlinks to `nix`. This saves a bit of disk space.
- **nix repl** now uses `libeditline` or `libreadline`.

C.3. Release 2.1 (2018-09-02)

This is primarily a bug fix release. It also reduces memory consumption in certain situations. In addition, it has the following new features:

- The Nix installer will no longer default to the Multi-User installation for macOS. You can still instruct the installer to run in multi-user mode.
- The Nix installer now supports performing a Multi-User installation for Linux computers which are running systemd. You can select a Multi-User installation by passing the `--daemon` flag to the installer: **sh <(curl https://nixos.org/nix/install) --daemon**.

  The multi-user installer cannot handle systems with SELinux. If your system has SELinux enabled, you can force the installer to run in single-user mode.
- New builtin functions: `builtins.bitAnd`, `builtins.bitOr`, `builtins.bitXor`, `builtins.fromTOML`, `builtins.concatMap`, `builtins.mapAttrs`.
- The S3 binary cache store now supports uploading NARs larger than 5 GiB.
- The S3 binary cache store now supports uploading to S3-compatible services with the `endpoint` option.
- The flag `--fallback` is no longer required to recover from disappeared NARs in binary caches.
- **nix-daemon** now respects `--store`.
- **nix run** now respects `nix-support/propagated-user-env-packages`.

This release has contributions from Adrien Devresse, Aleksandr Pashkov, Alexandre Esteves, Amine Chikhaoui, Andrew Dunham, Asad Saeeduddin, aszlig, Ben Challenor, Ben Gamari, Benjamin Hipple, Bogdan Seniuc, Corey O'Connor, Daiderd Jordan, Daniel Peebles, Daniel Poelzleithner, Danylo Hlynskyi, Dmitry Kalinkin, Domen Kožar, Doug Beardsley, Eelco Dolstra, Erik Arvstedt, Félix Baylac-Jacqué, Gleb Peregud, Graham Christensen, Guillaume Maudoux, Ivan Kozik, John Arnold, Justin Humm, Linus Heckemann, Lorenzo Manacorda, Matthew Justin Bauer, Matthew O'Gorman, Maximilian Bosch, Michael Bishop, Michael Fiano, Michael Mercier, Michael Raskin, Michael Weiss, Nicolas Dudebout, Peter Simons, Ryan Trinkle, Samuel Dionne-Riel, Sean Seefried, Shea Levy, Symphorien Gibol, Tim Engler, Tim Sears, Tuomas Tynkkynen, volth, Will Dietz, Yorick van Pelt and zimbatm.

C.4. Release 2.0 (2018-02-22)

The following incompatible changes have been made:

- The manifest-based substituter mechanism (**download-using-manifests**) has been removed. It has been superseded by the binary cache substituter mechanism since several years. As a result, the following programs have been removed:
  - **nix-pull**
  - **nix-generate-patches**
  - **bsdiff**
  - **bspatch**
- The "copy from other stores" substituter mechanism (**copy-from-other-stores** and the `NIX_OTHER_STORES` environment variable) has been removed. It was primarily used by the NixOS installer to copy available paths from the installation medium. The replacement is to use a chroot store as a substituter (e.g. `--substituters /mnt`), or to build into a chroot store (e.g. `--store /mnt --substituters /`).
- The command **nix-push** has been removed as part of the effort to eliminate Nix's dependency on Perl. You can use **nix copy** instead, e.g. `nix copy --to file:///tmp/my-binary-cache paths...`
- The "nested" log output feature (`--log-type pretty`) has been removed. As a result, **nix-log2xml** was also removed.
- OpenSSL-based signing has been removed. This feature was never well-supported. A better alternative is provided by the `secret-key-files` and `trusted-public-keys` options.
- Failed build caching has been removed. This feature was introduced to support the Hydra continuous build system, but Hydra no longer uses it.
- `nix-mode.el` has been removed from Nix. It is now a separate repository and can be installed through the MELPA package repository.

This release has the following new features:

- It introduces a new command named **nix**, which is intended to eventually replace all **nix-*** commands with a more consistent and better designed user interface. It currently provides replacements for some (but not all) of the functionality provided by **nix-store**, **nix-build**, **nix-shell -p**, **nix-env -qa**, **nix-instantiate --eval**, **nix-push** and **nix-copy-closure**. It has the following major features:
  - Unlike the legacy commands, it has a consistent way to refer to packages and package-like arguments (like store paths). For example, the following commands all copy the GNU Hello package to a remote machine:

    ```
    nix copy --to ssh://machine nixpkgs.hello
    ```

    ```
    1 nix copy --to ssh://machine /nix/store/0i2jd68mp5g6h2sa5k9c85rb80sn8hi9-hello-2.10
    ```

    ```
    1 nix copy --to ssh://machine '(with import <nixpkgs> {}; hello)'
    ```

    By contrast, **nix-copy-closure** only accepted store paths as arguments.
  - It is self-documenting: `--help` shows all available command-line arguments. If `--help` is given after a subcommand, it shows examples for that subcommand. **nix --help-config** shows all configuration options.
  - It is much less verbose. By default, it displays a single-line progress indicator that shows how many packages are left to be built or downloaded, and (if there are running builds) the most recent line of builder output. If a build fails, it shows the last few lines of builder output. The full build log can be retrieved using **nix log**.
  - It provides all `nix.conf` configuration options as command line flags. For example, instead of `--option http-connections 100` you can write `--http-connections 100`. Boolean options can be written as `--foo` or `--no-foo` (e.g. `--no-auto-optimise-store`).

– Many subcommands have a `--json` flag to write results to stdout in JSON format.

Warning

Please note that the **nix** command is a work in progress and the interface is subject to change.

It provides the following high-level ("porcelain") subcommands:

– **nix build** is a replacement for **nix-build**.

– **nix run** executes a command in an environment in which the specified packages are available. It is (roughly) a replacement for **nix-shell -p**. Unlike that command, it does not execute the command in a shell, and has a flag (**-c**) that specifies the unquoted command line to be executed.

It is particularly useful in conjunction with chroot stores, allowing Linux users who do not have permission to install Nix in **/nix/store** to still use binary substitutes that assume **/nix/store**. For example,

```
nix run --store ~/my-nix nixpkgs.hello -c hello --greeting 'Hi everybody!'
```

downloads (or if not substitutes are available, builds) the GNU Hello package into `~/my-nix/nix/store`, then runs **hello** in a mount namespace where `~/my-nix/nix/store` is mounted onto **/nix/store**.

– **nix search** replaces **nix-env -qa**. It searches the available packages for occurrences of a search string in the attribute name, package name or description. Unlike **nix-env -qa**, it has a cache to speed up subsequent searches.

– **nix copy** copies paths between arbitrary Nix stores, generalising **nix-copy-closure** and **nix-push**.

– **nix repl** replaces the external program **nix-repl**. It provides an interactive environment for evaluating and building Nix expressions. Note that it uses `linenoise-ng` instead of GNU Readline.

– **nix upgrade-nix** upgrades Nix to the latest stable version. This requires that Nix is installed in a profile. (Thus it won't work on NixOS, or if it's installed outside of the Nix store.)

– **nix verify** checks whether store paths are unmodified and/or "trusted" (see below). It replaces **nix-store --verify** and **nix-store --verify-path**.

– **nix log** shows the build log of a package or path. If the build log is not available locally, it will try to obtain it from the configured substituters (such as `cache.nixos.org`, which now provides build logs).

– **nix edit** opens the source code of a package in your editor.

– **nix eval** replaces **nix-instantiate --eval**.

– **nix why-depends** shows why one store path has another in its closure. This is primarily useful to finding the causes of closure bloat. For example,

```
nix why-depends nixpkgs.vlc nixpkgs.libdrm.dev
```

shows a chain of files and fragments of file contents that cause the VLC package to have the "dev" output of `libdrm` in its closure -- an undesirable situation.

– **nix path-info** shows information about store paths, replacing **nix-store -q**. A useful feature is the option `--closure-size` (`-S`). For example, the following command show the closure sizes of every path in the current NixOS system closure, sorted by size:

```
nix path-info -rS /run/current-system | sort -nk2
```

– **nix optimise-store** replaces **nix-store --optimise**. The main difference is that it has a progress indicator.

A number of low-level ("plumbing") commands are also available:

– **nix ls-store** and **nix ls-nar** list the contents of a store path or NAR file. The former is primarily useful in conjunction with remote stores, e.g.

```
1 nix ls-store --store https://cache.nixos.org/ -lR
    ↪ /nix/store/0i2jd68mp5g6h2sa5k9c85rb80sn8hi9-hello-2.10
```

lists the contents of path in a binary cache.

- **nix cat-store** and **nix cat-nar** allow extracting a file from a store path or NAR file.
- **nix dump-path** writes the contents of a store path to stdout in NAR format. This replaces **nix-store --dump**.
- **nix show-derivation** displays a store derivation in JSON format. This is an alternative to **pp-aterm**.
- **nix add-to-store** replaces **nix-store --add**.
- **nix sign-paths** signs store paths.
- **nix copy-sigs** copies signatures from one store to another.
- **nix show-config** shows all configuration options and their current values.

- The store abstraction that Nix has had for a long time to support store access via the Nix daemon has been extended significantly. In particular, substituters (which used to be external programs such as **download-from-binary-cache**) are now subclasses of the abstract `Store` class. This allows many Nix commands to operate on such store types. For example, **nix path-info** shows information about paths in your local Nix store, while **nix path-info --store https://cache.nixos.org/** shows information about paths in the specified binary cache. Similarly, **nix-copy-closure**, **nix-push** and substitution are all instances of the general notion of copying paths between different kinds of Nix stores.

  Stores are specified using an URI-like syntax, e.g. `https://cache.nixos.org/` or `ssh://machine`. The following store types are supported:

  - `LocalStore` (stori URI `local` or an absolute path) and the misnamed `RemoteStore` (`daemon`) provide access to a local Nix store, the latter via the Nix daemon. You can use `auto` or the empty string to auto-select a local or daemon store depending on whether you have write permission to the Nix store. It is no longer necessary to set the `NIX_REMOTE` environment variable to use the Nix daemon.

    As noted above, `LocalStore` now supports chroot builds, allowing the "physical" location of the Nix store (e.g. `/home/alice/nix/store`) to differ from its "logical" location (typically `/nix/store`). This allows non-root users to use Nix while still getting the benefits from prebuilt binaries from `cache.nixos.org`.
  - `BinaryCacheStore` is the abstract superclass of all binary cache stores. It supports writing build logs and NAR content listings in JSON format.
  - `HttpBinaryCacheStore` (`http://`, `https://`) supports binary caches via HTTP or HTTPS. If the server supports `PUT` requests, it supports uploading store paths via commands such as **nix copy**.
  - `LocalBinaryCacheStore` (`file://`) supports binary caches in the local filesystem.
  - `S3BinaryCacheStore` (`s3://`) supports binary caches stored in Amazon S3, if enabled at compile time.
  - `LegacySSHStore` (`ssh://`) is used to implement remote builds and **nix-copy-closure**.
  - `SSHStore` (`ssh-ng://`) supports arbitrary Nix operations on a remote machine via the same protocol used by **nix-daemon**.

