

Nix User's Guide

Draft (Version 0.16)

Eelco Dolstra

Delft University of Technology
Department of Software Technology

Copyright © 2004, 2005, 2006, 2007, 2008, 2009, 2010 Eelco Dolstra

Table of Contents

[1. Introduction](#)

- [1.1. About Nix](#)
- [1.2. About us](#)
- [1.3. About this manual](#)
- [1.4. License](#)
- [1.5. More information](#)

[2. Quick Start](#)

[3. Installation](#)

- [3.1. Supported platforms](#)
- [3.2. Obtaining Nix](#)
- [3.3. Prerequisites](#)
- [3.4. Building Nix from source](#)
- [3.5. Installing a binary distribution](#)
- [3.6. Security](#)
 - [3.6.1. Single-user mode](#)
 - [3.6.2. Multi-user mode](#)
 - [3.6.2.1. Setting up the build users](#)
 - [3.6.2.2. Nix store/database owned by root](#)
 - [3.6.2.3. Nix store/database not owned by root](#)
 - [3.6.2.4. Restricting access](#)

[3.7. Using Nix](#)

[4. Package Management](#)

- [4.1. Basic package management](#)
- [4.2. Profiles](#)
- [4.3. Garbage collection](#)
 - [4.3.1. Garbage collector roots](#)
- [4.4. Channels](#)
- [4.5. One-click installs](#)
- [4.6. Sharing packages between machines](#)

[5. Writing Nix Expressions](#)

[5.1. A simple Nix expression](#)

[5.1.1. The Nix expression](#)

[5.1.2. The builder](#)

[5.1.3. Composition](#)

[5.1.4. Testing](#)

[5.1.5. The generic builder](#)

[5.2. The Nix expression language](#)

[5.2.1. Values](#)

[5.2.2. Language constructs](#)

[5.2.3. Operators](#)

[5.2.4. Derivations](#)

[5.2.4.1. Advanced attributes](#)

[5.2.5. Built-in functions](#)

[5.3. The standard environment](#)

[5.3.1. Customising the generic builder](#)

[5.3.2. Debugging failed builds](#)

[6. Setting up a Build Farm](#)

[6.1. Overview](#)

[6.2. Setting up distributed builds](#)

[A. Command Reference](#)

[A.1. Common options](#)

[A.2. Common environment variables](#)

[A.3. Nix configuration file](#)

[A.4. Main commands](#)

[nix-env](#) — manipulate or query Nix user environments

[nix-instantiate](#) — instantiate store derivations from Nix expressions

[nix-store](#) — manipulate or query the Nix store

[A.5. Utilities](#)

[nix-build](#) — build a Nix expression

[nix-channel](#) — manage Nix channels

[nix-collect-garbage](#) — delete unreachable store paths

[nix-copy-closure](#) — copy a closure to or from a remote machine via SSH

[nix-hash](#) — compute the cryptographic hash of a path

[nix-install-package](#) — install a Nix Package file

[nix-prefetch-url](#) — copy a file from a URL into the store and print its MD5 hash

[nix-pull](#) — pull substitutes from a network cache

[nix-push](#) — push store paths onto a network cache

[nix-worker](#) — Nix multi-user support daemon

[B. Troubleshooting](#)

[B.1. Collisions in **nix-env**](#)

[B.2. “Too many links” error in the Nix store](#)

[C. Glossary](#)

[D. Nix Release Notes](#)

[D.1. Release 0.16 \(August 17, 2010\)](#)

[D.2. Release 0.15 \(March 17, 2010\)](#)

[D.3. Release 0.14 \(February 4, 2010\)](#)

[D.4. Release 0.13 \(November 5, 2009\)](#)

[D.5. Release 0.12 \(November 20, 2008\)](#)

[D.6. Release 0.11 \(December 31, 2007\)](#)

[D.7. Release 0.10.1 \(October 11, 2006\)](#)

[D.8. Release 0.10 \(October 6, 2006\)](#)

[D.9. Release 0.9.2 \(September 21, 2005\)](#)

[D.10. Release 0.9.1 \(September 20, 2005\)](#)

[D.11. Release 0.9 \(September 16, 2005\)](#)

[D.12. Release 0.8.1 \(April 13, 2005\)](#)

[D.13. Release 0.8 \(April 11, 2005\)](#)

[D.14. Release 0.7 \(January 12, 2005\)](#)

[D.15. Release 0.6 \(November 14, 2004\)](#)

[D.16. Release 0.5 and earlier](#)

List of Figures

4.1. [User environments](#)

List of Tables

5.1. [Operators](#)

List of Examples

5.1. [Nix expression for GNU Hello \(default.nix\)](#)

5.2. [Build script for GNU Hello \(builder.sh\)](#)

5.3. [Composing GNU Hello \(all-packages.nix\)](#)

5.4. [Build script using the generic build functions](#)

5.5. [Nix expression for Subversion](#)

5.6. [Passing information to a builder using toXML](#)

5.7. [XML representation produced by toXML](#)

6.1. [Remote machine configuration: remote-systems.conf](#)

A.1. [Nix configuration file](#)

Chapter 1. Introduction

Table of Contents

[1.1. About Nix](#)

[1.2. About us](#)

[1.3. About this manual](#)

[1.4. License](#)

[1.5. More information](#)

1.1. 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/r8vvq9kq18pz08v249h8my6r9vs7s0n3-firefox-2.0.0.1/

where `r8vvq9kq...` 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/51bfaxb722zp...-openssl-0.9.8d/include`, so if a package builds correctly on your system, this is because you specified the dependency explicitly.

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

Starting at version 0.11, 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 install 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 be 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 download a pre-built binary instead if it knows about it. *Nix channels* provide Nix expressions along with pre-built binaries.

Binary patching

In addition to downloading binaries automatically if they're available, Nix can download binary deltas that patch an existing package in the Nix store into a new version. This speeds up upgrades.

Nix Packages collection

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

Service deployment

Nix can be used not only for rolling out packages, but also complete *configurations* of services. This is done by treating all the static bits of a service (such as software packages, configuration files, control scripts, static web pages, etc.) as “packages” that can be built by Nix expressions. As a result, all the features above apply to services as well: for instance, you can roll back a web server configuration if a configuration change turns out to be undesirable, you can easily have multiple instances of a service (e.g., a test and production server), and because the whole service is built in a purely functional way from a Nix expression, it is repeatable so you can easily reproduce the service on another machine.

Portability

Nix should run on most Unix systems, including Linux, FreeBSD and Mac OS X. It is also supported on

Windows using Cygwin.

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's possible to easily 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](#).

1.2. About us

Nix was originally developed at the [Department of Information and Computing Sciences](#), Utrecht University by the [TraCE project](#) (2003-2008). The project was funded by the Software Engineering Research Program [Jacquard](#) to improve the support for variability in software systems. Further funding is now provided by the NIRICT LaQuSo Build Farm project.

1.3. About this manual

This manual tells you how to install and use Nix and how to write Nix expressions for software not already in the Nix Packages collection. It also discusses some advanced topics, such as setting up a Nix-based build farm.

1.4. License

Nix is free software; you can redistribute it and/or modify it under the terms of the [GNU Lesser General Public License](#) as published by the [Free Software Foundation](#); either version 2.1 of the License, or (at your option) any later version. Nix is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

1.5. More information

Some background information on Nix can be found in a number of papers. The ICSE 2004 paper [Imposing a Memory Management Discipline on Software Deployment](#) discusses the hashing mechanism used to ensure reliable dependency identification and non-interference between different versions and variants of packages. The LISA 2004 paper [Nix: A Safe and Policy-Free System for Software Deployment](#) gives a more general discussion of Nix from a system-administration perspective. The CBSE 2005 paper [Efficient Upgrading in a Purely Functional Component Deployment Model](#) is about transparent patch deployment in Nix. The SCM-12 paper [Service Configuration Management](#) shows how services (e.g., web servers) can be deployed and managed through Nix. A short overview of NixOS is given in the HotOS XI paper [Purely Functional System Configuration Management](#). The Nix homepage has [an up-to-date list of Nix-related papers](#).

Nix is the subject of Eelco Dolstra's PhD thesis [The Purely Functional Software Deployment Model](#), which contains most of the papers listed above.

Nix has a homepage at <http://nixos.org/>.

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 the following chapters.

1. Download a source tarball, RPM or Deb from <http://nixos.org/>. Build source distributions using the regular sequence:

```
$ tar xvfj nix-version.tar.bz2
$ ./configure
$ make
$ make install (as root)
```

This will install the Nix binaries in `/usr/local` and keep the Nix store and other state in `/nix`. You can change the former by specifying `--prefix=path`. The location of the store can be changed using `--with-store-dir=path`. However, you shouldn't change the store location, if at all possible, since that will make it impossible to use pre-built binaries from the Nixpkgs channel and other channels. The location of the state can be changed using `--localstatedir=path`.

2. You should add `prefix/etc/profile.d/nix.sh` to your `~/.bashrc` (or some other login file).

3. Subscribe to the Nix Packages channel.

```
$ nix-channel --add \
  http://nixos.org/releases/nixpkgs/channels/nixpkgs-unstable
```

4. Download the latest Nix expressions available in the channel.

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

Note that this in itself doesn't download any packages, it just downloads the Nix expressions that build them and stores them somewhere (under `~/.nix-defexpr`, in case you're curious). Also, it registers the fact that pre-built binaries are available remotely.

