

NixOS

Andres Löh

joint work with Eelco Dolstra

Department of Information and Computing Sciences
Utrecht University

June 17, 2008

Introduction

- ► NixOS is a Linux distribution.
- ▶ It is based on the Nix package manager.
- ▶ Nix offers a functional domain-specific language to describe system components.
- ▶ In NixOS, programs/packages but also configurations and services are described by Nix expressions.
- Like in a pure functional language, system configurations cannot be updated destructively. Instead, new configurations can be built by evaluating new expressions.



Overview

Nix – a purely functional package manager

Imperative vs. functional

The Nix store

Nix expressions

NixOS

Nixpkgs

System configuration

Nix – a purely functional package manager





Package management is usually imperative

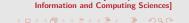
- ▶ Software is distributed in many components.
- ▶ Components may depend on each other.
- ▶ On one system, a certain selection of components is installed at one point in time.
- Components are installed into a common filesystem and usually find other components by looking in specific situations.
- ► The system configuration is like a mutable variable: new installations, upgrades, package removals destructively update the configuration: they overwrite files in the file system.



Consequences

Upgrading or changing one component can break other components:

- Consider a program (say GHC) that makes use of Perl, by calling /usr/bin/perl at run-time.
- Another program depends on a later version of Perl, so installing the program triggers a destructive upgrade of Perl.
- ▶ It is now not clear if GHC still works.



Other disadvantages

- Often incomplete dependency specifications (using version ranges only).
- ▶ Difficult to upgrade configuration files:
 - overwrite
 - keep old
 - merge in some way
- Difficult to install multiple variants of one component on a system at the same time.
- ▶ Difficult to reproduce a specific configuration.
- ▶ Difficult to recover from an inconsistent state.

Functional package management

- ► Components are described using Nix expressions.
- Evaluating Nix expressions corresponds to building one or several components.
- Built components are stored in the Nix store and immutable once built.
- ► The Nix store also serves as a cache: if the same component is required multiple times, it is built only once.



The Nix store

- ▶ Like a heap for components.
- Components are stored in isolation.
- ▶ Installing or upgrading components amounts to allocating new objects in the Nix store. The old components are unaffected and remain available.
- ▶ It is easy to install several variants of a component on the same system.



The Nix store in more detail

- ► The Nix store is one directory in the file system, usually /nix/store.
- ▶ Every entry in the store is a subdirectory. The subdirectory includes a cryptographic hash reflecting the identity of the component: the complete Nix expression including all dependencies determines the hash.

Example

/nix/store/rb4sqlpdnlcinsqr7pfbisdlpngc5jax-ghc-6.8.2



Advantages of using cryptographic hashes

- ► Since the hash is based on everything that determines the identity of a component, we get different components in isolation, but also automatic maximal sharing of components.
- Hashes are much more finegrained than a name and a version number. /nix/store/q5cq4g7rpm4vgk49qkmvlks4ijrz90n6-ghc-6.8.2
 - /nix/store/rb4sqlpdnlcinsqr7pfbisdlpngc5jax-ghc-6.8.2
- ► The hashes are difficult to guess, so it is difficult to find components except with the help of Nix.
- ► We can use hashes to check for run-time dependencies, by scanning the store entry for hash occurrences.





Description of a simple component

```
{stdenv, fetchurl, pkgconfig, libXaw, libXt}:
stdenv.mkDerivation {
  name = "xmessage-1.0.2";
  src = fetchurl {
    url = http://.../X11R7.3/.../xmessage-1.0.2.tar.bz2;
    sha256 = "1hy3n227iyrm323hnrdld8knj9h82fz6...";
  };
  buildInputs = [pkgconfig libXaw libXt];
}
```

Easy to define your own abstractions

```
{cabal, X11, xmessage}:
cabal.mkDerivation (self : {
  pname = "xmonad";
  version = "0.7";
  sha256 = "d5ee338eb6d0680082e20eaafa0b23b3...";
  extraBuildInputs = [X11];
  meta = {
    description = "xmonad is a tiling window manager for X";
  };
  preConfigure = ''
    substituteInPlace XMonad/Core.hs --replace \
      "xmessage", "${xmessage}/bin/xmessage,
  ,,;
})
```



Combining packages

```
rec {
    ...
    xmonad = import ../applications/window-managers/xmonad {
     inherit stdenv fetchurl ghc X11;
     inherit (xlibs) xmessage;
    };
    ...
}
```

Nix expressions

- ▶ Dynamically typed, pure, lazy functional language.
- ► Convenience features: URI literals, path literals, multi-line string literals with interpolation.
- Attribute sets (records).
- No real module system, but with import path; ... construct.