- Security has been improved in various ways:
  - Nix now stores signatures for local store paths. When paths are copied between stores (e.g., copied from a binary cache to a local store), signatures are propagated.

    Locally-built paths are signed automatically using the secret keys specified by the `secret-key-files` store option. Secret/public key pairs can be generated using **nix-store --generate-binary-cache-key**.

    In addition, locally-built store paths are marked as "ultimately trusted", but this bit is not propagated when paths are copied between stores.
  - Content-addressable store paths no longer require signatures -- they can be imported into a store by unprivileged users even if they lack signatures.
  - The command **nix verify** checks whether the specified paths are trusted, i.e., have a certain number of trusted signatures, are ultimately trusted, or are content-addressed.

  - Substitutions from binary caches now require signatures by default. This was already the case on NixOS.
  - In Linux sandbox builds, we now use `/build` instead of `/tmp` as the temporary build directory. This fixes potential security problems when a build accidentally stores its `TMPDIR` in some security-sensitive place, such as an RPATH.

- *Pure evaluation mode*. This is a variant of the existing restricted evaluation mode. In pure mode, the Nix evaluator forbids access to anything that could cause different evaluations of the same command line arguments to produce a different result. This includes builtin functions such as `builtins.getEnv`, but more importantly, *all* filesystem or network access unless a content hash or commit hash is specified. For example, calls to `builtins.fetchGit` are only allowed if a `rev` attribute is specified.

  The goal of this feature is to enable true reproducibility and traceability of builds (including NixOS system configurations) at the evaluation level. For example, in the future, **nixos-rebuild** might build configurations from a Nix expression in a Git repository in pure mode. That expression might fetch other repositories such as Nixpkgs via `builtins.fetchGit`. The commit hash of the top-level repository then uniquely identifies a running system, and, in conjunction with that repository, allows it to be reproduced or modified.

- There are several new features to support binary reproducibility (i.e. to help ensure that multiple builds of the same derivation produce exactly the same output). When `enforce-determinism` is set to `false`, it's no longer a fatal error if build rounds produce different output. Also, a hook named `diff-hook` is provided to allow you to run tools such as **diffoscope** when build rounds produce different output.

- Configuring remote builds is a lot easier now. Provided you are not using the Nix daemon, you can now just specify a remote build machine on the command line, e.g. `--option builders 'ssh://my-mac ↪ x86_64-darwin'`. The environment variable `NIX_BUILD_HOOK` has been removed and is no longer needed. The environment variable `NIX_REMOTE_SYSTEMS` is still supported for compatibility, but it is also possible to specify builders in **nix.conf** by setting the option `builders = @path`.

- If a fixed-output derivation produces a result with an incorrect hash, the output path is moved to the location corresponding to the actual hash and registered as valid. Thus, a subsequent build of the fixed-output derivation with the correct hash is unnecessary.

- **nix-shell** now sets the `IN_NIX_SHELL` environment variable during evaluation and in the shell itself. This can be used to perform different actions depending on whether you're in a Nix shell or in a regular build. Nixpkgs provides `lib.inNixShell` to check this variable during evaluation.

- `NIX_PATH` is now lazy, so URIs in the path are only downloaded if they are needed for evaluation.

- You can now use `channel:channel-name` as a short-hand for `https://nixos.org/channels/channel-name`/nixexprs.tar.x For example, `nix-build channel:nixos-15.09 -A hello` will build the GNU Hello package from the `nixos-15.09` channel. In the future, this may use Git to fetch updates more efficiently.

- When `--no-build-output` is given, the last 10 lines of the build log will be shown if a build fails.

- Networking has been improved:
  - HTTP/2 is now supported. This makes binary cache lookups much more efficient.
  - We now retry downloads on many HTTP errors, making binary caches substituters more resilient to temporary failures.
  - HTTP credentials can now be configured via the standard `netrc` mechanism.
  - If S3 support is enabled at compile time, `s3://` URIs are supported in all places where Nix allows URIs.
  - Brotli compression is now supported. In particular, `cache.nixos.org` build logs are now compressed using Brotli.

- **nix-env** now ignores packages with bad derivation names (in particular those starting with a digit or containing a dot).

- Many configuration options have been renamed, either because they were unnecessarily verbose (e.g. `build-use-sandbox` is now just `sandbox`) or to reflect generalised behaviour (e.g. `binary-caches` is now `substituters` because it allows arbitrary store URIs). The old names are still supported for compatibility.

- The `max-jobs` option can now be set to `auto` to use the number of CPUs in the system.
- Hashes can now be specified in base-64 format, in addition to base-16 and the non-standard base-32.
- **nix-shell** now uses `bashInteractive` from Nixpkgs, rather than the **bash** command that happens to be in the caller's `PATH`. This is especially important on macOS where the **bash** provided by the system is seriously outdated and cannot execute `stdenv`'s setup script.
- Nix can now automatically trigger a garbage collection if free disk space drops below a certain level during a build. This is configured using the `min-free` and `max-free` options.
- **nix-store -q --roots** and **nix-store --gc --print-roots** now show temporary and in-memory roots.
- Nix can now be extended with plugins. See the documentation of the `plugin-files` option for more details.

The Nix language has the following new features:

- It supports floating point numbers. They are based on the C++ `float` type and are supported by the existing numerical operators. Export and import to and from JSON and XML works, too.
- Derivation attributes can now reference the outputs of the derivation using the `placeholder` builtin function. For example, the attribute

  ```
  configureFlags = "--prefix=${placeholder "out"} --includedir=${placeholder "dev"}";
  ```

  will cause the `configureFlags` environment variable to contain the actual store paths corresponding to the `out` and `dev` outputs.

The following builtin functions are new or extended:

- `builtins.fetchGit` allows Git repositories to be fetched at evaluation time. Thus it differs from the `fetchgit` function in Nixpkgs, which fetches at build time and cannot be used to fetch Nix expressions during evaluation. A typical use case is to import external NixOS modules from your configuration, e.g.

  ```
  imports = [ (builtins.fetchGit https://github.com/edolstra/dwarffs +
     ↪ "/module.nix") ];
  ```

- Similarly, `builtins.fetchMercurial` allows you to fetch Mercurial repositories.
- `builtins.path` generalises `builtins.filterSource` and path literals (e.g. `./foo`). It allows specifying a store path name that differs from the source path name (e.g. `builtins.path { path = ./foo; name = ↪ "bar"; }`) and also supports filtering out unwanted files.
- `builtins.fetchurl` and `builtins.fetchTarball` now support `sha256` and `name` attributes.
- `builtins.split` splits a string using a POSIX extended regular expression as the separator.
- `builtins.partition` partitions the elements of a list into two lists, depending on a Boolean predicate.
- `<nix/fetchurl.nix>` now uses the content-addressable tarball cache at `http://tarballs.nixos.org/`, just like `fetchurl` in Nixpkgs. (f2682e6e18a76ecbfb8a12c17e3a0ca15c084197)
- In restricted and pure evaluation mode, builtin functions that download from the network (such as `fetchGit`) are permitted to fetch underneath a list of URI prefixes specified in the option `allowed-uris`.

The Nix build environment has the following changes:

- Values such as Booleans, integers, (nested) lists and attribute sets can now be passed to builders in a non-lossy way. If the special attribute `__structuredAttrs` is set to `true`, the other derivation attributes are serialised in JSON format and made available to the builder via the file `.attrs.json` in the builder's temporary directory. This obviates the need for `passAsFile` since JSON files have no size restrictions, unlike process environments.

  As a convenience to Bash builders, Nix writes a script named `.attrs.sh` to the builder's directory that initialises shell variables corresponding to all attributes that are representable in Bash. This includes non-nested (associative) arrays. For example, the attribute `hardening.format = true` ends up as the Bash associative array element `${hardening[format]}`.

- Builders can now communicate what build phase they are in by writing messages to the file descriptor specified in `NIX_LOG_FD`. The current phase is shown by the **nix** progress indicator.
- In Linux sandbox builds, we now provide a default `/bin/sh` (namely `ash` from BusyBox).
- In structured attribute mode, `exportReferencesGraph` exports extended information about closures in JSON format. In particular, it includes the sizes and hashes of paths. This is primarily useful for NixOS image builders.
- Builds are now killed as soon as Nix receives EOF on the builder's stdout or stderr. This fixes a bug that allowed builds to hang Nix indefinitely, regardless of timeouts.
- The `sandbox-paths` configuration option can now specify optional paths by appending a `?`, e.g. `/dev/nvidiactl?` will bind-mount `/dev/nvidiactl` only if it exists.
- On Linux, builds are now executed in a user namespace with UID 1000 and GID 100.

A number of significant internal changes were made:

- Nix no longer depends on Perl and all Perl components have been rewritten in C++ or removed. The Perl bindings that used to be part of Nix have been moved to a separate package, `nix-perl`.
- All `Store` classes are now thread-safe. `RemoteStore` supports multiple concurrent connections to the daemon. This is primarily useful in multi-threaded programs such as **hydra-queue-runner**.

This release has contributions from Adrien Devresse, Alexander Ried, Alex Cruice, Alexey Shmalko, AmineChikhaoui, Andy Wingo, Aneesh Agrawal, Anthony Cowley, Armijn Hemel, aszlig, Ben Gamari, Benjamin Hipple, Benjamin Staffin, Benno Fünfstück, Bjørn Forsman, Brian McKenna, Charles Strahan, Chase Adams, Chris Martin, Christian Theune, Chris Warburton, Daiderd Jordan, Dan Connolly, Daniel Peebles, Dan Peebles, davidak, David McFarland, Dmitry Kalinkin, Domen Kožar, Eelco Dolstra, Emery Hemingway, Eric Litak, Eric Wolf, Fabian Schmitthenner, Frederik Rietdijk, Gabriel Gonzalez, Giorgio Gallo, Graham Christensen, Guillaume Maudoux, Harmen, Iavael, James Broadhead, James Earl Douglas, Janus Troelsen, Jeremy Shaw, Joachim Schiele, Joe Hermaszewski, Joel Moberg, Johannes 'fish' Ziemke, Jörg Thalheim, Jude Taylor, kballou, Keshav Kini, Kjetil Orbekk, Langston Barrett, Linus Heckemann, Ludovic Courtès, Manav Rathi, Marc Scholten, Markus Hauck, Matt Audesse, Matthew Bauer, Matthias Beyer, Matthieu Coudron, N1X, Nathan Zadoks, Neil Mayhew, Nicolas B. Pierron, Niklas Hambüchen, Nikolay Amiantov, Ole Jørgen Brønner, Orivej Desh, Peter Simons, Peter Stuart, Pyry Jahkola, regnat, Renzo Carbonara, Rhys, Robert Vollmert, Scott Olson, Scott R. Parish, Sergei Trofimovich, Shea Levy, Sheena Artrip, Spencer Baugh, Stefan Junker, Susan Potter, Thomas Tuegel, Timothy Allen, Tristan Hume, Tuomas Tynkkynen, tv, Tyson Whitehead, Vladimír Čunát, Will Dietz, wmertens, Wout Mertens, zimbatm and Zoran Plesivčak.