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

```
$ nix-env -qa '*' (mind the quotes!)
docbook-xml-4.2
firefox-1.0pre-PR-0.10.1
hello-2.1.1
libxslt-1.1.0
...
```

6. Install some packages from the channel:

```
$ nix-env -i hello firefox ...
```

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

7. Test that they work:

```
$ which hello
/home/eelco/.nix-profile/bin/hello
$ hello
Hello, world!
$ firefox
(read Slashdot or something)
```

8. Uninstall a package:

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

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

```
$ nix-channel --update
$ 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).

10. You can also install specific packages directly from your web browser. For instance, you can go to <http://hydra.nixos.org/jobset/nixpkgs/trunk/channel/latest> and click on any link for the individual packages for your platform. Associate application/nix-package with the program /nix/bin/nix-install-package. A window should appear asking you whether it's okay to install the package. Say y. The package and all its dependencies will be installed.
11. 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
```

12. 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
```

Chapter 3. Installation

Table of Contents

[3.1. Supported platforms](#)

[3.2. Obtaining Nix](#)

[3.3. Prerequisites](#)

[3.4. Building Nix from source](#)

[3.5. Installing a binary distribution](#)

[3.6. Security](#)

[3.6.1. Single-user mode](#)

[3.6.2. Multi-user mode](#)

[3.6.2.1. Setting up the build users](#)

[3.6.2.2. Nix store/database owned by root](#)

[3.6.2.3. Nix store/database not owned by root](#)

[3.6.2.4. Restricting access](#)

[3.7. Using Nix](#)

3.1. Supported platforms

Nix is currently supported on the following platforms:

- Linux (particularly on x86, x86_64, and PowerPC).
- Mac OS X, both on Intel and PowerPC.
- FreeBSD (only tested on Intel).
- Windows through [Cygwin](#).

Warning

On Cygwin, Nix *must* be installed on an NTFS partition. It will not work correctly on a FAT partition.

Nix is pretty portable, so it should work on most other Unix platforms as well.

3.2. Obtaining Nix

The easiest way to obtain Nix is to download a [source distribution](#). RPMs for Red Hat, SuSE, and Fedora Core are also available.

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

```
$ svn checkout https://svn.nixos.org/repos/nix/nix/trunk nix
```

Likewise, specific releases can be obtained from the [tags directory](#) of the repository.

3.3. Prerequisites

The following prerequisites only apply when you build from source. Binary releases (e.g., RPMs) have no prerequisites.

A fairly recent version of GCC/G++ is required. Version 2.95 and higher should work.

To build this manual and the man-pages you need the **xmllint** and **xsltproc** programs, which are part of the `libxm12` 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 Subversion repository.

To build the parser, very *recent* versions of Bison and Flex are required. (This is because Nix needs GLR support in Bison and reentrancy support in Flex.) For Bison, you need version 2.3 or higher (1.875 does *not* work), which can be obtained from the [GNU FTP server](#). For Flex, you need version 2.5.33, 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 Subversion repository.

Nix uses the bzip2 compressor (including the bzip2 library). It is included in the Nix source distribution. If you build from the Subversion repository, you must download it yourself and place it in the `externals/` directory. See `externals/Makefile.am` for the precise URLs of this packages. Alternatively, if you already have it installed, you can use **configure**'s `--with-bzip2` options to point to their respective locations.

3.4. Building Nix from source

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

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

When building from the Subversion 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`.

If you want to rebuild the documentation, pass the full path to the DocBook RELAX NG schemas and to the DocBook XSL stylesheets using the `--with-docbook-rng=path` and `--with-docbook-xsl=path` options.

3.5. Installing a binary distribution

RPM and Deb packages of Nix for a number of different versions of Fedora, openSUSE, Debian and Ubuntu can be downloaded from <http://nixos.org/>. Once downloaded, the RPMs can be installed or upgraded using **rpm -U**. For example,

```
$ rpm -U nix-0.13pre18104-1.i386.rpm
```

Likewise, for a Deb package:

```
$ dpkg -i nix_0.13pre18104-1_amd64.deb
```

Nix can be uninstalled using **rpm -e nix** or **dpkg -r nix**. After this you should manually remove the Nix store and other auxiliary data, if desired:

```
$ rm -rf /nix/store
$ rm -rf /nix/var
```

3.6. 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.

3.6.1. 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.

3.6.2. 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 can run [nix-pull](#) to register the availability of pre-built binaries. However, those registrations are shared by all users, so they still get the benefit from **nix-pulls** done by root.

3.6.2.1. 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* (usually called `nixbld`). This group should have no other members. The build users should not be members of any other group.

Here is a typical `/etc/group` definition of the build users group with 10 build users:

```
nixbld:!:30000:nixbld1,nixbld2,nixbld3,nixbld4,nixbld5,nixbld6,nixbld7,nixbld8,nixbld9,nixbld10
```

In this example the `nixbld` group has UID 30000, but of course it can be anything that doesn't collide with an existing group.

Here is the corresponding part of `/etc/passwd`:

```
nixbld1:x:30001:65534:Nix build user 1:/var/empty:/noshell
nixbld2:x:30002:65534:Nix build user 2:/var/empty:/noshell
nixbld3:x:30003:65534:Nix build user 3:/var/empty:/noshell
...
nixbld10:x:30010:65534:Nix build user 10:/var/empty:/noshell
```

The home directory of the build users should not exist or should be an empty directory to which they do not have write access.

The build users should have write access to the Nix store, but they should not have the right to delete files. Thus the Nix store's group should be the build users group, and it should have the sticky bit turned on (like `/tmp`):

```
$ chgrp nixbld /nix/store
$ chmod 1777 /nix/store
```

Finally, you should tell Nix to use the build users by specifying the build users group in the [build-users-group option](#) in the [Nix configuration file](#) (`/nix/etc/nix/nix.conf`):

```
build-users-group = nixbld
```

3.6.2.2. Nix store/database owned by root

The simplest setup is to let root own the Nix store and database. I.e.,

```
$ chown -R root /nix/store /nix/var/nix
```

The [Nix daemon](#) should be started as follows (as root):

```
$ nix-worker --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.

3.6.2.3. Nix store/database not owned by root

It is also possible to let the Nix store and database be owned by a non-root user, which should be more secure^[1]. Typically, this user is a special account called `nix`, but it can be named anything. It should own the Nix store and database:

```
$ chown -R root /nix/store /nix/var/nix
```

and of course **nix-worker --daemon** should be started under that user, e.g.,

```
$ su - nix -c "exec /nix/bin/nix-worker --daemon"
```

There is a catch, though: non-root users cannot start builds under the build user accounts, since the `setuid` system call is obviously privileged. To allow a non-root Nix daemon to use the build user feature, it calls a `setuid-root` helper program, **nix-setuid-helper**. This program is installed in `prefix/libexec/nix-setuid-helper`. To set the permissions properly (Nix's **make install** doesn't do this, since we don't want to ship `setuid-root` programs out-of-the-box):

```
$ chown root.root /nix/libexec/nix-setuid-helper
$ chmod 4755 /nix/libexec/nix-setuid-helper
```

(This example assumes that the Nix binaries are installed in `/nix`.)

Of course, the **nix-setuid-helper** command should not be usable by just anybody, since then anybody could run commands under the Nix build user accounts. For that reason there is a configuration file `/etc/nix-setuid.conf` that restricts the use of the helper. This file should be a text file containing precisely two lines, the first being the Nix daemon user and the second being the build users group, e.g.,

```
nix
nixbld
```

The `setuid-helper` barfs if it is called by a user other than the one specified on the first line, or if it is asked to execute a build under a user who is not a member of the group specified on the second line. The file `/etc/nix-setuid.conf` must be owned by root, and must not be group- or world-writable. The `setuid-helper` barfs if this is not the case.

3.6.2.4. 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/socket`, so they cannot perform Nix operations.

3.7. Using Nix

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 `~/.bashrc` (or similar), like this:

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

[1] Note however that even when the Nix daemon runs as root, not *that* much code is executed as root: Nix expression evaluation is performed by the calling (unprivileged) user, and builds are performed under the special build user accounts. So only the code that accesses the database and starts builds is executed as root.

Chapter 4. Package Management

Table of Contents

[4.1. Basic package management](#)

[4.2. Profiles](#)

[4.3. Garbage collection](#)

[4.3.1. Garbage collector roots](#)

[4.4. Channels](#)

[4.5. One-click installs](#)

[4.6. Sharing packages between machines](#)

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 [Chapter 5, Writing Nix Expressions](#).

4.1. 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 Nix 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 Nix Package collection; you could write your own Nix expressions based on it, or completely new ones.) You can download the latest version from <http://nixos.org/nixpkgs/download.html>.

Assuming that you have downloaded and unpacked a release of Nix Packages, you can view the set of available packages in the release:

```
$ nix-env -qaf nixpkgs-version '*'
ant-blackdown-1.4.2
aterm-2.2
bash-3.0
binutils-2.15
```

```
bison-1.875d
blackdown-1.4.2
bzip2-1.0.2
...
```

where `nixpkgs-version` is where you've unpacked the release. The flag `-q` specifies a query operation; `-a` means that you want to show the “available” (i.e., installable) packages, as opposed to the installed packages; and `-f nixpkgs-version` specifies the source of the packages. The argument `'*'` shows all installable packages. (The quotes are necessary to prevent shell expansion.) You can also select specific packages by name:

```
$ nix-env -qaf nixpkgs-version gcc
gcc-3.4.6
gcc-4.0.3
gcc-4.1.1
```

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 -qasf nixpkgs-version '*'
...
-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.

So now that we have a set of Nix expressions we can build the packages contained in them. This is done using `nix-env -i`. For instance,

```
$ nix-env -f nixpkgs-version -i subversion
```

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

When you do this for the first time, Nix will start building Subversion and all its dependencies. This will take quite a while — typically an hour or two on modern machines. Fortunately, there is a faster way (so do a Ctrl-C on that install operation!): you just need to tell Nix that pre-built binaries of all those packages are available somewhere. This is done using the **nix-pull** command, which must be supplied with a URL containing a *manifest* describing what binaries are available. This URL should correspond to the Nix Packages release that you're using. For instance, if you obtained a release from <http://nixos.org/releases/nixpkgs/nixpkgs-0.12pre11712-4lrp7j8x>, then you should do:

```
$ nix-pull http://nixos.org/releases/nixpkgs/nixpkgs-0.12pre11712-4lrp7j8x/MANIFEST
```

If you then issue the installation command, it should start downloading binaries from `nixos.org`, instead of building them from source. This might still take a while since all dependencies must be downloaded, but on a reasonably fast connection such as an DSL line it's on the order of a few minutes.