C.5. Release 1.11.10 (2017-06-12)

This release fixes a security bug in Nix's "build user" build isolation mechanism. Previously, Nix builders had the ability to create setuid binaries owned by a `nixbld` user. Such a binary could then be used by an attacker to assume a `nixbld` identity and interfere with subsequent builds running under the same UID.

To prevent this issue, Nix now disallows builders to create setuid and setgid binaries. On Linux, this is done using a seccomp BPF filter. Note that this imposes a small performance penalty (e.g. 1% when building GNU Hello). Using seccomp, we now also prevent the creation of extended attributes and POSIX ACLs since these cannot be represented in the NAR format and (in the case of POSIX ACLs) allow bypassing regular Nix store permissions. On macOS, the restriction is implemented using the existing sandbox mechanism, which now uses a minimal "allow all except the creation of setuid/setgid binaries" profile when regular sandboxing is disabled. On other platforms, the "build user" mechanism is now disabled.

Thanks go to Linus Heckemann for discovering and reporting this bug.

C.6. Release 1.11 (2016-01-19)

This is primarily a bug fix release. It also has a number of new features:

- **nix-prefetch-url** can now download URLs specified in a Nix expression. For example,

```
$ nix-prefetch-url -A hello.src
```

will prefetch the file specified by the `fetchurl` call in the attribute `hello.src` from the Nix expression in the current directory, and print the cryptographic hash of the resulting file on stdout. This differs from `nix-build -A hello.src` in that it doesn't verify the hash, and is thus useful when you're updating a Nix expression.

You can also prefetch the result of functions that unpack a tarball, such as `fetchFromGitHub`. For example:

```
$ nix-prefetch-url --unpack https://github.com/NixOS/patchelf/archive/0.8.tar.gz
```

or from a Nix expression:

```
$ nix-prefetch-url -A nix-repl.src
```

- The builtin function `<nix/fetchurl.nix>` now supports downloading and unpacking NARs. This removes the need to have multiple downloads in the Nixpkgs stdenv bootstrap process (like a separate busybox binary for Linux, or curl/mkdir/sh/bzip2 for Darwin). Now all those files can be combined into a single NAR, optionally compressed using **xz**.
- Nix now supports SHA-512 hashes for verifying fixed-output derivations, and in `builtins.hashString`.
- The new flag `--option build-repeat N` will cause every build to be executed N+1 times. If the build output differs between any round, the build is rejected, and the output paths are not registered as valid. This is primarily useful to verify build determinism. (We already had a `--check` option to repeat a previously succeeded build. However, with `--check`, non-deterministic builds are registered in the DB. Preventing that is useful for Hydra to ensure that non-deterministic builds don't end up getting published to the binary cache.)
- The options `--check` and `--option build-repeat N`, if they detect a difference between two runs of the same derivation and `-K` is given, will make the output of the other run available under `store-path`-check. This makes it easier to investigate the non-determinism using tools like **diffoscope**, e.g.,

```
$ nix-build pkgs/stdenv/linux -A stage1.pkgs.zlib --check -K
error: derivation ‘/nix/store/l54i8wlw2265...-zlib-1.2.8.drv' may not
be deterministic: output ‘/nix/store/11a27shh6n2i...-zlib-1.2.8'
differs from ‘/nix/store/11a27shh6n2i...-zlib-1.2.8-check'

$ diffoscope /nix/store/11a27shh6n2i...-zlib-1.2.8
   ↪ /nix/store/11a27shh6n2i...-zlib-1.2.8-check
...
 lib/libz.a
    metadata
    @@ -1,15 +1,15 @@
    -rw-r--r-- 30001/30000   3096 Jan 12 15:20 2016 adler32.o
...
    +rw-r--r-- 30001/30000   3096 Jan 12 15:28 2016 adler32.o
...
```

- Improved FreeBSD support.
- **nix-env -qa --xml --meta** now prints license information.
- The maximum number of parallel TCP connections that the binary cache substituter will use has been decreased from 150 to 25. This should prevent upsetting some broken NAT routers, and also improves performance.
- All "chroot"-containing strings got renamed to "sandbox". In particular, some Nix options got renamed, but the old names are still accepted as lower-priority aliases.

This release has contributions from Anders Claesson, Anthony Cowley, Bjørn Forsman, Brian McKenna, Danny Wilson, davidak, Eelco Dolstra, Fabian Schmitthenner, FrankHB, Ilya Novoselov, janus, Jim Garrison, John Ericson, Jude Taylor, Ludovic Courtès, Manuel Jacob, Mathnerd314, Pascal Wittmann, Peter Simons, Philip Potter, Preston

Bennes, Rommel M. Martinez, Sander van der Burg, Shea Levy, Tim Cuthbertson, Tuomas Tynkkynen, Utku Demir and Vladimír Čunát.

C.7. Release 1.10 (2015-09-03)

This is primarily a bug fix release. It also has a number of new features:

- A number of builtin functions have been added to reduce Nixpkgs/NixOS evaluation time and memory consumption: `all`, `any`, `concatStringsSep`, `foldl'`, `genList`, `replaceStrings`, `sort`.
- The garbage collector is more robust when the disk is full.
- Nix supports a new API for building derivations that doesn't require a `.drv` file to be present on disk; it only requires an in-memory representation of the derivation. This is used by the Hydra continuous build system to make remote builds more efficient.
- The function `<nix/fetchurl.nix>` now uses a *builtin* builder (i.e. it doesn't require starting an external process; the download is performed by Nix itself). This ensures that derivation paths don't change when Nix is upgraded, and obviates the need for ugly hacks to support chroot execution.
- `--version -v` now prints some configuration information, in particular what compile-time optional features are enabled, and the paths of various directories.
- Build users have their supplementary groups set correctly.

This release has contributions from Eelco Dolstra, Guillaume Maudoux, Iwan Aucamp, Jaka Hudoklin, Kirill Elagin, Ludovic Courtès, Manolis Ragkousis, Nicolas B. Pierron and Shea Levy.

C.8. Release 1.9 (2015-06-12)

In addition to the usual bug fixes, this release has the following new features:

- Signed binary cache support. You can enable signature checking by adding the following to `nix.conf`:

  ```
  signed-binary-caches = *
  binary-cache-public-keys =
      ↪ cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY=
  ```

  This will prevent Nix from downloading any binary from the cache that is not signed by one of the keys listed in `binary-cache-public-keys`.

  Signature checking is only supported if you built Nix with the `libsodium` package.

  Note that while Nix has had experimental support for signed binary caches since version 1.7, this release changes the signature format in a backwards-incompatible way.

- Automatic downloading of Nix expression tarballs. In various places, you can now specify the URL of a tarball containing Nix expressions (such as Nixpkgs), which will be downloaded and unpacked automatically. For example:

  - In **nix-env**:

    ```
    $ nix-env -f https://github.com/NixOS/nixpkgs-channels/archive/nixos-14.12.tar.gz
        ↪ -iA firefox
    ```

    This installs Firefox from the latest tested and built revision of the NixOS 14.12 channel.

  - In **nix-build** and **nix-shell**:

    ```
    $ nix-build https://github.com/NixOS/nixpkgs/archive/master.tar.gz -A hello
    ```

    This builds GNU Hello from the latest revision of the Nixpkgs master branch.

  - In the Nix search path (as specified via `NIX_PATH` or `-I`). For example, to start a shell containing the Pan package from a specific version of Nixpkgs:

```
$ nix-shell -p pan -I
   ↪ nixpkgs=https://github.com/NixOS/nixpkgs-channels/archive/8a3eea054838b55aca962
```

– In **nixos-rebuild** (on NixOS):

```
$ nixos-rebuild test -I
   ↪ nixpkgs=https://github.com/NixOS/nixpkgs-channels/archive/nixos-unstable.tar.gz
```

– In Nix expressions, via the new builtin function `fetchTarball`:

```
with import (fetchTarball
   ↪ https://github.com/NixOS/nixpkgs-channels/archive/nixos-14.12.tar.gz) {}; ...
```

(This is not allowed in restricted mode.)

- **nix-shell** improvements:
  – **nix-shell** now has a flag `--run` to execute a command in the **nix-shell** environment, e.g. `nix-shell ↪ --run make`. This is like the existing `--command` flag, except that it uses a non-interactive shell (ensuring that hitting Ctrl-C won't drop you into the child shell).
  – **nix-shell** can now be used as a `#!`-interpreter. This allows you to write scripts that dynamically fetch their own dependencies. For example, here is a Haskell script that, when invoked, first downloads GHC and the Haskell packages on which it depends:

```
#! /usr/bin/env nix-shell
#! nix-shell -i runghc -p haskellPackages.ghc haskellPackages.HTTP

import Network.HTTP

main = do
  resp <- Network.HTTP.simpleHTTP (getRequest "http://nixos.org/")
  body <- getResponseBody resp
  print (take 100 body)
```

Of course, the dependencies are cached in the Nix store, so the second invocation of this script will be much faster.

- Chroot improvements:
  – Chroot builds are now supported on Mac OS X (using its sandbox mechanism).
  – If chroots are enabled, they are now used for all derivations, including fixed-output derivations (such as `fetchurl`). The latter do have network access, but can no longer access the host filesystem. If you need the old behaviour, you can set the option `build-use-chroot` to `relaxed`.
  – On Linux, if chroots are enabled, builds are performed in a private PID namespace once again. (This functionality was lost in Nix 1.8.)
  – Store paths listed in `build-chroot-dirs` are now automatically expanded to their closure. For instance, if you want `/nix/store/...-bash/bin/sh` mounted in your chroot as `/bin/sh`, you only need to say `build-chroot-dirs = /bin/sh=/nix/store/...-bash/bin/sh`; it is no longer necessary to specify the dependencies of Bash.
- The new derivation attribute `passAsFile` allows you to specify that the contents of derivation attributes should be passed via files rather than environment variables. This is useful if you need to pass very long strings that exceed the size limit of the environment. The Nixpkgs function `writeTextFile` uses this.
- You can now use `~` in Nix file names to refer to your home directory, e.g. `import ~/.nixpkgs/config.nix`.
- Nix has a new option `restrict-eval` that allows limiting what paths the Nix evaluator has access to. By passing `--option restrict-eval true` to Nix, the evaluator will throw an exception if an attempt is made to access any file outside of the Nix search path. This is primarily intended for Hydra to ensure that a Hydra jobset only refers to its declared inputs (and is therefore reproducible).

- **nix-env** now only creates a new "generation" symlink in `/nix/var/nix/profiles` if something actually changed.
- The environment variable `NIX_PAGER` can now be set to override `PAGER`. You can set it to `cat` to disable paging for Nix commands only.
- Failing `<...>` lookups now show position information.
- Improved Boehm GC use: we disabled scanning for interior pointers, which should reduce the "`Repeated ↪ allocation of very large block`" warnings and associated retention of memory.

This release has contributions from aszlig, Benjamin Staffin, Charles Strahan, Christian Theune, Daniel Hahler, Danylo Hlynskyi Daniel Peebles, Dan Peebles, Domen Kožar, Eelco Dolstra, Harald van Dijk, Hoang Xuan Phu, Jaka Hudoklin, Jeff Ramnani, j-keck, Linquize, Luca Bruno, Michael Merickel, Oliver Dunkl, Rob Vermaas, Rok Garbas, Shea Levy, Tobias Geerinckx-Rice and William A. Kennington III.

C.9. Release 1.8 (2014-12-14)

- Breaking change: to address a race condition, the remote build hook mechanism now uses **nix-store --serve** on the remote machine. This requires build slaves to be updated to Nix 1.8.
- Nix now uses HTTPS instead of HTTP to access the default binary cache, `cache.nixos.org`.
- **nix-env** selectors are now regular expressions. For instance, you can do

  ```
  $ nix-env -qa '.*zip.*'
  ```

  to query all packages with a name containing `zip`.
- **nix-store --read-log** can now fetch remote build logs. If a build log is not available locally, then 'nix-store -l' will now try to download it from the servers listed in the 'log-servers' option in nix.conf. For instance, if you have the configuration option

  ```
  log-servers = http://hydra.nixos.org/log
  ```

  then it will try to get logs from `http://hydra.nixos.org/log/base name of the store path`. This allows you to do things like:

  ```
  $ nix-store -l $(which xterm)
  ```

  and get a log even if **xterm** wasn't built locally.
- New builtin functions: `attrValues`, `deepSeq`, `fromJSON`, `readDir`, `seq`.
- **nix-instantiate --eval** now has a `--json` flag to print the resulting value in JSON format.
- **nix-copy-closure** now uses **nix-store --serve** on the remote side to send or receive closures. This fixes a race condition between **nix-copy-closure** and the garbage collector.
- Derivations can specify the new special attribute `allowedRequisites`, which has a similar meaning to `allowedReferences`. But instead of only enforcing to explicitly specify the immediate references, it requires the derivation to specify all the dependencies recursively (hence the name, requisites) that are used by the resulting output.
- On Mac OS X, Nix now handles case collisions when importing closures from case-sensitive file systems. This is mostly useful for running NixOps on Mac OS X.
- The Nix daemon has new configuration options `allowed-users` (specifying the users and groups that are allowed to connect to the daemon) and `trusted-users` (specifying the users and groups that can perform privileged operations like specifying untrusted binary caches).
- The configuration option `build-cores` now defaults to the number of available CPU cores.
- Build users are now used by default when Nix is invoked as root. This prevents builds from accidentally running as root.

- Nix now includes systemd units and Upstart jobs.
- Speed improvements to **nix-store --optimise**.
- Language change: the `==` operator now ignores string contexts (the "dependencies" of a string).
- Nix now filters out Nix-specific ANSI escape sequences on standard error. They are supposed to be invisible, but some terminals show them anyway.
- Various commands now automatically pipe their output into the pager as specified by the `PAGER` environment variable.
- Several improvements to reduce memory consumption in the evaluator.

This release has contributions from Adam Szkoda, Aristid Breitkreuz, Bob van der Linden, Charles Strahan, darealshinji, Eelco Dolstra, Gergely Risko, Joel Taylor, Ludovic Courtès, Marko Durkovic, Mikey Ariel, Paul Colomiets, Ricardo M. Correia, Ricky Elrod, Robert Helgesson, Rob Vermaas, Russell O'Connor, Shea Levy, Shell Turner, Sönke Hahn, Steve Purcell, Vladimír Čunát and Wout Mertens.

C.10. Release 1.7 (2014-04-11)

In addition to the usual bug fixes, this release has the following new features:

- Antiquotation is now allowed inside of quoted attribute names (e.g. `set."${foo}"`). In the case where the attribute name is just a single antiquotation, the quotes can be dropped (e.g. the above example can be written `set.${foo}`). If an attribute name inside of a set declaration evaluates to `null` (e.g. `{ ${null} = false; }`), then that attribute is not added to the set.
- Experimental support for cryptographically signed binary caches. See the commit for details.
- An experimental new substituter, **download-via-ssh**, that fetches binaries from remote machines via SSH. Specifying the flags `--option use-ssh-substituter true --option ssh-substituter-hosts` ↪ `user@hostname` will cause Nix to download binaries from the specified machine, if it has them.
- **nix-store -r** and **nix-build** have a new flag, `--check`, that builds a previously built derivation again, and prints an error message if the output is not exactly the same. This helps to verify whether a derivation is truly deterministic. For example:

```
$ nix-build '<nixpkgs>' -A patchelf
...
$ nix-build '<nixpkgs>' -A patchelf --check
...
error: derivation `/nix/store/1ipvxs...-patchelf-0.6' may not be deterministic:
  hash mismatch in output `/nix/store/4pc1dm...-patchelf-0.6.drv'
```

- The **nix-instantiate** flags `--eval-only` and `--parse-only` have been renamed to `--eval` and `--parse`, respectively.
- **nix-instantiate**, **nix-build** and **nix-shell** now have a flag `--expr` (or `-E`) that allows you to specify the expression to be evaluated as a command line argument. For instance, `nix-instantiate --eval -E '1 +` ↪ `2'` will print `3`.
- **nix-shell** improvements:
  - It has a new flag, `--packages` (or `-p`), that sets up a build environment containing the specified packages from Nixpkgs. For example, the command

    ```
    $ nix-shell -p sqlite xorg.libX11 hello
    ```

    will start a shell in which the given packages are present.
  - It now uses `shell.nix` as the default expression, falling back to `default.nix` if the former doesn't exist. This makes it convenient to have a `shell.nix` in your project to set up a nice development environment.

  - It evaluates the derivation attribute `shellHook`, if set. Since `stdenv` does not normally execute this hook, it allows you to do **nix-shell**-specific setup.
  - It preserves the user's timezone setting.

- In chroots, Nix now sets up a `/dev` containing only a minimal set of devices (such as `/dev/null`). Note that it only does this if you *don't* have `/dev` listed in your `build-chroot-dirs` setting; otherwise, it will bind-mount the `/dev` from outside the chroot.

  Similarly, if you don't have `/dev/pts` listed in `build-chroot-dirs`, Nix will mount a private `devpts` filesystem on the chroot's `/dev/pts`.

- New built-in function: `builtins.toJSON`, which returns a JSON representation of a value.
- **nix-env -q** has a new flag `--json` to print a JSON representation of the installed or available packages.
- **nix-env** now supports meta attributes with more complex values, such as attribute sets.
- The `-A` flag now allows attribute names with dots in them, e.g.

```
$ nix-instantiate --eval '<nixos>' -A 'config.systemd.units."nscd.service".text'
```

- The `--max-freed` option to **nix-store --gc** now accepts a unit specifier. For example, `nix-store --gc ↪ --max-freed 1G` will free up to 1 gigabyte of disk space.
- **nix-collect-garbage** has a new flag `--delete-older-than` *N*`d`, which deletes all user environment generations older than *N* days. Likewise, **nix-env --delete-generations** accepts a *N*`d` age limit.
- Nix now heuristically detects whether a build failure was due to a disk-full condition. In that case, the build is not flagged as "permanently failed". This is mostly useful for Hydra, which needs to distinguish between permanent and transient build failures.
- There is a new symbol `__curPos` that expands to an attribute set containing its file name and line and column numbers, e.g. `{ file = "foo.nix"; line = 10; column = 5; }`. There also is a new builtin function, `unsafeGetAttrPos`, that returns the position of an attribute. This is used by Nixpkgs to provide location information in error messages, e.g.

```
$ nix-build '<nixpkgs>' -A libreoffice --argstr system x86_64-darwin
error: the package ‘libreoffice-4.0.5.2' in
   ↪ ‘.../applications/office/libreoffice/default.nix:263'
  is not supported on ‘x86_64-darwin'
```

- The garbage collector is now more concurrent with other Nix processes because it releases certain locks earlier.
- The binary tarball installer has been improved. You can now install Nix by running:

```
$ bash <(curl https://nixos.org/nix/install)
```

- More evaluation errors include position information. For instance, selecting a missing attribute will print something like

```
error: attribute `nixUnstabl' missing, at
   ↪ /etc/nixos/configurations/misc/eelco/mandark.nix:216:15
```

- The command **nix-setuid-helper** is gone.
- Nix no longer uses Automake, but instead has a non-recursive, GNU Make-based build system.
- All installed libraries now have the prefix `libnix`. In particular, this gets rid of `libutil`, which could clash with libraries with the same name from other packages.
- Nix now requires a compiler that supports C++11.

This release has contributions from Danny Wilson, Domen Kožar, Eelco Dolstra, Ian-Woo Kim, Ludovic Courtès, Maxim Ivanov, Petr Rockai, Ricardo M. Correia and Shea Levy.

C.11. Release 1.6.1 (2013-10-28)

This is primarily a bug fix release. Changes of interest are:

- Nix 1.6 accidentally changed the semantics of antiquoted paths in strings, such as `"${/foo}/bar"`. This release reverts to the Nix 1.5.3 behaviour.
- Previously, Nix optimised expressions such as "`${expr}`" to *`expr`*. Thus it neither checked whether *`expr`* could be coerced to a string, nor applied such coercions. This meant that `"${123}"` evaluatued to `123`, and `"${./foo}"` evaluated to `./foo` (even though `"${./foo} "` evaluates to `"/nix/store/hash-foo "`). Nix now checks the type of antiquoted expressions and applies coercions.
- Nix now shows the exact position of undefined variables. In particular, undefined variable errors in a `with` previously didn't show *any* position information, so this makes it a lot easier to fix such errors.
- Undefined variables are now treated consistently. Previously, the `tryEval` function would catch undefined variables inside a `with` but not outside. Now `tryEval` never catches undefined variables.
- Bash completion in **nix-shell** now works correctly.
- Stack traces are less verbose: they no longer show calls to builtin functions and only show a single line for each derivation on the call stack.
- New built-in function: `builtins.typeOf`, which returns the type of its argument as a string.

C.12. Release 1.6 (2013-09-10)

In addition to the usual bug fixes, this release has several new features:

- The command **nix-build --run-env** has been renamed to **nix-shell**.
- **nix-shell** now sources `$stdenv/setup` *inside* the interactive shell, rather than in a parent shell. This ensures that shell functions defined by `stdenv` can be used in the interactive shell.
- **nix-shell** has a new flag `--pure` to clear the environment, so you get an environment that more closely corresponds to the "real" Nix build.
- **nix-shell** now sets the shell prompt (`PS1`) to ensure that Nix shells are distinguishable from your regular shells.
- **nix-env** no longer requires a `*` argument to match all packages, so `nix-env -qa` is equivalent to `nix-env -qa '*'`.
- **nix-env -i** has a new flag `--remove-all` (`-r`) to remove all previous packages from the profile. This makes it easier to do declarative package management similar to NixOS's `environment.systemPackages`. For instance, if you have a specification `my-packages.nix` like this:

```
with import <nixpkgs> {};
[ thunderbird
  geeqie
  ...
]
```

  then after any change to this file, you can run:

```
$ nix-env -f my-packages.nix -ir
```

  to update your profile to match the specification.