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 -f nixpkgs-version -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 -f nixpkgs-version -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'
```

If you grow bored of specifying the Nix expressions using `-f` all the time, you can set a default location:

```
$ nix-env -I nixpkgs-version
```

After this you can just say, for instance, `nix-env -u '*'`.^[2]

4.2. 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/5mq2jcn36ld1mh93yj1n8s9c95pj7c5s-subversion-1.1.2`. The long strings prefixed to the directory names are cryptographic hashes^[3] 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 4.1, “User environments”](#) shows a part of a typical Nix store.

Figure 4.1. 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 4.1, “User environments”](#) the user environment `/nix/store/5mq2jcn36ld1...-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 mozilla
```

on a set of Nix expressions that contained Mozilla 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
```

Actually, there is another level of indirection not shown in the figure above. 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
```

```
$ 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 [Section 4.3, "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.

4.3. 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
```

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

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

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.

4.3.1. 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.

4.4. 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, use **nix-pull** to register pre-built binaries (if available), 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 http://nixos.org/releases/nixpkgs/channels/nixpkgs-unstable
```

subscribes you to a channel that always contains that latest version of the Nix Packages collection. (Instead of `nixpkgs-unstable` you could also subscribe to `nixpkgs-stable`, which should have a higher level of stability, but right now is just outdated.) Subscribing really just means that the URL is added to the file `~/.nix-channels`. Right now there is no command to “unsubscribe”; you should just edit that file manually and delete the offending URL.

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

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

This downloads the Nix expressions in every channel (downloaded from `url/nixexprs.tar.bz2`) and registers any available pre-built binaries in every channel (by **`nix-pulling url/MANIFEST`**). It also makes the union of each channel’s Nix expressions the default for **`nix-env`** operations. Consequently, you can then say

```
$ nix-env -u '*'
```

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

4.5. One-click installs

Often, when you want to install a specific package (e.g., from the [Nix Packages collection](#)), subscribing to a channel is a bit cumbersome. And channels don’t help you at all if you want to install an older version of a package than the one provided by the current contents of the channel, or a package that has been removed from the channel. That’s when *one-click installs* come in handy: you can just go to the web page that contains the package, click on it, and it will be installed with all the necessary dependencies.

For instance, you can go to <http://hydra.nixos.org/jobset/nixpkgs/trunk/channel/latest> and click on any link for the individual packages for your platform. The first time you do this, your browser will ask what to do with `application/nix-package` files. You should open them with `/nix/bin/nix-install-package`. This will open a window that asks you to confirm that you want to install the package. When you answer Y, the package and all its dependencies will be installed. This is a binary deployment mechanism — you get packages pre-compiled for the selected platform type.

You can also install `application/nix-package` files from the command line directly. See [`nix-install-package\(1\)`](#) for details.

4.6. 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.

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

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

Finally, if you can mount the Nix store of a remote machine in your local filesystem, Nix can copy paths from the remote Nix store to the local Nix store *on demand*. For instance, suppose that you mount a remote machine containing a Nix store via [sshfs](#):

```
$ sshfs alice@itchy.example.org:/ /mnt
```

You should then set the `NIX_OTHER_STORES` environment variable to tell Nix about this remote Nix store:

```
$ export NIX_OTHER_STORES=/mnt/nix
```

Then if you do any Nix operation, e.g.

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

and Nix has to build a path that it sees is already present in `/mnt/nix`, then it will just copy from there instead of building it from source.

[2] Setting a default using `-I` currently clashes with using Nix channels, since `nix-channel --update` calls `nix-env -I` to set the default to the Nix expressions it downloaded from the channel, replacing whatever default you had set.

[3] 160-bit truncations of SHA-256 hashes encoded in a base-32 notation, to be precise.

Chapter 5. Writing Nix Expressions

Table of Contents

[5.1. A simple Nix expression](#)

[5.1.1. The Nix expression](#)

[5.1.2. The builder](#)

[5.1.3. Composition](#)

[5.1.4. Testing](#)

[5.1.5. The generic builder](#)

[5.2. The Nix expression language](#)

[5.2.1. Values](#)

[5.2.2. Language constructs](#)

[5.2.3. Operators](#)

[5.2.4. Derivations](#)

[5.2.4.1. Advanced attributes](#)

[5.2.5. Built-in functions](#)

[5.3. The standard environment](#)

[5.3.1. Customising the generic builder](#)

[5.3.2. Debugging failed builds](#)

This chapter shows you how to write Nix expressions, which are the things that tell 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.

5.1. 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^[4] 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.

5.1.1. The Nix expression


Example 5.1. 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;  
    md5 = "70c9ccf9fac07f762c24f2df2290784d";  
  };  
  inherit perl;   
}
```


[Example 5.1, “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):


 1 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.


 2 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*. An attribute set is just a list of key/value pairs where each value is an arbitrary Nix expression. They take the general form `{name1 = expr1; ... nameN = exprN;}`.

 3 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`.

 4 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 5.2, “Build script for GNU Hello \(builder.sh\)”](#), discussed below.

 5 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 an MD5 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).

 6 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.


5.1.2. The builder


Example 5.2. Build script for GNU Hello (builder.sh)

```
source $stdenv/setup  1
```


```
PATH=$perl/bin:$PATH  2
```

```
tar xvfz $src  3  
cd hello-*
```


```
./configure --prefix=$out  4
```


```
make   
make install
```

[Example 5.2, “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:


 1 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^[5]. 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 5.1, “Nix expression for GNU Hello \(default.nix\)”](#), but `mkDerivation` adds it automatically.)

 2 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.

 3 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.

 4 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.



 5 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.

5.1.3. Composition

Example 5.3. Composing GNU Hello (`all-packages.nix`)

...

```
rec {   
  
  hello = (import ../applications/misc/hello/ex-1  2) {  3  
    inherit fetchurl stdenv perl;  
  };  
  
  perl = (import ../development/interpreters/perl) {  4  
    inherit fetchurl stdenv;
```

```


};


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

stdenv = ...;
}


```

The Nix expression in [Example 5.1, “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 5.3, “Composing GNU Hello \(all-packages.nix\)”](#) shows some fragments of `all-packages.nix`.


 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.

 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 5.1, “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.

 This is where the actual composition takes place. Here we *call* the function imported from `../applications/misc/hello/ex-1` with an attribute 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 5.1, “Nix expression for GNU Hello \(default.nix\)”](#)).

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

5.1.4. Testing

You can now try to build Hello. Of course, you could do `nix-env -f pkgs/top-level/all-packages.nix -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 pkgs/top-level/all-packages.nix -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 from `all-packages.nix`. 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 a 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 path will not be registered as valid. If you try to build the derivation again, Nix will remove the output path if it exists (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 pkgs/top-level/all-packages.nix -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.




5.1.5. The generic builder


Recall from [Example 5.2, “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 is shown in [Example 5.4, “Build script using the generic build functions”](#).


Example 5.4. Build script using the generic build functions

```
buildInputs="$perl" 1
source $stdenv/setup 2
genericBuild 3
```

1 The `buildInputs` variable tells `setup` to use the indicated packages as “inputs”. This means that if a

package provides a `bin` subdirectory, it's added to `PATH`; if it has a `include` subdirectory, it's added to GCC's header search path; and so on.^[6]

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

 3 The final step calls the shell function `genericBuild`, which performs the steps that were done explicitly in [Example 5.2, “Build script for GNU Hello \(`builder.sh`\)”](#). The generic builder is smart enough to figure out whether to unpack the sources using `gzip`, `bzip2`, etc. It can be customised in many ways; see [Section 5.3, “The standard environment”](#).

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.

5.2. The 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. It's main job is to describe packages, compositions of packages, and the variability within packages.

This section presents the various features of the language.

5.2.1. 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 three 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 ', i.e., '\${. '' can be escaped by prefixing it with ', i.e., '''. Finally, linefeed, carriage-return and tab characters can be writted as '\n', '\r', '\t.

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](https://tools.ietf.org/html/rfc2396) 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.

- *Integers*, e.g., 123.
- *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^[7]. 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 `../xyzyz/fnord.nix`, the absolutised path is `/foo/xyzyz/fnord.nix`.

- *Booleans* with values `true` and `false`.

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 an attribute set.

Attribute sets

Attribute sets are really the core of the language, since ultimately it's all about creating derivations, which are really just sets of attributes to be passed to build scripts.

Attribute sets are just a list of name/value pairs 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 an attribute set with attributes named `x`, `test`, `y`. The order of the attributes is irrelevant. An attribute name may only occur once.

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

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

evaluates to `"Foo"`.

5.2.2. Language constructs

Recursive attribute sets

Recursive attribute sets are just normal attribute 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) attribute set, attributes are not added to the lexical scope; in a recursive set, they are.

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

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

does not terminate^[8].

Let-expressions

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

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

evaluates to "foobar".

Note

There is also an obsolete form of let-expression, `let { attrs }`, which is translated to `rec { attrs }.body`. That is, the body of the let-expression is the *body* attribute of the attribute set.

Inheriting attributes

When defining an attribute set 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;  
}
```

evaluates 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 attribute 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 = ...;  
libjpeg = ...;  
...
```

the attribute 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) attribute 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"]`.

- An *attribute set pattern* of the form `{name1, name2, ..., nameN}` matches an attribute 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 requires that the argument matches with the patterns on the left- and right-hand side of the `@`-sign. For example:

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

Here `args` is bound to the entire argument, which is further matches against the pattern `{x, y, z, ...}`.

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 5.5. 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 5.5, “Nix expression for Subversion”](#) show how assertions are used in the Nix expression for Subversion.

-  This assertion states that if Subversion is to have support for local repositories, then Berkeley DB is needed. So if the Subversion function is called with the `localServer` argument set to true but the `db4` argument set to null, then the evaluation fails.
-  This is a more subtle condition: if Subversion is built with Apache (`httpServer`) support, then the Expat library (an XML library) used by Subversion should be same as the one used by Apache. This is because in this configuration Subversion code ends up being linked with Apache code, and if the Expat libraries do not match, a build- or runtime link error or incompatibility might occur.
-  This assertion says that in order for Subversion to have SSL support (so that it can access https URLs), an OpenSSL library must be passed. Additionally, it says that *if* Apache support is enabled, then Apache's OpenSSL should match Subversion's. (Note that if Apache support is not enabled, we don't care about Apache's OpenSSL.)
-  The conditional here is not really related to assertions, but is worth pointing out: it ensures that if SSL support is disabled, then the Subversion derivation is not dependent on OpenSSL, even if a non-null value was passed. This prevents an unnecessary rebuild of Subversion if OpenSSL changes.

With-expressions

A with-expression,

with *e1*; *e2*

introduces the attribute 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 rec-expression.

Comments

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

5.2.3. Operators

[Table 5.1, “Operators”](#) lists the operators in the Nix expression language, in order of precedence (from strongest to weakest binding).

Table 5.1. Operators

Syntax	Associativity	Description
<i>e</i> . <i>id</i>	none	Select attribute named <i>id</i> from attribute set <i>e</i> . Abort evaluation if the attribute doesn't exist.
<i>e1</i> <i>e2</i>	left	Call function <i>e1</i> with argument <i>e2</i> .
<i>e</i> ? <i>id</i>	none	Test whether attribute set <i>e</i> contains an attribute named <i>id</i> ; return true or false.
<i>e1</i> ++ <i>e2</i>	right	List concatenation.
<i>e1</i> + <i>e2</i>	left	String or path concatenation.
! <i>e</i>	left	Boolean negation.
<i>e1</i> // <i>e2</i>	right	Return an attribute set consisting of the attributes in <i>e1</i> and <i>e2</i> (with the latter taking precedence over the former in case of equally named attributes).
<i>e1</i> == <i>e2</i>	none	Equality.
<i>e1</i> != <i>e2</i>	none	Inequality.
<i>e1</i> && <i>e2</i>	left	Logical AND.
<i>e1</i> <i>e2</i>	left	Logical OR.
<i>e1</i> -> <i>e2</i>	none	Logical implication (equivalent to ! <i>e1</i> <i>e2</i>).

5.2.4. Derivations

The most important built-in function is `derivation`, which is used to describe a single derivation (a build action). It takes as input an attribute 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 `"powerpc-darwin"`^[9] 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 [Section 6.2, “Setting up distributed 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 hash in the output path 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, URIs, and integers 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; the 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.

(Note that `mkDerivation` in the 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 [Section 5.3, “The standard environment”](#).)

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`).
- `out` is set to point to the output path of the derivation, which is a subdirectory of the Nix store. The output path is a concatenation of the cryptographic hash of all build inputs, and the `name` attribute.
- If the 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 the output for references to the paths of the inputs. These so-called *retained dependencies* could be used when the output of the derivation is used (e.g., when it's executed or used as input to another derivation), so if we deploy the derivation, we should copy the retained dependencies as well. The scan is performed by looking for the hash parts of file names of the inputs.
- 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.

5.2.4.1. 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. 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.

`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).

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;
  md5 = "70c9ccf9fac07f762c24f2df2290784d";
}
```

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;
  md5 = "70c9ccf9fac07f762c24f2df2290784d";
}
```

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, md5}:

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 MD5 hash md5.
  outputHashMode = "flat";
  outputHashAlgo = "md5";
  outputHash = md5;

  inherit url;
}
```

The `outputHashAlgo` attribute specifies the hash algorithm used to compute the hash. It can currently be `"md5"`, `"sha1"` or `"sha256"`.

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 **md5sum** 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.)

`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.

5.2.5. 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 an attribute set that contains all built-in functions and values. For instance, `derivation` is also available as `builtins.derivation`.

`abort s`

Abort Nix expression evaluation, print error message `s`.

`builtins.add e1 e2`

Return the sum of the integers `e1` and `e2`.

`builtins.attrNames attrs`

Return the names of the attributes in the attribute set `attrs` in a sorted list. For instance, `builtins.attrNames {y = 1; x = "foo";}` evaluates to `["x" "y"]`. There is no built-in function `attrValues`, but you can easily define it yourself:

```
attrValues = attrs: map (name: builtins.getAttr name attrs) (builtins.attrNames attrs);
```

`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`

The attribute 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.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 "powerpc-darwin".

`derivation attrs`

derivation is described in [Section 5.2.4, "Derivations"](#).

`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 integers *e1* and *e2*.

`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).

`builtins.getAttr s attrs`

`getAttr` returns the attribute named `s` from the attribute set `attrs`. 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 attrs`

`hasAttr` returns true if the attribute set `attrs` 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.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`

Load, parse and return the Nix expression in the file `path`. 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 an attribute set or a function) in a separate file, and use it from Nix expressions in other files.

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 an attribute 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 an attribute 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 a int, and false otherwise.

`builtins.isBool e`

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

`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 integer `e1` is less than the integer `e2`, and false otherwise. Evaluation aborts if either `e1` or `e2` does not evaluate to an integer.

`builtins.listToAttrs e`

Construct an attribute set from a list specifying the names and values of each attribute. Each element of the list should be an attribute 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`

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.mul e1 e2`

Return the product of the integers *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 an attribute set {name, version}. Thus, `builtins.parseDrvName "nix-0.12pre12876"` returns {name = "nix"; version = "0.12pre12876";}.

`builtins.pathExists path`

Return true if the path *path* exists, and false otherwise. One application of this function is to conditionally include a Nix expression containing user configuration:

```
let
  fileName = builtins.getEnv "CONFIG_FILE";
  config =
    if fileName != "" && builtins.pathExists (builtins.toPath fileName)
    then import (builtins.toPath fileName)
    else { someSetting = false; }; # default configuration
in config.someSetting
```

(Note that CONFIG_FILE must be an absolute path for this to work.)

`builtins.readFile path`

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

`removeAttrs attrs list`

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

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

evaluates to {y = 2;}.

`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 integers *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.

`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`

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 5.1, “Nix expression for GNU Hello \(default.nix\)”](#) and [Example 5.2, “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://nix.cs.uu.nl/dist/tarballs/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  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 5.2.1, “Values”](#)), so the result of the expression `configFile` (i.e., a path like `/nix/store/m7p7jfn445k...-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`.

`builtins.toPath s`





Convert the string value `s` into a path value. The string `s` must represent an absolute path (i.e., must start with `/`). The path need not exist. The resulting path is canonicalised, e.g., `builtins.toPath "///foo/xyzz/../../bar/"` returns `/foo/bar`.

`toString e`

Convert the expression `e` to a string. `e` can be a string (in which case `toString` is a no-op) or a path (e.g., `toString /foo/bar` yields `"/foo/bar"`).

`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 5.6, “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 attribute sets containing the path and war of the servlet (). 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 () is applied to it () to generate the XML configuration file for the Jetty server. The XML representation produced from  by `toXML` is shown in [Example 5.7, “XML representation produced by toXML”](#).

Note that [Example 5.6, “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 5.6. 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.xml" 
    "<?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 5.7. 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.

5.3. The standard environment

The standard environment is used by passing it as an input called `stdenv` to the derivation, and then doing

```
source $stdenv/setup
```

at the top of the builder.

Apart from adding the aforementioned commands to the `PATH`, `setup` also does the following:

- All input packages specified in the `buildInputs` environment variable have their `/bin` subdirectory added to `PATH`, their `/include` subdirectory added to the C/C++ header file search path, and their `/lib` subdirectory added to the linker search path. This can be extended. For instance, when the **pkgconfig** package is used, the subdirectory `/lib/pkgconfig` of each input is added to the `PKG_CONFIG_PATH`

environment variable.

- The environment variable `NIX_CFLAGS_STRIP` is set so that the compiler strips debug information from object files. This can be disabled by setting `NIX_STRIP_DEBUG` to `0`.

The setup script also exports a function called `genericBuild` that knows how to build typical Autoconf-style packages. It can be customised to perform builds for any type of package. It is advisable to use `genericBuild` since it provides facilities that are almost always useful such as unpacking of sources, patching of sources, nested logging, etc.

The definitive, up-to-date documentation of the generic builder is the source itself, which resides in `pkgs/stdenv/generic/setup.sh`.

5.3.1. Customising the generic builder

The operation of the generic builder can be modified in many places by setting certain variables. These *hook variables* are typically set to the name of some shell function defined by you. For instance, to perform some additional steps after **make install** you would set the `postInstall` variable:

```
postInstall=myPostInstall

myPostInstall() {
    mkdir $out/share/extra
    cp extrafiles/* $out/share/extra
}
```

5.3.2. Debugging failed builds

At the beginning of each phase, the set of all shell variables is written to the file `env-vars` at the top-level build directory. This is useful for debugging: it allows you to recreate the environment in which a build was performed. For instance, if a build fails, then assuming you used the `-K` flag, you can go to the output directory and “switch” to the environment of the builder:

```
$ nix-build -K ./foo.nix
... fails, keeping build directory `/tmp/nix-1234-0'

$ cd /tmp/nix-1234-0

$ source env-vars

(edit some files...)