- The '`with`' language construct is now more lazy. It only evaluates its argument if a variable might actually refer to an attribute in the argument. For instance, this now works:

```
let
  pkgs = with pkgs; { foo = "old"; bar = foo; } // overrides;
  overrides = { foo = "new"; };
in pkgs.bar
```

This evaluates to `"new"`, while previously it gave an "infinite recursion" error.

- Nix now has proper integer arithmetic operators. For instance, you can write `x + y` instead of `builtins.add x y`, or `x < y` instead of `builtins.lessThan x y`. The comparison operators also work on strings.
- On 64-bit systems, Nix integers are now 64 bits rather than 32 bits.
- When using the Nix daemon, the **nix-daemon** worker process now runs on the same CPU as the client, on systems that support setting CPU affinity. This gives a significant speedup on some systems.
- If a stack overflow occurs in the Nix evaluator, you now get a proper error message (rather than "Segmentation fault") on some systems.
- In addition to directories, you can now bind-mount regular files in chroots through the (now misnamed) option `build-chroot-dirs`.

This release has contributions from Domen Kožar, Eelco Dolstra, Florian Friesdorf, Gergely Risko, Ivan Kozik, Ludovic Courtès and Shea Levy.

C.13. Release 1.5.2 (2013-05-13)

This is primarily a bug fix release. It has contributions from Eelco Dolstra, Lluís Batlle i Rossell and Shea Levy.

C.14. Release 1.5 (2013-02-27)

This is a brown paper bag release to fix a regression introduced by the hard link security fix in 1.4.

C.15. Release 1.4 (2013-02-26)

This release fixes a security bug in multi-user operation. It was possible for derivations to cause the mode of files outside of the Nix store to be changed to 444 (read-only but world-readable) by creating hard links to those files (details).

There are also the following improvements:

- New built-in function: `builtins.hashString`.
- Build logs are now stored in `/nix/var/log/nix/drvs/XX/`, where *XX* is the first two characters of the derivation. This is useful on machines that keep a lot of build logs (such as Hydra servers).
- The function `corepkgs/fetchurl` can now make the downloaded file executable. This will allow getting rid of all bootstrap binaries in the Nixpkgs source tree.
- Language change: The expression `"${./path} ..."` now evaluates to a string instead of a path.

C.16. Release 1.3 (2013-01-04)

This is primarily a bug fix release. When this version is first run on Linux, it removes any immutable bits from the Nix store and increases the schema version of the Nix store. (The previous release removed support for setting the immutable bit; this release clears any remaining immutable bits to make certain operations more efficient.)

This release has contributions from Eelco Dolstra and Stuart Pernsteiner.

C.17. Release 1.2 (2012-12-06)

This release has the following improvements and changes:

- Nix has a new binary substituter mechanism: the *binary cache*. A binary cache contains pre-built binaries of Nix packages. Whenever Nix wants to build a missing Nix store path, it will check a set of binary caches to see if any of them has a pre-built binary of that path. The configuration setting `binary-caches` contains a list of URLs of binary caches. For instance, doing

  ```
  $ nix-env -i thunderbird --option binary-caches http://cache.nixos.org
  ```

will install Thunderbird and its dependencies, using the available pre-built binaries in `http://cache.nixos.org`. The main advantage over the old "manifest"-based method of getting pre-built binaries is that you don't have to worry about your manifest being in sync with the Nix expressions you're installing from; i.e., you don't need to run **nix-pull** to update your manifest. It's also more scalable because you don't need to redownload a giant manifest file every time.

A Nix channel can provide a binary cache URL that will be used automatically if you subscribe to that channel. If you use the Nixpkgs or NixOS channels (`http://nixos.org/channels`) you automatically get the cache `http://cache.nixos.org`.

Binary caches are created using **nix-push**. For details on the operation and format of binary caches, see the **nix-push** manpage. More details are provided in this nix-dev posting.

- Multiple output support should now be usable. A derivation can declare that it wants to produce multiple store paths by saying something like

```
outputs = [ "lib" "headers" "doc" ];
```

This will cause Nix to pass the intended store path of each output to the builder through the environment variables `lib`, `headers` and `doc`. Other packages can refer to a specific output by referring to `pkg.`*`output`*, e.g.

```
buildInputs = [ pkg.lib pkg.headers ];
```

If you install a package with multiple outputs using **nix-env**, each output path will be symlinked into the user environment.

- Dashes are now valid as part of identifiers and attribute names.
- The new operation **nix-store --repair-path** allows corrupted or missing store paths to be repaired by redownloading them. **nix-store --verify --check-contents --repair** will scan and repair all paths in the Nix store. Similarly, **nix-env**, **nix-build**, **nix-instantiate** and **nix-store --realise** have a `--repair` flag to detect and fix bad paths by rebuilding or redownloading them.
- Nix no longer sets the immutable bit on files in the Nix store. Instead, the recommended way to guard the Nix store against accidental modification on Linux is to make it a read-only bind mount, like this:

```
$ mount --bind /nix/store /nix/store
$ mount -o remount,ro,bind /nix/store
```

Nix will automatically make `/nix/store` writable as needed (using a private mount namespace) to allow modifications.

- Store optimisation (replacing identical files in the store with hard links) can now be done automatically every time a path is added to the store. This is enabled by setting the configuration option `auto-optimise-store` to `true` (disabled by default).
- Nix now supports **xz** compression for NARs in addition to **bzip2**. It compresses about 30% better on typical archives and decompresses about twice as fast.
- Basic Nix expression evaluation profiling: setting the environment variable `NIX_COUNT_CALLS` to `1` will cause Nix to print how many times each primop or function was executed.
- New primops: `concatLists`, `elem`, `elemAt` and `filter`.
- The command **nix-copy-closure** has a new flag `--use-substitutes` (`-s`) to download missing paths on the target machine using the substitute mechanism.
- The command **nix-worker** has been renamed to **nix-daemon**. Support for running the Nix worker in "slave" mode has been removed.
- The `--help` flag of every Nix command now invokes **man**.
- Chroot builds are now supported on systemd machines.

This release has contributions from Eelco Dolstra, Florian Friesdorf, Mats Erik Andersson and Shea Levy.

C.18. Release 1.1 (2012-07-18)

This release has the following improvements:

- On Linux, when doing a chroot build, Nix now uses various namespace features provided by the Linux kernel to improve build isolation. Namely:
  - The private network namespace ensures that builders cannot talk to the outside world (or vice versa): each build only sees a private loopback interface. This also means that two concurrent builds can listen on the same port (e.g. as part of a test) without conflicting with each other.
  - The PID namespace causes each build to start as PID 1. Processes outside of the chroot are not visible to those on the inside. On the other hand, processes inside the chroot *are* visible from the outside (though with different PIDs).
  - The IPC namespace prevents the builder from communicating with outside processes using SysV IPC mechanisms (shared memory, message queues, semaphores). It also ensures that all IPC objects are destroyed when the builder exits.
  - The UTS namespace ensures that builders see a hostname of `localhost` rather than the actual hostname.
  - The private mount namespace was already used by Nix to ensure that the bind-mounts used to set up the chroot are cleaned up automatically.
- Build logs are now compressed using **bzip2**. The command **nix-store -l** decompresses them on the fly. This can be disabled by setting the option `build-compress-log` to `false`.
- The creation of build logs in `/nix/var/log/nix/drvs` can be disabled by setting the new option `build-keep-log` to `false`. This is useful, for instance, for Hydra build machines.
- Nix now reserves some space in `/nix/var/nix/db/reserved` to ensure that the garbage collector can run successfully if the disk is full. This is necessary because SQLite transactions fail if the disk is full.
- Added a basic `fetchurl` function. This is not intended to replace the `fetchurl` in Nixpkgs, but is useful for bootstrapping; e.g., it will allow us to get rid of the bootstrap binaries in the Nixpkgs source tree and download them instead. You can use it by doing `import <nix/fetchurl.nix> { url = url;` sha256 = "*hash*"; }. (Shea Levy)
- Improved RPM spec file. (Michel Alexandre Salim)
- Support for on-demand socket-based activation in the Nix daemon with **systemd**.
- Added a manpage for nix.conf(5).
- When using the Nix daemon, the `-s` flag in **nix-env -qa** is now much faster.

C.19. Release 1.0 (2012-05-11)

There have been numerous improvements and bug fixes since the previous release. Here are the most significant:

- Nix can now optionally use the Boehm garbage collector. This significantly reduces the Nix evaluator's memory footprint, especially when evaluating large NixOS system configurations. It can be enabled using the `--enable-gc` configure option.
- Nix now uses SQLite for its database. This is faster and more flexible than the old *ad hoc* format. SQLite is also used to cache the manifests in `/nix/var/nix/manifests`, resulting in a significant speedup.
- Nix now has an search path for expressions. The search path is set using the environment variable `NIX_PATH` and the `-I` command line option. In Nix expressions, paths between angle brackets are used to specify files that must be looked up in the search path. For instance, the expression `<nixpkgs/default.nix>` looks for a file `nixpkgs/default.nix` relative to every element in the search path.
- The new command **nix-build --run-env** builds all dependencies of a derivation, then starts a shell in an environment containing all variables from the derivation. This is useful for reproducing the environment of a derivation for development.
- The new command **nix-store --verify-path** verifies that the contents of a store path have not changed.

- The new command **nix-store --print-env** prints out the environment of a derivation in a format that can be evaluated by a shell.
- Attribute names can now be arbitrary strings. For instance, you can write `{ "foo-1.2" = ...; "bla bla" = ...; }."bla bla"`.
- Attribute selection can now provide a default value using the `or` operator. For instance, the expression `x.y.z or e` evaluates to the attribute `x.y.z` if it exists, and `e` otherwise.
- The right-hand side of the `?` operator can now be an attribute path, e.g., `attrs ? a.b.c`.
- On Linux, Nix will now make files in the Nix store immutable on filesystems that support it. This prevents accidental modification of files in the store by the root user.
- Nix has preliminary support for derivations with multiple outputs. This is useful because it allows parts of a package to be deployed and garbage-collected separately. For instance, development parts of a package such as header files or static libraries would typically not be part of the closure of an application, resulting in reduced disk usage and installation time.
- The Nix store garbage collector is faster and holds the global lock for a shorter amount of time.
- The option `--timeout` (corresponding to the configuration setting `build-timeout`) allows you to set an absolute timeout on builds -- if a build runs for more than the given number of seconds, it is terminated. This is useful for recovering automatically from builds that are stuck in an infinite loop but keep producing output, and for which `--max-silent-time` is ineffective.
- Nix development has moved to GitHub (https://github.com/NixOS/nix).