$ make

(execution continues with the same GCC, make, etc.)
```

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

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

[6] 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.

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

[8] Actually, Nix detects infinite recursion in this case and aborts (“infinite recursion encountered”).

[9] 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.

Chapter 6. Setting up a Build Farm

Table of Contents

[6.1. Overview](#)

[6.2. Setting up distributed builds](#)

This chapter provides some sketchy information on how to set up a Nix-based build farm. Nix is particularly suited as a basis for a build farm, since:

- Nix supports distributed builds: a local Nix installation can forward Nix builds to other machines over the network. This allows multiple builds to be performed in parallel (thus improving performance), but more importantly, it allows Nix to perform multi-platform builds in a semi-transparent way. For instance, if you perform a build for a powerpc-darwin on an i686-linux machine, Nix can automatically forward the build to a powerpc-darwin machine, if available.
- The Nix expression language is ideal for describing build jobs, plus all their dependencies. For instance, if your package has some dependency, you don't have to manually install it on all the machines in the build farm; they will be built automatically.
- Proper release management requires that builds (if deployed) are traceable: it should be possible to figure out from exactly what sources they were built, in what configuration, etc.; and it should be possible to reproduce the build, if necessary. Nix makes this possible since Nix's hashing scheme uniquely identifies builds, and Nix expressions are self-contained.
- Nix will only rebuild things that have actually changed. For instance, if the sources of a package haven't changed between runs of the build farm, the package won't be rebuilt (unless it was garbage-collected). Also, dependencies typically don't change very often, so they only need to be built once.
- The results of a Nix build farm can be made available through a channel, so successful builds can be deployed to users immediately.

6.1. Overview

TODO

The sources of the Nix build farm are at <https://svn.nixos.org/repos/nix/release/trunk>.

6.2. Setting up distributed builds

You can enable distributed builds by setting the environment variable `NIX_BUILD_HOOK` to point to a program that Nix will call whenever it wants to build a derivation. The build hook (typically a shell or Perl script) can decline the build, in which Nix will perform it in the usual way if possible, or it can accept it, in which case it is responsible for somehow getting the inputs of the build to another machine, doing the build there, and getting the results back. The details of the build hook protocol are described in the documentation of the [NIX_BUILD_HOOK variable](#).

Example 6.1. Remote machine configuration: `remote-systems.conf`

```
nix@mcflurry.labs.cs.uu.nl powerpc-darwin /home/nix/.ssh/id_quarterpounder_auto 2
```

An example build hook can be found in the Nix build farm sources:

<https://svn.nixos.org/repos/nix/release/trunk/common/distributed/build-remote.pl>. It should be suitable for most purposes, with maybe some minor adjustments. It uses **ssh** and **rsync** to copy the build inputs and outputs and perform the remote build. You should define a list of available build machines and set the environment variable `REMOTE_SYSTEMS` to point to it. An example configuration is shown in [Example 6.1, “Remote machine configuration: remote-systems.conf”](#). Each line in the file specifies a machine, with the following bits of information:

1. The name of the remote machine, with optionally the user under which the remote build should be performed. This is actually passed as an argument to **ssh**, so it can be an alias defined in your `~/.ssh/config`.
2. The Nix platform type identifier, such as `powerpc-darwin`.
3. The SSH private key to be used to log in to the remote machine. Since builds should be non-interactive, this key should not have a passphrase!
4. The maximum “load” of the remote machine. This is just the maximum number of jobs that `build-remote.pl` will execute in parallel on the machine. Typically this should be equal to the number of CPUs.

You should also set up the environment variable `CURRENT_LOAD` to point at a file that `build-remote.pl` uses to remember how many jobs it is currently executing remotely. It doesn't look at the actual load on the remote machine, so if you have multiple instances of Nix running, they should use the same `CURRENT_LOAD` file^[10]. Maybe in the future `build-remote.pl` will look at the actual remote load. The load file should exist, so you should just create it as an empty file initially.

^[10] Although there are probably some race conditions in the script right now.

Appendix A. Command Reference

Table of Contents

[A.1. Common options](#)

[A.2. Common environment variables](#)

[A.3. Nix configuration file](#)

[A.4. Main commands](#)

[nix-env](#) — manipulate or query Nix user environments

[nix-instantiate](#) — instantiate store derivations from Nix expressions

[nix-store](#) — manipulate or query the Nix store

[A.5. Utilities](#)

[nix-build](#) — build a Nix expression

[nix-channel](#) — manage Nix channels

[nix-collect-garbage](#) — delete unreachable store paths

[nix-copy-closure](#) — copy a closure to or from a remote machine via SSH

[nix-hash](#) — compute the cryptographic hash of a path

[nix-install-package](#) — install a Nix Package file

[nix-prefetch-url](#) — copy a file from a URL into the store and print its MD5 hash

[nix-pull](#) — pull substitutes from a network cache
[nix-push](#) — push store paths onto a network cache
[nix-worker](#) — Nix multi-user support daemon

A.1. 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.

`--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`

Sets the maximum number of build jobs that Nix will perform in parallel to the specified number. The default is specified by the [build-max-jobs](#) configuration setting, which itself defaults to 1. A higher value

is useful on SMP systems or to exploit I/O latency.

`--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 [build-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 [build-max-silent-time](#) configuration setting. 0 means no time-out.

`--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).

`--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.

`--log-type type`

This option determines how the output written to standard error is formatted. Nix's diagnostic messages are typically *nested*. For instance, when tracing Nix expression evaluation (`nix-env -vvvvv`, messages from subexpressions are nested inside their parent expressions. Nix builder output is also often nested. For instance, the Nix Packages generic builder nests the various build tasks (unpack, configure, compile, etc.), and the GNU Make in `stdenv-linux` has been patched to provide nesting for recursive Make invocations.

`type` can be one of the following:

`pretty`

Pretty-print the output, indicating different nesting levels using spaces. This is the default.

escapes

Indicate nesting using escape codes that can be interpreted by the **nix-log2xml** tool in the Nix source distribution. The resulting XML file can be fed into the **log2html.xsl** stylesheet to create an HTML file that can be browsed interactively, using Javascript to expand and collapse parts of the output.

flat

Remove all nesting.

`--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 file `pkgs/top-level/all-packages.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`

In **nix-env**, **nix-instantiate** and **nix-build**, `--attr` allows you to select an attribute from the top-level Nix expression being evaluated. 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.

`--show-trace`

Causes Nix to print out a stack trace in case of Nix expression evaluation errors.

A.2. Common environment variables

Most Nix commands interpret the following environment variables:

`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/log/nix*).

`NIX_STATE_DIR`

Overrides the location of the Nix state directory (default *prefix/var/nix*).

`NIX_DB_DIR`

Overrides the location of the Nix database (default *\$NIX_STATE_DIR/db*, i.e., *prefix/var/nix/db*).

`NIX_CONF_DIR`

Overrides the location of the Nix configuration directory (default *prefix/etc/nix*).

`NIX_LOG_TYPE`

Equivalent to the [--log-type option](#).

`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_BUILD_HOOK`

Specifies the location of the *build hook*, which is a program (typically some script) that Nix will call whenever it wants to build a derivation. This is used to implement distributed builds (see [Section 6.2, “Setting up distributed builds”](#)). The protocol by which the calling Nix process and the build hook communicate is as follows.

The build hook is called with the following command-line arguments:

1. A boolean value 0 or 1 specifying whether Nix can locally execute more builds, as per the [--max-jobs option](#). The purpose of this argument is to allow the hook to not have to maintain bookkeeping for the local machine.

2. The Nix platform identifier for the local machine (e.g., `i686-linux`).
3. The Nix platform identifier for the derivation, i.e., its [system attribute](#).
4. The store path of the derivation.

On the basis of this information, and whatever persistent state the build hook keeps about other machines and their current load, it has to decide what to do with the build. It should print out on standard error one of the following responses (terminated by a newline, `"\n"`):

`# decline`

The build hook is not willing or able to perform the build; the calling Nix process should do the build itself, if possible.

`# postpone`

The build hook cannot perform the build now, but can do so in the future (e.g., because all available build slots on remote machines are in use). The calling Nix process should postpone this build until at least one currently running build has terminated.

`# accept`

The build hook has accepted the build.

After sending `# accept`, the hook should read one line from standard input, which will be the string `okay`. It can then proceed with the build. Before sending `okay`, Nix will store in the hook's current directory a number of text files that contain information about the derivation:

`inputs`

The set of store paths that are inputs to the build process (one per line). These have to be copied *to* the remote machine (in addition to the store derivation itself).

`outputs`

The set of store paths that are outputs of the derivation (one per line). These have to be copied *from* the remote machine if the build succeeds.

`references`

The reference graph of the inputs, in the format accepted by the command **`nix-store --register-validity`**. It is necessary to run this command on the remote machine after copying the inputs to inform Nix on the remote machine that the inputs are valid paths.

The hook should copy the inputs to the remote machine, register the validity of the inputs, perform the remote build, and copy the outputs back to the local machine. An exit code other than `0` indicates that the hook has failed. An exit code equal to `100` means that the remote build failed (as opposed to, e.g., a network error).

`NIX_REMOTE`

This variable should be set to `daemon` if you want to use the Nix daemon to executed Nix operations, which is necessary in [multi-user Nix installations](#). Otherwise, it should be left unset.

`NIX_OTHER_STORES`

This variable contains the paths of remote Nix installations from which paths can be copied, separated by colons. See [Section 4.6, "Sharing packages between machines"](#) for details. Each path should be the `/nix` directory of a remote Nix installation (i.e., not the `/nix/store` directory). The paths are subject to

globbing, so you can set it so something like `/var/run/nix/remote-stores/*/nix` and mount multiple remote filesystems in `/var/run/nix/remote-stores`.

Note that if you're building through the [Nix daemon](#), the only setting for this variable that matters is the one that the **nix-worker** process uses. So if you want to change it, you have to restart the daemon.

A.3. Nix configuration file

A number of persistent settings of Nix are stored in the file `prefix/etc/nix/nix.conf`. This file is a list of *name* = *value* pairs, one per line. Comments start with a `#` character. An example configuration file is shown in [Example A.1, “Nix configuration file”](#).

Example A.1. Nix configuration file

```
gc-keep-outputs = true      # Nice for developers
gc-keep-derivations = true  # Idem
env-keep-derivations = false
```

The following variables are currently available:

`gc-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`.

`gc-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 `gc-keep-outputs` is also turned on).

`env-keep-derivations`

If `false` (default), derivations are not stored in Nix user environments. That is, the derivation 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 `gc-keep-outputs`.

The difference between this option and `gc-keep-derivations` is that this one is “sticky”: it applies to any user environment created while this option was enabled, while `gc-keep-derivations` only applies at the moment the garbage collector is run.

`build-max-jobs`

This option defines the maximum number of jobs that Nix will try to build in parallel. The default is 1. You should generally set it to the number of CPUs in your system (e.g., 2 on a Athlon 64 X2). It can be

overridden using the [--max-jobs](#) (-j) command line switch.

build-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.

build-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 a 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.

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`, or the uid that owns the setuid **nix-worker** program if `NIX_REMOTE` is `slave`). Obviously, this should not be used in multi-user settings with untrusted users.

build-use-chroot

If set to `true`, builds will be performed in a *chroot environment*, i.e., the build will be isolated from the normal file system hierarchy and will only see the Nix store, the temporary build directory, and the directories configured with the [build-chroot-dirs option](#) (such as `/proc` and `/dev`). This is useful to prevent undeclared dependencies on files in directories such as `/usr/bin`.

The use of a chroot requires that Nix is run as root (but you can still use the [“build users” feature](#) to perform builds under different users than root). Currently, chroot builds only work on Linux because Nix uses “bind mounts” to make the Nix store and other directories available inside the chroot.

build-chroot-dirs

When builds are performed in a chroot environment, Nix will mount (using **mount --bind** on Linux) some directories from the normal file system hierarchy inside the chroot. These are the Nix store, the

temporary build directory (usually `/tmp/nix-pid-number`) and the directories listed here. The default is `dev /proc`. Files in `/dev` (such as `/dev/null`) are needed by many builds, and some files in `/proc` may also be needed occasionally.

The value used on NixOS is

```
build-use-chroot = /dev /proc /bin
```

to make the `/bin/sh` symlink available (which is still needed by many builders).

system

This option specifies the canonical Nix system name of the current installation, such as `i686-linux` or `powerpc-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 `powerpc-darwin` and `i686-darwin`.

It defaults to the canonical Nix system name detected by `configure` at build time.

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 `false`.

A.4. Main commands

Name

`nix-env` — manipulate or query Nix user environments

Synopsis

```
nix-env [--help] [--version] [--verbose...] [-v...] [--no-build-output] [-Q] [ { --max-jobs | -j } number ] [ [--cores] number ] [ [--max-silent-time] number ] [--keep-going] [-k] [--keep-failed] [-K] [--fallback] [--readonly-mode] [--log-type type] [--show-trace]
[--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.

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 [Section A.1, “Common options”](#).

`--file, -f`

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`.

`--profile, -p`

Specifies the profile to be used by those operations that operate on a profile (designated below as the *active profile*). A profile is sequence of user environments called *generations*, one of which is the *current generation*. The default profile is the target of the symbolic link `~/.nix-profile` (see below).

`--dry-run`

For the `--install`, `--upgrade`, `--uninstall`, `--switch-generation` 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` only include derivations matching the current platform. This option allows you to use derivations for the specified platform *system*. The special value `*` causes derivations for any platform to be included.

Files

`~/.nix-defexpr`

A directory that contains 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.

The Nix expressions in this directory are combined into a single attribute set, with each file as an attribute that has the name of the file. Thus, if `~/.nix-defexpr` contains two files, `foo` and `bar`, then the default Nix expression will essentially be

```
{
  foo = import ~/.nix-defexpr/foo;
  bar = import ~/.nix-defexpr/bar;
}
```

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] 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 -qaA '*'`.
- 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 a 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.

Flags

`--prebuild-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.

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 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 -i 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 attribute 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 pkgs/top-level/all-packages.nix -i f-spot --dry-run
(dry run; not doing anything)
installing `f-spot-0.0.10'
the following derivations will be built:
/nix/store/0g63jv9aagwbpci4nnzs2dkxqz84kdja-libgnomeprintui-2.12.1.tar.bz2.drv
/nix/store/0gfarvxq6sannsdw8a1ir40j1ys2mqb4-ORBit2-2.14.2.tar.bz2.drv
/nix/store/0i9gs5zc04668qiy60ga2rc16abkj7g8-sqlite-2.8.17.drv
...
the following paths will be substituted:
/nix/store/8zbipvm4gp9jfqh9nnk1n3bary1a37gs-perl-XML-Parser-2.34
/nix/store/b8a2bg7gnyvvvjibp4axg9x1hzkw36c-mono-1.1.4
...
```

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 following by a letter. *x* is considered an upgrade

of *y* if their package names match, and the version of *y* is higher than 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-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/myy0y59q3ig70dgg37jqwg1j0rsapzsl-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 ] [ { --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 wildcard `*` shows all derivations.

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.

`--prebuild-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`.

Examples

```
$ nix-env -q '*' (show installed derivations)
```

```
bison-1.875c
docbook-xml-4.2
firefox-1.0.4
MPlayer-1.0pre7
ORBit2-2.8.3
...
```

```
$ nix-env -qa '*' (show available derivations)
```

```
firefox-1.0.7
GConf-2.4.0.1
MPlayer-1.0pre7
ORBit2-2.8.3
...
```

```
$ nix-env -qas '*' (show status of available derivations)
```

```
-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)
...
```

(show available derivations in the Nix expression foo.nix)

```
$ nix-env -f ./foo.nix -qa '*'
foo-1.2.3
```

```
$ nix-env -qc '*' (compare installed versions to what's available)
```

```
...
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)
...
```

(show info about a specific package, in XML)

```
$ nix-env -qa --xml --description firefox
<?xml version='1.0' encoding='utf-8'?>
<items>
  <item attrPath="0.0.firefoxWrapper"
    description="Mozilla Firefox - the browser, reloaded (with various plugins)"
    name="firefox-1.5.0.7" system="i686-linux" />
</items>
```

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, or the special value `old` to delete all non-current generations. Periodically deleting old generations is important to make garbage collection effective.

Examples

```
$ nix-env --delete-generations 3 4 8
```

```
$ 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 --rolback
```

error: no generation older than the current (91) exists

Name

nix-instantiate — instantiate store derivations from Nix expressions

Synopsis

```
nix-instantiate [--help] [--version] [--verbose...] [-v...] [--no-build-output] [-Q] [ { --max-jobs | -j }
number ] [ [--cores] number ] [ [--max-silent-time] number ] [--keep-going] [-k] [--keep-failed] [-K] [--
fallback] [--readonly-mode] [--log-type type] [--show-trace]
[--arg name value] [ { --attr | -A } attrPath ] [--add-root path] [--indirect] [ { --parse-only | --eval-only [-
strict] } ] [--xml] ] files...
```

Description

The command **nix-instantiate** generates [store derivations](#) from (high-level) Nix expressions. It loads and evaluates the Nix expressions in each of *files*. 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.

Most users and developers don't need to use this command (**nix-env** and **nix-build** perform store derivation instantiation from Nix expressions automatically). It is most commonly used for implementing new deployment policies.

See also [Section A.1, “Common options”](#) for a list of common options.

Options

--add-root *path*, --indirect

See the [corresponding options](#) in **nix-store**.

--parse-only

Just parse the input files, and print their abstract syntax trees on standard output in ATerm format.

--eval-only

Just parse and evaluate the input files, and print the resulting values on standard output. No instantiation of store derivations takes place.

--xml

When used with --parse-only and --eval-only, print the resulting expression 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](#).

--strict

When used with --eval-only, 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.

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
...
```

Parsing and evaluating Nix expressions:

```
$ echo '"foo" + "bar"' | nix-instantiate --parse-only -
OpPlus(Str("foo"),Str("bar"))

$ echo '"foo" + "bar"' | nix-instantiate --eval-only -
Str("foobar")

$ echo '"foo" + "bar"' | nix-instantiate --eval-only --xml -
<?xml version='1.0' encoding='utf-8'?>
<expr>
  <string value="foobar" />
</expr>
```

The difference between non-strict and strict evaluation:

```
$ echo 'rec { x = "foo"; y = x; }' | nix-instantiate --eval-only --xml -
...
  <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).