C.20. Release 0.16 (2010-08-17)

This release has the following improvements:

- The Nix expression evaluator is now much faster in most cases: typically, 3 to 8 times compared to the old implementation. It also uses less memory. It no longer depends on the ATerm library.
- Support for configurable parallelism inside builders. Build scripts have always had the ability to perform multiple build actions in parallel (for instance, by running **make -j 2**), but this was not desirable because the number of actions to be performed in parallel was not configurable. Nix now has an option `--cores N` as well as a configuration setting `build-cores = N` that causes the environment variable `NIX_BUILD_CORES` to be set to *N* when the builder is invoked. The builder can use this at its discretion to perform a parallel build, e.g., by calling **make -j *N***. In Nixpkgs, this can be enabled on a per-package basis by setting the derivation attribute `enableParallelBuilding` to `true`.
- **nix-store -q** now supports XML output through the `--xml` flag.
- Several bug fixes.

C.21. Release 0.15 (2010-03-17)

This is a bug-fix release. Among other things, it fixes building on Mac OS X (Snow Leopard), and improves the contents of `/etc/passwd` and `/etc/group` in `chroot` builds.

C.22. Release 0.14 (2010-02-04)

This release has the following improvements:

- The garbage collector now starts deleting garbage much faster than before. It no longer determines liveness of all paths in the store, but does so on demand.
- Added a new operation, **nix-store --query --roots**, that shows the garbage collector roots that directly or indirectly point to the given store paths.
- Removed support for converting Berkeley DB-based Nix databases to the new schema.
- Removed the `--use-atime` and `--max-atime` garbage collector options. They were not very useful in practice.

- On Windows, Nix now requires Cygwin 1.7.x.
- A few bug fixes.

C.23. Release 0.13 (2009-11-05)

This is primarily a bug fix release. It has some new features:

- Syntactic sugar for writing nested attribute sets. Instead of

```
{
  foo = {
    bar = 123;
    xyzzy = true;
  };
  a = { b = { c = "d"; }; };
}
```

  you can write

```
{
  foo.bar = 123;
  foo.xyzzy = true;
  a.b.c = "d";
}
```

  This is useful, for instance, in NixOS configuration files.

- Support for Nix channels generated by Hydra, the Nix-based continuous build system. (Hydra generates NAR archives on the fly, so the size and hash of these archives isn't known in advance.)
- Support `i686-linux` builds directly on `x86_64-linux` Nix installations. This is implemented using the `personality()` syscall, which causes **uname** to return `i686` in child processes.
- Various improvements to the `chroot` support. Building in a `chroot` works quite well now.
- Nix no longer blocks if it tries to build a path and another process is already building the same path. Instead it tries to build another buildable path first. This improves parallelism.
- Support for large (> 4 GiB) files in NAR archives.
- Various (performance) improvements to the remote build mechanism.
- New primops: `builtins.addErrorContext` (to add a string to stack traces -- useful for debugging), `builtins.isBool`, `builtins.isString`, `builtins.isInt`, `builtins.intersectAttrs`.
- OpenSolaris support (Sander van der Burg).
- Stack traces are no longer displayed unless the `--show-trace` option is used.
- The scoping rules for `inherit (e) ...` in recursive attribute sets have changed. The expression *`e`* can now refer to the attributes defined in the containing set.

C.24. Release 0.12 (2008-11-20)

- Nix no longer uses Berkeley DB to store Nix store metadata. The principal advantages of the new storage scheme are: it works properly over decent implementations of NFS (allowing Nix stores to be shared between multiple machines); no recovery is needed when a Nix process crashes; no write access is needed for read-only operations; no more running out of Berkeley DB locks on certain operations.

  You still need to compile Nix with Berkeley DB support if you want Nix to automatically convert your old Nix store to the new schema. If you don't need this, you can build Nix with the `configure` option `--disable-old-db-compat`.

  After the automatic conversion to the new schema, you can delete the old Berkeley DB files:

```
$ cd /nix/var/nix/db
$ rm __db* log.* derivers references referrers reserved validpaths DB_CONFIG
```

The new metadata is stored in the directories `/nix/var/nix/db/info` and `/nix/var/nix/db/referrer`. Though the metadata is stored in human-readable plain-text files, they are not intended to be human-editable, as Nix is rather strict about the format.

The new storage schema may or may not require less disk space than the Berkeley DB environment, mostly depending on the cluster size of your file system. With 1 KiB clusters (which seems to be the `ext3` default nowadays) it usually takes up much less space.

- There is a new substituter that copies paths directly from other (remote) Nix stores mounted somewhere in the filesystem. For instance, you can speed up an installation by mounting some remote Nix store that already has the packages in question via NFS or `sshfs`. The environment variable `NIX_OTHER_STORES` specifies the locations of the remote Nix directories, e.g. `/mnt/remote-fs/nix`.
- New **nix-store** operations `--dump-db` and `--load-db` to dump and reload the Nix database.
- The garbage collector has a number of new options to allow only some of the garbage to be deleted. The option `--max-freed N` tells the collector to stop after at least *N* bytes have been deleted. The option `--max-links N` tells it to stop after the link count on `/nix/store` has dropped below *N*. This is useful for very large Nix stores on filesystems with a 32000 subdirectories limit (like `ext3`). The option `--use-atime` causes store paths to be deleted in order of ascending last access time. This allows non-recently used stuff to be deleted. The option `--max-atime time` specifies an upper limit to the last accessed time of paths that may be deleted. For instance,

  ```
  $ nix-store --gc -v --max-atime $(date +%s -d "2 months ago")
  ```

  deletes everything that hasn't been accessed in two months.
- **nix-env** now uses optimistic profile locking when performing an operation like installing or upgrading, instead of setting an exclusive lock on the profile. This allows multiple **nix-env -i** / **-u** / **-e** operations on the same profile in parallel. If a **nix-env** operation sees at the end that the profile was changed in the meantime by another process, it will just restart. This is generally cheap because the build results are still in the Nix store.
- The option `--dry-run` is now supported by **nix-store -r** and **nix-build**.
- The information previously shown by `--dry-run` (i.e., which derivations will be built and which paths will be substituted) is now always shown by **nix-env**, **nix-store -r** and **nix-build**. The total download size of substitutable paths is now also shown. For instance, a build will show something like

  ```
  the following derivations will be built:
    /nix/store/129sbxnk5n466zg6r1qmq1xjv9zymyy7-activate-configuration.sh.drv
    /nix/store/7mzy971rdm8l566ch8hgxaf89x7lr7ik-upstart-jobs.drv
    ...
  the following paths will be downloaded/copied (30.02 MiB):
    /nix/store/4m8pvgy2dcjgppf5b4cj5l6wyshjhalj-samba-3.2.4
    /nix/store/7h1kwcj29ip8vk26rhmx6bfjraxp0g4l-libunwind-0.98.6
    ...
  ```

- Language features:
  - @-patterns as in Haskell. For instance, in a function definition

    ```
    f = args @ {x, y, z}: ...;
    ```

    `args` refers to the argument as a whole, which is further pattern-matched against the attribute set pattern `{x, y, z}`.
  - "`...`" (ellipsis) patterns. An attribute set pattern can now say `...` at the end of the attribute name list to specify that the function takes *at least* the listed attributes, while ignoring additional attributes. For instance,

```
{stdenv, fetchurl, fuse, ...}: ...
```

defines a function that accepts any attribute set that includes at least the three listed attributes.

- New primops: `builtins.parseDrvName` (split a package name string like `"nix-0.12pre12876"` into its name and version components, e.g. `"nix"` and `"0.12pre12876"`), `builtins.compareVersions` (compare two version strings using the same algorithm that **nix-env** uses), `builtins.length` (efficiently compute the length of a list), `builtins.mul` (integer multiplication), `builtins.div` (integer division).

- **nix-prefetch-url** now supports `mirror://` URLs, provided that the environment variable `NIXPKGS_ALL` points at a Nixpkgs tree.
- Removed the commands **nix-pack-closure** and **nix-unpack-closure**. You can do almost the same thing but much more efficiently by doing `nix-store --export $(nix-store -qR paths) > closure` and `nix-store --import < closure`.
- Lots of bug fixes, including a big performance bug in the handling of `with`-expressions.

C.25. Release 0.11 (2007-12-31)

Nix 0.11 has many improvements over the previous stable release. The most important improvement is secure multi-user support. It also features many usability enhancements and language extensions, many of them prompted by NixOS, the purely functional Linux distribution based on Nix. Here is an (incomplete) list:

- Secure multi-user support. A single Nix store can now be shared between multiple (possible untrusted) users. This is an important feature for NixOS, where it allows non-root users to install software. The old setuid method for sharing a store between multiple users has been removed. Details for setting up a multi-user store can be found in the manual.
- The new command **nix-copy-closure** gives you an easy and efficient way to exchange software between machines. It copies the missing parts of the closure of a set of store path to or from a remote machine via **ssh**.
- A new kind of string literal: strings between double single-quotes (`''`) have indentation "intelligently" removed. This allows large strings (such as shell scripts or configuration file fragments in NixOS) to cleanly follow the indentation of the surrounding expression. It also requires much less escaping, since `''` is less common in most languages than `"`.
- **nix-env** `--set` modifies the current generation of a profile so that it contains exactly the specified derivation, and nothing else. For example, `nix-env -p /nix/var/nix/profiles/browser --set firefox` lets the profile named `browser` contain just Firefox.
- **nix-env** now maintains meta-information about installed packages in profiles. The meta-information is the contents of the `meta` attribute of derivations, such as `description` or `homepage`. The command `nix-env -q --xml --meta` shows all meta-information.
- **nix-env** now uses the `meta.priority` attribute of derivations to resolve filename collisions between packages. Lower priority values denote a higher priority. For instance, the GCC wrapper package and the Binutils package in Nixpkgs both have a file `bin/ld`, so previously if you tried to install both you would get a collision. Now, on the other hand, the GCC wrapper declares a higher priority than Binutils, so the former's `bin/ld` is symlinked in the user environment.
- **nix-env -i / -u**: instead of breaking package ties by version, break them by priority and version number. That is, if there are multiple packages with the same name, then pick the package with the highest priority, and only use the version if there are multiple packages with the same priority.

  This makes it possible to mark specific versions/variant in Nixpkgs more or less desirable than others. A typical example would be a beta version of some package (e.g., `gcc-4.2.0rc1`) which should not be installed even though it is the highest version, except when it is explicitly selected (e.g., `nix-env -i gcc-4.2.0rc1`).
- **nix-env --set-flag** allows meta attributes of installed packages to be modified. There are several attributes that can be usefully modified, because they affect the behaviour of **nix-env** or the user environment build script:

  - `meta.priority` can be changed to resolve filename clashes (see above).
  - `meta.keep` can be set to `true` to prevent the package from being upgraded or replaced. Useful if you want to hang on to an older version of a package.
  - `meta.active` can be set to `false` to "disable" the package. That is, no symlinks will be generated to the files of the package, but it remains part of the profile (so it won't be garbage-collected). Set it back to `true` to re-enable the package.