```
$ echo 'rec { x = "foo"; y = x; }' | nix-instantiate --eval-only --xml --strict -
...
  <attr name="x">
    <string value="foo" />
  </attr>
  <attr name="y">
    <string value="foo" />
  </attr>
...
```

Name

nix-store — manipulate or query the Nix store

Synopsis

```
nix-store [--help] [--version] [--verbose...] [-v...] [--no-build-output] [-Q] [ { --max-jobs | -j } number ] [ [-cores] number ] [ [--max-silent-time] number ] [--keep-going] [-k] [--keep-failed] [-K] [--fallback] [--readonly-mode] [--log-type type] [--show-trace]
[--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 [Section A.1, “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 4.3.1, “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-spo
```

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) substitutes, realisation fails.

The output path of each derivation is printed on standard output. (For non-derivations argument, the argument itself is printed.)

If the `--dry-run` option is used, then **nix-store** will print on standard error a description of what packages would be built or downloaded, and then quit.

Examples

This operation is typically used to build store derivations produced by [nix-instantiate](#):

```
$ nix-store -r $(nix-instantiate ./test.nix)
/nix/store/31axcgrlbfsczmffff1gyj1bf62hvkby2-aterm-2.3.1
```

This is essentially what [nix-build](#) does.

Operation `--gc`

Synopsis

```
nix-store --gc [ --print-roots | --print-live | --print-dead | --delete ] [ --max-freed bytes ] [ --max-links nrlinks ]
```

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 4.3.1, “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.

`--max-links nrlinks`

Keep deleting paths until the hard link count on `/nix/store` is less than *nrlinks*, then stop. This is useful for very large Nix stores on filesystems with a 32000 subdirectories limit (like ext3).

The behaviour of the collector is also influenced by the [gc-keep-outputs](#) and [gc-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/kq82idx6g0nyzsp2s14gfsc38npai71f-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/zq0h41l75v1b4z45kzgjjmsjxvcv1qk7-mesa-6.4
0 bytes freed (0.00 MiB)
error: cannot delete path `/nix/store/zq0h41l75v1b4z45kzgjjmsjxvcv1qk7-mesa-6.4' since it is still al
```

Operation `--query`

Synopsis

```
nix-store { --query | -q } { --outputs | --requisites | -R | --references | --referrers | --referrers-closure |
--deriver | --deriver | --graph | --tree | --binding name | --hash | --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 query 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`

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.

`--binding 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*. Since the hash is stored in the Nix database, this is a fast operation.

`--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/5mbglq5ldqld8sj57273aljwkvfj22mc-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/07a2bzxmwz5hp58nf03pahrv2ygwgs3-gcc-wrapper.sh
/nix/store/0ma7c9wsbaxahwwl04gbw3fcd806ski4-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/d8afh10z72n8l1cr5w42366abiblg54-builder.sh
+---/nix/store/fmzxpjx2lh849ph0l36snfj9zdibw67-bash-3.0.drv
|   +---/nix/store/570hmx3v57605c9yfvvyh0nnb8k8-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/5mbglq5ldqld8sj57273aljwkvfj22mc-subversion-1.1.4
/nix/store/dpmvp969yhdqs7lm2r1a3gng7pyq6vy4-subversion-1.1.3
/nix/store/151240xqsgg8a7yrbqdx1rfzyv6l26fx-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-libgnumprintui-2.8.2
/nix/store/1513yi0d45prm7a82pcrkndh6nzmxa-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 --verify

Synopsis

```
nix-store --verify [--check-contents]
```

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.

There is one option:

`--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.

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
```

For an example of how `--export` and `--import` can be used, see the source of the [nix-copy-closure](#) command.

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/qhgx7l2f1kmwihc9bnxs7rc159hsxnf3-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/p8n1jpqs27mgkpw07pb5269717nzf5f8-ktorrent-2.2.1.tar.gz
ktorrent-2.2.1/
ktorrent-2.2.1/NEWS
...
```

Operation --dump-db

Synopsis

```
nix-store --dump-db
```

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.

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.

A.5. Utilities

Name

nix-build — build a Nix expression

Synopsis

```
nix-build [--help] [--version] [--verbose...] [-v...] [--no-build-output] [-Q] [ { --max-jobs | -j } number ] [ [-cores] number ] [ [--max-silent-time] number ] [--keep-going] [-k] [--keep-failed] [-K] [--fallback] [--readonly-mode] [--log-type type] [--show-trace]
[--arg name value] [--argstr name value] [ { --attr | -A } attrPath ] [--add-drv-link] [--drv-link drvLink] [--no-out-link] [ { --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.

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

See also [Section A.1, “Common options”](#). All options not listed here are passed to **nix-store --realise**, except for **--arg** and **--attr / -A** which are passed to **nix-instantiate**.

--add-drv-link

Add a symlink in the current directory to the store derivation produced by **nix-instantiate**. The symlink is called *derivation* (which is numbered in the case of multiple derivations). The derivation is a root of the garbage collector until the symlink is deleted or renamed.

--drv-link *drvLink*

Change the name of the symlink to the derivation created when **--add-drv-link** is used from *derivation* to *drvLink*.

--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**.

--out-link / -o *outLink*

Change the name of the symlink to the output path created unless **--no-out-link** is used from *result* to *outLink*.

Examples

```
$ nix-build pkgs/top-level/all-packages.nix -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
```

Name

nix-channel — manage Nix channels

Synopsis

```
nix-channel { --add url | --remove url | --list | --update }
```

Description

A Nix channel is 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 that contains a set of Nix expressions, as well as a **nix-push** manifest. See also [Section 4.4, “Channels”](#).

This command has the following operations:

`--add url`

Adds *url* to the list of subscribed channels.

`--remove url`

Removes *url* from the list of subscribed channels.

`--list`

Prints the URLs of all subscribed channels on standard output.

`--update`

Downloads the Nix expressions of all subscribed channels, makes them the default for **nix-env** operations (by symlinking them in the directory `~/.nix-defexpr`), and performs a **nix-pull** on the manifests of all channels to make pre-built binaries available.

Note that `--add` and `--remove` do not automatically perform an update.

The list of subscribed channels is stored in `~/.nix-channels`.

A channel consists of two elements: a bzipipped Tar archive containing the Nix expressions, and a manifest created by **nix-push**. These must be stored under `url/nixexprs.tar.bz2` and `url/MANIFEST`, respectively.

Name

nix-collect-garbage — delete unreachable store paths

Synopsis

```
nix-collect-garbage [--delete-old] [-d] [ --print-roots | --print-live | --print-dead | --delete ]
```

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 an additional option `-d` (`--delete-old`) that 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.

Example

To delete from the Nix store everything that is not used by the current generations of each profile, do

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

Name

`nix-copy-closure` — copy a closure to or from a remote machine via SSH

Synopsis

```
nix-copy-closure [ --to | --from ] [--sign] [--gzip] [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 *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.
- `--sign`

Let the sending machine cryptographically sign the dump of each path with the key in `/nix/etc/nix/signing-key.sec`. If the user on the target machine does not have direct access to the Nix store (i.e., if the target machine has a multi-user Nix installation), then the target machine will check the dump against `/nix/etc/nix/signing-key.pub` before unpacking it in its Nix store. This allows secure sharing of store paths between untrusted users on two machines, provided that there is a trust relation between the Nix installations on both machines (namely, they have matching public/secret keys).

`--gzip`

Compress the dump of each path with **gzip** before sending it.

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:

```
$ nix-copy-closure --from alice@itchy.labs \  
  /nix/store/0dj0503hjxy5mbwlaflv1rsbdiyx1gkdy-subversion-1.4.4  
$ nix-env -i /nix/store/0dj0503hjxy5mbwlaflv1rsbdiyx1gkdy-subversion-1.4.4
```

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.

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*

Specify a cryptographic hash, 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 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/
nvd61k9nalji1zl9rrdfmsmvyjqpzg4

$ 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
nvd61k9nalji1zl9rrdfmsmvyjqpzg4

$ nix-hash --type sha1 --to-base16 nvd61k9nalji1zl9rrdfmsmvyjqpzg4
e4fd8ba5f7bbeaea5ace89fe10255536cd60dab6
```

Name

nix-install-package — install a Nix Package file

Synopsis

```
nix-install-package [--non-interactive] [ { --profile | -p } path ]  
{ { --url url } | { file } }
```

Description

The command **nix-install-package** interactively installs a Nix Package file (*.nixpkg), which is a small file that contains a store path to be installed along with the URL of a [nix-push manifest](#). The Nix Package file is either *file*, or automatically downloaded from *url* if the --url switch is used.

nix-install-package is used in [one-click installs](#) to download and install pre-built binary packages with all necessary dependencies. **nix-install-package** is intended to be associated with the MIME type application/nix-package in a web browser so that it is invoked automatically when you click on *.nixpkg files. When invoked, it restarts itself in a terminal window (since otherwise it would be invisible when run from a browser), asks the user to confirm whether to install the package, and if so downloads and installs the package into the user's current profile.

To obtain a window, **nix-install-package** tries to restart itself with **xterm**, **konsole** and **gnome-terminal**.

Options

--non-interactive

Do not open a new terminal window and do not ask for confirmation.

--profile, -p

Install the package into the specified profile rather than the user's current profile.

Examples

To install subversion-1.4.0.nixpkg into the user's current profile, without any prompting:

```
$ nix-install-package --non-interactive subversion-1.4.0.nixpkg
```

To install the same package from some URL into a different profile:

```
$ nix-install-package --non-interactive -p /nix/var/nix/profiles/eelco \  
  --url http://nix.cs.uu.nl/dist/nix/nixpkgs-0.10pre6622/pkgs/subversion-1.4.0-i686-linux.nixpkg
```

Format of nixpkg files

A Nix Package file consists of a single line with the following format:

```
NIXPKG1 manifestURL name system drvPath outPath
```

The elements are as follows:

```
NIXPKG1
```


The version of the Nix Package file.

manifestURL

The manifest to be pulled by **nix-pull**. The manifest must contain *outPath*.

name

The symbolic name and version of the package.

system

The platform identifier of the platform for which this binary package is intended.

drvPath

The path in the Nix store of the derivation from which *outPath* was built. Not currently used.

outPath

The path in the Nix store of the package. After **nix-install-package** has obtained the manifest from *manifestURL*, it performs a `nix-env -i outPath` to install the binary package.

An example follows:

```
NIXPKG1 http://.../nixpkgs-0.10pre6622/MANIFEST subversion-1.4.0 i686-darwin \  
/nix/store/4kh60jkg...-subversion-1.4.0.drv \  
/nix/store/nkw7wpgb...-subversion-1.4.0
```

(The line breaks (\) are for presentation purposes and not part of the actual file.)

Name

`nix-prefetch-url` — copy a file from a URL into the store and print its MD5 hash

Synopsis

```
nix-prefetch-url 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.

The environment variable `NIX_HASH_ALGO` specifies which hash algorithm to use. It can be either `md5`, `sha1`, or `sha256`. The default is `sha256`.

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 environment variable `PRINT_PATH` is set, the path of the downloaded file in the Nix store is also printed.

Examples

```
$ nix-prefetch-url ftp://ftp.nluug.nl/pub/gnu/make/make-3.80.tar.bz2
0bbd1df101bc0294d440471e50feca71
```

```
$ PRINT_PATH=1 nix-prefetch-url ftp://ftp.nluug.nl/pub/gnu/make/make-3.80.tar.bz2
0bbd1df101bc0294d440471e50feca71
/nix/store/wvyz8ifdn7wyz1p3pqyn0ra45ka2l492-make-3.80.tar.bz2
```

Name

`nix-pull` — pull substitutes from a network cache

Synopsis

`nix-pull` *url*

Description

The command **nix-pull** obtains a list of pre-built store paths from the URL *url*, and for each of these store paths, registers a substitute derivation that downloads and unpacks it into the Nix store. This is used to speed up installations: if you attempt to install something that has already been built and stored into the network cache, Nix can transparently re-use the pre-built store paths.

The file at *url* must be compatible with the files created by *nix-push*.

Examples

```
$ nix-pull http://nix.cs.uu.nl/dist/nix/nixpkgs-0.5pre753/MANIFEST
```

Name

`nix-push` — push store paths onto a network cache

Synopsis

`nix-push` { { *archivesPutURL archivesGetURL manifestPutURL* } | { `--copy` *archivesDir manifestFile* } } *paths...*

Description

The command **nix-push** builds a set of store paths (if necessary), and then packages and uploads all store paths in the resulting closures to a server. A network cache thus populated can subsequently be used to speed up software deployment on other machines using the **nix-pull** command.

nix-push performs the following actions.

1. Each path in *paths* is realised (using [nix-store --realise](#)).
2. All paths in the closure of the store expressions stored in *paths* are determined (using `nix-store --query --requisites --include-outputs`). It should be noted that since the `--include-outputs` flag is used, you get a combined source/binary distribution.
3. All store paths determined in the previous step are packaged and compressed into a **bzipped** NAR archive (extension `.nar.bz2`).
4. A *manifest* is created that contains information on the store paths, their eventual URLs in the cache, and cryptographic hashes of the contents of the NAR archives.
5. Each store path is uploaded to the remote directory specified by *archivesPutURL*. HTTP PUT requests are used to do this. However, before a file *x* is uploaded to *archivesPutURL/x*, **nix-push** first determines whether this upload is unnecessary by issuing a HTTP HEAD request on *archivesGetURL/x*. This allows a cache to be shared between many partially overlapping **nix-push** invocations. (We use two URLs because the upload URL typically refers to a CGI script, while the download URL just refers to a file system directory on the server.)
6. The manifest is uploaded using an HTTP PUT request to *manifestPutURL*. The corresponding URL to download the manifest can then be used by **nix-pull**.

Examples

To upload files there typically is some CGI script on the server side. This script should be protected with a password. The following example uploads the store paths resulting from building the Nix expressions in `foo.nix`, passing appropriate authentication information:

```
$ nix-push \
  http://foo@bar:server.domain/cgi-bin/upload.pl/cache \
  http://server.domain/cache \
  http://foo@bar:server.domain/cgi-bin/upload.pl/MANIFEST \
  $(nix-instantiate foo.nix)
```

This will push both sources and binaries (and any build-time dependencies used in the build, such as compilers).

If we just want to push binaries, not sources and build-time dependencies, we can do:

```
$ nix-push urls $(nix-store -r $(nix-instantiate foo.nix))
```

Name

nix-worker — Nix multi-user support daemon

Synopsis

```
nix-worker --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.

Appendix B. Troubleshooting

Table of Contents

[B.1. Collisions in `nix-env`](#)

[B.2. “Too many links” error in the Nix store](#)

This section provides solutions for some common problems. See the [Nix bug tracker](#) for a list of currently known issues.

B.1. Collisions in `nix-env`

Symptom: when installing or upgrading, you get an error message such as

```
$ nix-env -i docbook-xml
...
adding /nix/store/s5hyxgm62gk2...-docbook-xml-4.2
collision between `/nix/store/s5hyxgm62gk2...-docbook-xml-4.2/xml/dtd/docbook/calstblx.dtd'
  and `/nix/store/06h377hr4b33...-docbook-xml-4.3/xml/dtd/docbook/calstblx.dtd'
  at /nix/store/...-builder.pl line 62.
```

The cause is that two installed packages in the user environment have overlapping filenames (e.g., `xml/dtd/docbook/calstblx.dtd`). This usually happens when you accidentally try to install two versions of the same package. For instance, in the example above, the Nix Packages collection contains two versions of `docbook-xml`, so **`nix-env -i`** will try to install both. The default user environment builder has no way to resolve such conflicts, so it just gives up.

Solution: remove one of the offending packages from the user environment (if already installed) using **`nix-env -e`**, or specify exactly which version should be installed (e.g., `nix-env -i docbook-xml-4.2`).

Alternatively, you can modify the user environment builder script (in `prefix/share/nix/corepkgs/buildenv/builder.pl`) to implement some conflict resolution policy. E.g., the script could be modified to rename conflicting file names, or to pick one over the other.

B.2. “Too many links” error in the Nix store

Symptom: when building something, you get an error message such as

```
...
mkdir: cannot create directory `/nix/store/name': Too many links
```

This is usually because you have more than 32,000 subdirectories in `/nix/store`, as can be seen using **`ls -l`**:

```
$ ls -l /nix/store
drwxrwxrwt 32000 nix nix 4620288 Sep 8 15:08 store
```

The ext2 file system is limited to a inode link count of 32,000 (each subdirectory increasing the count by one). Furthermore, the `st_nlink` field of the `stat` system call is a 16-bit value.

This only happens on very large Nix installations (such as build machines).

Quick solution: run the garbage collector. You may want to use the `--max-links` option.

Real solution: put the Nix store on a file system that supports more than 32,000 subdirectories per directory, such as ReiserFS. (This doesn’t solve the `st_nlink` limit, but ReiserFS lies to the kernel by reporting a link count of 1 if it exceeds the limit.)

Appendix C. 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 normal the 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. This implies than an execution involving *P* potentially needs *Q* to be present. The *references* of a store path are the set of store paths to which it has a reference.

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 instance, if the store object at path *P* contains a reference to path *Q*, then *Q* is in the closure of *P*. 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.

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 [Section 4.2, “Profiles”](#).

profile

A symlink to the current [user environment](#) of a user, e.g., `/nix/var/nix/profiles/default`.

Appendix D. Nix Release Notes

Table of Contents

[D.1. Release 0.16 \(August 17, 2010\)](#)

[D.2. Release 0.15 \(March 17, 2010\)](#)

[D.3. Release 0.14 \(February 4, 2010\)](#)

[D.4. Release 0.13 \(November 5, 2009\)](#)

[D.5. Release 0.12 \(November 20, 2008\)](#)

[D.6. Release 0.11 \(December 31, 2007\)](#)

[D.7. Release 0.10.1 \(October 11, 2006\)](#)

[D.8. Release 0.10 \(October 6, 2006\)](#)

[D.9. Release 0.9.2 \(September 21, 2005\)](#)

[D.10. Release 0.9.1 \(September 20, 2005\)](#)

[D.11. Release 0.9 \(September 16, 2005\)](#)

[D.12. Release 0.8.1 \(April 13, 2005\)](#)

[D.13. Release 0.8 \(April 11, 2005\)](#)

[D.14. Release 0.7 \(January 12, 2005\)](#)

[D.15. Release 0.6 \(November 14, 2004\)](#)

[D.16. Release 0.5 and earlier](#)

D.1. Release 0.16 (August 17, 2010)

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.

D.2. Release 0.15 (March 17, 2010)

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.

D.3. Release 0.14 (February 4, 2010)

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.

D.4. Release 0.13 (November 5, 2009)

This is primarily a bug fix release. It has some new features:

- Syntactic sugar for writing nested attribute sets. Instead of

```
{
  foo = {
    bar = 123;
    xyzzz = true;
  };
  a = { b = { c = "d"; }; };
}
```

you can write

```
{
  foo.bar = 123;
  foo.xyzzz = 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.

D.5. Release 0.12 (November 20, 2008)

- 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.* derivs 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/129sbxk5n466zg6r1qmq1xjv9zymyy7-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/7h1kwCj29ip8vk26rhmx6bfjrxp0g4l-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.

D.6. Release 0.11 (December 31, 2007)

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`.

D.7. Release 0.10.1 (October 11, 2006)

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.

D.8. Release 0.10 (October 6, 2006)

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 slightly 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

```
--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 **lsuf** 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.
- **bsdifff** 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.

D.9. Release 0.9.2 (September 21, 2005)

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`

D.10. Release 0.9.1 (September 20, 2005)

This bug fix release addresses a problem with the ATerm library when the `--with-aterm` flag in **configure** was *not* used.

D.11. Release 0.9 (September 16, 2005)

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.

D.12. Release 0.8.1 (April 13, 2005)

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.

D.13. Release 0.8 (April 11, 2005)

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/csw87wag8bqlqk7ip1lbwypb14xainap-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/4b0jx7vq80l9aqcnkszxhymf1ffa5jd-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/8lz9yc6zgm0v1qmn2ipcpkj1mbi51vv-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 succesful 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.

D.14. Release 0.7 (January 12, 2005)

- 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.

D.15. Release 0.6 (November 14, 2004)

- 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 into 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.

D.16. Release 0.5 and earlier

Please refer to the Subversion commit log messages.