- **nix-env -q** now has a flag `--prebuilt-only` (`-b`) that causes **nix-env** to show only those derivations whose output is already in the Nix store or that can be substituted (i.e., downloaded from somewhere). In other words, it shows the packages that can be installed "quickly", i.e., don't need to be built from source. The `-b` flag is also available in **nix-env -i** and **nix-env -u** to filter out derivations for which no pre-built binary is available.
- The new option `--argstr` (in **nix-env**, **nix-instantiate** and **nix-build**) is like `--arg`, except that the value is a string. For example, `--argstr system i686-linux` is equivalent to `--arg system \"i686-linux\"` (note that `--argstr` prevents annoying quoting around shell arguments).
- **nix-store** has a new operation `--read-log` (`-l`) *`paths`* that shows the build log of the given paths.
- Nix now uses Berkeley DB 4.5. The database is upgraded automatically, but you should be careful not to use old versions of Nix that still use Berkeley DB 4.4.
- The option `--max-silent-time` (corresponding to the configuration setting `build-max-silent-time`) allows you to set a timeout on builds -- if a build produces no output on `stdout` or `stderr` for the given number of seconds, it is terminated. This is useful for recovering automatically from builds that are stuck in an infinite loop.
- **nix-channel**: each subscribed channel is its own attribute in the top-level expression generated for the channel. This allows disambiguation (e.g. `nix-env -i -A nixpkgs_unstable.firefox`).
- The substitutes table has been removed from the database. This makes operations such as **nix-pull** and **nix-channel --update** much, much faster.
- **nix-pull** now supports bzip2-compressed manifests. This speeds up channels.
- **nix-prefetch-url** now has a limited form of caching. This is used by **nix-channel** to prevent unnecessary downloads when the channel hasn't changed.
- **nix-prefetch-url** now by default computes the SHA-256 hash of the file instead of the MD5 hash. In calls to `fetchurl` you should pass the `sha256` attribute instead of `md5`. You can pass either a hexadecimal or a base-32 encoding of the hash.
- Nix can now perform builds in an automatically generated "chroot". This prevents a builder from accessing stuff outside of the Nix store, and thus helps ensure purity. This is an experimental feature.
- The new command **nix-store --optimise** reduces Nix store disk space usage by finding identical files in the store and hard-linking them to each other. It typically reduces the size of the store by something like 25-35%.
- `~/.nix-defexpr` can now be a directory, in which case the Nix expressions in that directory are combined into an attribute set, with the file names used as the names of the attributes. The command **nix-env --import** (which set the `~/.nix-defexpr` symlink) is removed.
- Derivations can specify the new special attribute `allowedReferences` to enforce that the references in the output of a derivation are a subset of a declared set of paths. For example, if `allowedReferences` is an empty list, then the output must not have any references. This is used in NixOS to check that generated files such as initial ramdisks for booting Linux don't have any dependencies.
- The new attribute `exportReferencesGraph` allows builders access to the references graph of their inputs. This is used in NixOS for tasks such as generating ISO-9660 images that contain a Nix store populated with the closure of certain paths.

- Fixed-output derivations (like `fetchurl`) can define the attribute `impureEnvVars` to allow external environment variables to be passed to builders. This is used in Nixpkgs to support proxy configuration, among other things.
- Several new built-in functions: `builtins.attrNames`, `builtins.filterSource`, `builtins.isAttrs`, `builtins.isFunction`, `builtins.listToAttrs`, `builtins.stringLength`, `builtins.sub`, `builtins.substring`, `throw`, `builtins.trace`, `builtins.readFile`.

C.26. Release 0.10.1 (2006-10-11)

This release fixes two somewhat obscure bugs that occur when evaluating Nix expressions that are stored inside the Nix store (`NIX-67`). These do not affect most users.

C.27. Release 0.10 (2006-10-06)

Note

This version of Nix uses Berkeley DB 4.4 instead of 4.3. The database is upgraded automatically, but you should be careful not to use old versions of Nix that still use Berkeley DB 4.3. In particular, if you use a Nix installed through Nix, you should run

```
$ nix-store --clear-substitutes
```

first.

Warning

Also, the database schema has changed slighted to fix a performance issue (see below). When you run any Nix 0.10 command for the first time, the database will be upgraded automatically. This is irreversible.

- **nix-env** usability improvements:
  - An option `--compare-versions` (or `-c`) has been added to **nix-env --query** to allow you to compare installed versions of packages to available versions, or vice versa. An easy way to see if you are up to date with what's in your subscribed channels is `nix-env -qc \*`.
  - `nix-env --query` now takes as arguments a list of package names about which to show information, just like `--install`, etc.: for example, `nix-env -q gcc`. Note that to show all derivations, you need to specify `\*`.
  - `nix-env -i pkgname` will now install the highest available version of *`pkgname`*, rather than installing all available versions (which would probably give collisions) (`NIX-31`).
  - `nix-env (-i|-u)--dry-run` now shows exactly which missing paths will be built or substituted.
  - `nix-env -qa --description` shows human-readable descriptions of packages, provided that they have a `meta.description` attribute (which most packages in Nixpkgs don't have yet).
- New language features:
  - Reference scanning (which happens after each build) is much faster and takes a constant amount of memory.
  - String interpolation. Expressions like

    ```
    "--with-freetype2-library=" + freetype + "/lib"
    ```

    can now be written as

    ```
    1 "--with-freetype2-library=${freetype}/lib"
    ```

    You can write arbitrary expressions within `${...}`, not just identifiers.
  - Multi-line string literals.

  - String concatenations can now involve derivations, as in the example `"--with-freetype2-library=" ↪ + freetype + "/lib"`. This was not previously possible because we need to register that a derivation that uses such a string is dependent on `freetype`. The evaluator now properly propagates this information. Consequently, the subpath operator (`~`) has been deprecated.
  - Default values of function arguments can now refer to other function arguments; that is, all arguments are in scope in the default values (`NIX-45`).
  - Lots of new built-in primitives, such as functions for list manipulation and integer arithmetic. See the manual for a complete list. All primops are now available in the set `builtins`, allowing one to test for the availability of primop in a backwards-compatible way.
  - Real let-expressions: `let x = ...; ... z = ...; in ...`.
- New commands **nix-pack-closure** and **nix-unpack-closure** than can be used to easily transfer a store path with all its dependencies to another machine. Very convenient whenever you have some package on your machine and you want to copy it somewhere else.
- XML support:
  - `nix-env -q --xml` prints the installed or available packages in an XML representation for easy processing by other tools.
  - `nix-instantiate --eval-only --xml` prints an XML representation of the resulting term. (The new flag `--strict` forces 'deep' evaluation of the result, i.e., list elements and attributes are evaluated recursively.)
  - In Nix expressions, the primop `builtins.toXML` converts a term to an XML representation. This is primarily useful for passing structured information to builders.
- You can now unambiguously specify which derivation to build or install in **nix-env**, **nix-instantiate** and **nix-build** using the `--attr` / `-A` flags, which takes an attribute name as argument. (Unlike symbolic package names such as `subversion-1.4.0`, attribute names in an attribute set are unique.) For instance, a quick way to perform a test build of a package in Nixpkgs is `nix-build pkgs/top-level/all-packages.nix -A foo`. `nix-env -q --attr` shows the attribute names corresponding to each derivation.
- If the top-level Nix expression used by **nix-env**, **nix-instantiate** or **nix-build** evaluates to a function whose arguments all have default values, the function will be called automatically. Also, the new command-line switch `--arg name` *`value`* can be used to specify function arguments on the command line.
- `nix-install-package --url URL` allows a package to be installed directly from the given URL.
- Nix now works behind an HTTP proxy server; just set the standard environment variables `http_proxy`, `https_proxy`, `ftp_proxy` or `all_proxy` appropriately. Functions such as `fetchurl` in Nixpkgs also respect these variables.
- `nix-build -o symlink` allows the symlink to the build result to be named something other than `result`.
- Platform support:
  - Support for 64-bit platforms, provided a suitably patched ATerm library is used. Also, files larger than 2 GiB are now supported.
  - Added support for Cygwin (Windows, `i686-cygwin`), Mac OS X on Intel (`i686-darwin`) and Linux on PowerPC (`powerpc-linux`).
  - Users of SMP and multicore machines will appreciate that the number of builds to be performed in parallel can now be specified in the configuration file in the `build-max-jobs` setting.
- Garbage collector improvements:
  - Open files (such as running programs) are now used as roots of the garbage collector. This prevents programs that have been uninstalled from being garbage collected while they are still running. The script that detects these additional runtime roots (`find-runtime-roots.pl`) is inherently system-specific, but it should work on Linux and on all platforms that have the **lsof** utility.

  - `nix-store --gc` (a.k.a. **nix-collect-garbage**) prints out the number of bytes freed on standard output. `nix-store --gc --print-dead` shows how many bytes would be freed by an actual garbage collection.
  - `nix-collect-garbage -d` removes all old generations of *all* profiles before calling the actual garbage collector (`nix-store --gc`). This is an easy way to get rid of all old packages in the Nix store.
  - **nix-store** now has an operation `--delete` to delete specific paths from the Nix store. It won't delete reachable (non-garbage) paths unless `--ignore-liveness` is specified.
- Berkeley DB 4.4's process registry feature is used to recover from crashed Nix processes.
- A performance issue has been fixed with the `referer` table, which stores the inverse of the `references` table (i.e., it tells you what store paths refer to a given path). Maintaining this table could take a quadratic amount of time, as well as a quadratic amount of Berkeley DB log file space (in particular when running the garbage collector) (`NIX-23`).
- Nix now catches the `TERM` and `HUP` signals in addition to the `INT` signal. So you can now do a `killall` ↪ `nix-store` without triggering a database recovery.
- **bsdiff** updated to version 4.3.
- Substantial performance improvements in expression evaluation and `nix-env -qa`, all thanks to Valgrind. Memory use has been reduced by a factor 8 or so. Big speedup by memoisation of path hashing.
- Lots of bug fixes, notably:
  - Make sure that the garbage collector can run successfully when the disk is full (`NIX-18`).
  - **nix-env** now locks the profile to prevent races between concurrent **nix-env** operations on the same profile (`NIX-7`).
  - Removed misleading messages from `nix-env -i` (e.g., `installing `foo'` followed by `uninstalling` ↪ `` `foo' ``) (`NIX-17`).
- Nix source distributions are a lot smaller now since we no longer include a full copy of the Berkeley DB source distribution (but only the bits we need).
- Header files are now installed so that external programs can use the Nix libraries.

C.28. Release 0.9.2 (2005-09-21)

This bug fix release fixes two problems on Mac OS X:

- If Nix was linked against statically linked versions of the ATerm or Berkeley DB library, there would be dynamic link errors at runtime.
- **nix-pull** and **nix-push** intermittently failed due to race conditions involving pipes and child processes with error messages such as `open2: open(GLOB(0x180b2e4), >&=9)failed: Bad file descriptor at` ↪ `/nix/bin/nix-pull line 77` (issue `NIX-14`).

C.29. Release 0.9.1 (2005-09-20)

This bug fix release addresses a problem with the ATerm library when the `--with-aterm` flag in **configure** was *not* used.

C.30. Release 0.9 (2005-09-16)

NOTE: this version of Nix uses Berkeley DB 4.3 instead of 4.2. The database is upgraded automatically, but you should be careful not to use old versions of Nix that still use Berkeley DB 4.2. In particular, if you use a Nix installed through Nix, you should run

```
$ nix-store --clear-substitutes
```

first.

- Unpacking of patch sequences is much faster now since we no longer do redundant unpacking and repacking of intermediate paths.
- Nix now uses Berkeley DB 4.3.
- The `derivation` primitive is lazier. Attributes of dependent derivations can mutually refer to each other (as long as there are no data dependencies on the `outPath` and `drvPath` attributes computed by `derivation`).

  For example, the expression `derivation attrs` now evaluates to (essentially)

```
attrs // {
  type = "derivation";
  outPath = derivation! attrs;
  drvPath = derivation! attrs;
}
```

  where `derivation!` is a primop that does the actual derivation instantiation (i.e., it does what `derivation` used to do). The advantage is that it allows commands such as **nix-env -qa** and **nix-env -i** to be much faster since they no longer need to instantiate all derivations, just the `name` attribute.

  Also, it allows derivations to cyclically reference each other, for example,

```
webServer = derivation {
  ...
  hostName = "svn.cs.uu.nl";
  services = [svnService];
};

svnService = derivation {
  ...
  hostName = webServer.hostName;
};
```

  Previously, this would yield a black hole (infinite recursion).
- **nix-build** now defaults to using `./default.nix` if no Nix expression is specified.
- **nix-instantiate**, when applied to a Nix expression that evaluates to a function, will call the function automatically if all its arguments have defaults.
- Nix now uses libtool to build dynamic libraries. This reduces the size of executables.
- A new list concatenation operator `++`. For example, `[1 2 3] ++ [4 5 6]` evaluates to `[1 2 3 4 5 6]`.
- Some currently undocumented primops to support low-level build management using Nix (i.e., using Nix as a Make replacement). See the commit messages for `r3578` and `r3580`.
- Various bug fixes and performance improvements.

C.31. Release 0.8.1 (2005-04-13)

This is a bug fix release.

- Patch downloading was broken.
- The garbage collector would not delete paths that had references from invalid (but substitutable) paths.

C.32. Release 0.8 (2005-04-11)

NOTE: the hashing scheme in Nix 0.8 changed (as detailed below). As a result, **nix-pull** manifests and channels built for Nix 0.7 and below will now work anymore. However, the Nix expression language has not changed, so you can still build from source. Also, existing user environments continue to work. Nix 0.8 will automatically upgrade the database schema of previous installations when it is first run.

If you get the error message

```
you have an old-style manifest `/nix/var/nix/manifests/[...]'; please
delete it
```

you should delete previously downloaded manifests:

```
$ rm /nix/var/nix/manifests/*
```

If **nix-channel** gives the error message

```
manifest `http://catamaran.labs.cs.uu.nl/dist/nix/channels/[channel]/MANIFEST'
is too old (i.e., for Nix <= 0.7)
```

then you should unsubscribe from the offending channel (**nix-channel --remove *URL***; leave out `/MANIFEST`), and subscribe to the same URL, with `channels` replaced by `channels-v3` (e.g., http://catamaran.labs.cs.uu.nl/dist/nix/channels-v3/nixpkgs-unstable).

Nix 0.8 has the following improvements:

- The cryptographic hashes used in store paths are now 160 bits long, but encoded in base-32 so that they are still only 32 characters long (e.g., `/nix/store/csw87wag8bqlqk7ipllbwypb14xainap-atk-1.9.0`). (This is actually a 160 bit truncation of a SHA-256 hash.)
- Big cleanups and simplifications of the basic store semantics. The notion of "closure store expressions" is gone (and so is the notion of "successors"); the file system references of a store path are now just stored in the database.

  For instance, given any store path, you can query its closure:

  ```
  $ nix-store -qR $(which firefox)
  ... lots of paths ...
  ```

  Also, Nix now remembers for each store path the derivation that built it (the "deriver"):

  ```
  $ nix-store -qR $(which firefox)
  /nix/store/4b0jx7vq80l9aqcnkszxhymsf1ffa5jd-firefox-1.0.1.drv
  ```

  So to see the build-time dependencies, you can do

  ```
  $ nix-store -qR $(nix-store -qd $(which firefox))
  ```

  or, in a nicer format:

  ```
  $ nix-store -q --tree $(nix-store -qd $(which firefox))
  ```

  File system references are also stored in reverse. For instance, you can query all paths that directly or indirectly use a certain Glibc:

  ```
  $ nix-store -q --referrers-closure \
      /nix/store/8lz9yc6zgmc0vlqmn2ipcpkjlmbi51vv-glibc-2.3.4
  ```

- The concept of fixed-output derivations has been formalised. Previously, functions such as `fetchurl` in Nixpkgs used a hack (namely, explicitly specifying a store path hash) to prevent changes to, say, the URL of the file from propagating upwards through the dependency graph, causing rebuilds of everything. This can now be done cleanly by specifying the `outputHash` and `outputHashAlgo` attributes. Nix itself checks that the content of the output has the specified hash. (This is important for maintaining certain invariants necessary for future work on secure shared stores.)
- One-click installation :-) It is now possible to install any top-level component in Nixpkgs directly, through the web -- see, e.g., http://catamaran.labs.cs.uu.nl/dist/nixpkgs-0.8/. All you have to do is associate `/nix/bin/nix-install-package` with the MIME type `application/nix-package` (or the extension `.nixpkg`), and clicking on a package link will cause it to be installed, with all appropriate dependencies. If you just want to install some specific application, this is easier than subscribing to a channel.

- **nix-store -r *PATHS*** now builds all the derivations PATHS in parallel. Previously it did them sequentially (though exploiting possible parallelism between subderivations). This is nice for build farms.
- **nix-channel** has new operations `--list` and `--remove`.
- New ways of installing components into user environments:
  - Copy from another user environment:

    ```
    $ nix-env -i --from-profile .../other-profile firefox
    ```

  - Install a store derivation directly (bypassing the Nix expression language entirely):

    ```
    $ nix-env -i /nix/store/z58v41v21xd3...-aterm-2.3.1.drv
    ```

    (This is used to implement **nix-install-package**, which is therefore immune to evolution in the Nix expression language.)
  - Install an already built store path directly:

    ```
    $ nix-env -i /nix/store/hsyj5pbn0d9i...-aterm-2.3.1
    ```

  - Install the result of a Nix expression specified as a command-line argument:

    ```
    $ nix-env -f .../i686-linux.nix -i -E 'x: x.firefoxWrapper'
    ```

    The difference with the normal installation mode is that `-E` does not use the `name` attributes of derivations. Therefore, this can be used to disambiguate multiple derivations with the same name.
- A hash of the contents of a store path is now stored in the database after a successful build. This allows you to check whether store paths have been tampered with: **nix-store --verify --check-contents**.
- Implemented a concurrent garbage collector. It is now always safe to run the garbage collector, even if other Nix operations are happening simultaneously.

  However, there can still be GC races if you use **nix-instantiate** and **nix-store --realise** directly to build things. To prevent races, use the `--add-root` flag of those commands.
- The garbage collector now finally deletes paths in the right order (i.e., topologically sorted under the "references" relation), thus making it safe to interrupt the collector without risking a store that violates the closure invariant.
- Likewise, the substitute mechanism now downloads files in the right order, thus preserving the closure invariant at all times.
- The result of **nix-build** is now registered as a root of the garbage collector. If the `./result` link is deleted, the GC root disappears automatically.
- The behaviour of the garbage collector can be changed globally by setting options in `/nix/etc/nix/nix.conf`.
  - `gc-keep-derivations` specifies whether deriver links should be followed when searching for live paths.
  - `gc-keep-outputs` specifies whether outputs of derivations should be followed when searching for live paths.
  - `env-keep-derivations` specifies whether user environments should store the paths of derivations when they are added (thus keeping the derivations alive).
- New **nix-env** query flags `--drv-path` and `--out-path`.
- **fetchurl** allows SHA-1 and SHA-256 in addition to MD5. Just specify the attribute `sha1` or `sha256` instead of `md5`.
- Manual updates.

C.33. Release 0.7 (2005-01-12)

- Binary patching. When upgrading components using pre-built binaries (through nix-pull / nix-channel), Nix can automatically download and apply binary patches to already installed components instead of full downloads. Patching is "smart": if there is a *sequence* of patches to an installed component, Nix will use it. Patches are currently generated automatically between Nixpkgs (pre-)releases.
- Simplifications to the substitute mechanism.
- Nix-pull now stores downloaded manifests in `/nix/var/nix/manifests`.
- Metadata on files in the Nix store is canonicalised after builds: the last-modified timestamp is set to 0 (00:00:00 1/1/1970), the mode is set to 0444 or 0555 (readable and possibly executable by all; setuid/setgid bits are dropped), and the group is set to the default. This ensures that the result of a build and an installation through a substitute is the same; and that timestamp dependencies are revealed.

C.34. Release 0.6 (2004-11-14)

- Rewrite of the normalisation engine.
  - Multiple builds can now be performed in parallel (option `-j`).
  - Distributed builds. Nix can now call a shell script to forward builds to Nix installations on remote machines, which may or may not be of the same platform type.
  - Option `--fallback` allows recovery from broken substitutes.
  - Option `--keep-going` causes building of other (unaffected) derivations to continue if one failed.
- Improvements to the garbage collector (i.e., it should actually work now).
- Setuid Nix installations allow a Nix store to be shared among multiple users.
- Substitute registration is much faster now.
- A utility **nix-build** to build a Nix expression and create a symlink to the result int the current directory; useful for testing Nix derivations.
- Manual updates.
- **nix-env** changes:
  - Derivations for other platforms are filtered out (which can be overridden using `--system-filter`).
  - `--install` by default now uninstall previous derivations with the same name.
  - `--upgrade` allows upgrading to a specific version.
  - New operation `--delete-generations` to remove profile generations (necessary for effective garbage collection).
  - Nicer output (sorted, columnised).
- More sensible verbosity levels all around (builder output is now shown always, unless `-Q` is given).
- Nix expression language changes:
  - New language construct: `with E1`; *`E2`* brings all attributes defined in the attribute set *`E1`* in scope in *`E2`*.
  - Added a `map` function.
  - Various new operators (e.g., string concatenation).
- Expression evaluation is much faster.
- An Emacs mode for editing Nix expressions (with syntax highlighting and indentation) has been added.
- Many bug fixes.

C.35. Release 0.5 and earlier

Please refer to the Subversion commit log messages.