Derivations

▶ The functions

```
stdenv.mkDerivation cabal.mkDerivation
```

are wrappers around the built-in function derivation.

► The built-in function **derivation** takes an attribute set describing a build action:

```
{ system = ...; # the architecture of the system
  name = ...; # name of the package
  builder = ...; # shell script to perform the build
    ... # augmenting the build environment
}
```

Evaluating derivations

- Evaluating the derivation
 - evaluates all the store paths that occur in the input attribute set (build-time dependencies)
 - computes the store location for the derivation
 - builds the package in a restricted build environment (if it does not yet exist)
 - returns the store location of the built package
- ► Evaluating a derivation is the only way to get hold of a store location.



The build environment

- ► Additional attributes passed to derivation are added to the build environment as environment variables.
- ▶ In particular, other derivations can be passed to make their store paths known.
- ► The builder (a shell script) can use this information to facilitate the build process:
 - binaries from store paths that are dependencies are added to the search path
 - libraries from store paths that are dependencies are added to the linker search path
 - **>** . . .
- ► The builder of a component can only write to the temporary build environment and to the designated output path of the Nix store.

Binary distribution

- ▶ Store paths are unique even across different systems.
- Downloading a pre-built binary instead of building a component locally is a simple optimization.
- ▶ It is possible to apply binary patching techniques in order to reduce the size of downloads necessary.



Profiles

- ▶ A profile provides a view on a set of store entries. A specific set of components can be symlinked into a directory tree so that it can easily be used.
- ► There can be a system-wide profile with defaults for all users, but every user can have a personal profile.
- ▶ A profile consists of a history of **user environments**.
- ► Each user environment is an immutable store entry.
- Installing a package as a user builds an updated user environment and exports it as a new generation of the user's profile.
- Rollbacks are easy by switching to an older generation of a profile.

Installing components

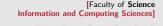
- nix-env -i ghc installs the latest version of ghc defined in the default Nix expression into the default profile.
- nix-env -i ghc-6.8.1 selects a specific version.
- nix-env -u ghc upgrades ghc in the default profile to the latest version.
- nix-env -e ghc removes ghc from the default profile (but not from the store!).

Garbage collection

- ► Garbage collection removes unused entries from the Nix store.
- ▶ Only run on explicit request:

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

- ► Conservative garbage collection via hashes.
- Can be used to remove build-time dependencies that are not run-time dependencies.
- ▶ Can affect the ability to roll back to previous versions.



NixOS





From Nix to NixOS

- ► The Nix package manager can in theory be used together with multiple operating systems (various Linux distributions, various BSD distributions, MacOS, Windows with Cygwin, . . .)
- NixOS is a full Linux distribution using Nix not just for the software, but also for the system configuration.
- ► Things built by Nix expressions in NixOS include:
 - software
 - the kernel and kernel modules
 - configuration files
 - services





Nixpkgs – the Nix packages collection

- ▶ Nix expressions for more than 1300 packages and growing.
- ► GCC, X11, KDE, Gnome, Apache, PostgreSQL, GHC, OCaml, . . .
- ▶ Also expressions for some closed-source software such as Acrobat Reader.
- Selection slightly biased on the needs of current contributors.
- Provides functions for each package plus predefined combinations of current versions that are being tested on a build farm.
- Selected binaries (depending on license issues, relevance, build time) are automatically made available for binary distribution.

System configuration

- ▶ One Nix expression, located in /etc/nixos/nixos/default.nix describes an attribute set with an attribute system. Evaluating that attribute (re-)builds the whole system configuration.
- ► The Nix expression imports a configuration file /etc/nixos/configuration.nix that allows to adjust the configuration in several ways.



Example configuration

```
boot = {
  grubDevice = "/dev/sda";
fileSystems = [
  { mountPoint = "/";
    device = "/dev/sda1";
services = {
  sshd = {
    enable = true:
    forwardX11 = true;
  };
  xserver = {
    enable = true;
    videoDriver = "vesa";
    sessionType = "xterm";
    windowManager = "xmonad";
};
```

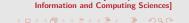
System components

- ► The Linux kernel is built including selected external modules, and is a normal store entry.
- On a kernel upgrade, external modules are automatically rebuilt.
- ▶ The initial ramdisk is also built in the Nix store.
- ► Services (X server, dhcp client, sshd) are built as upstart services and linked to /etc/event.d.



Configuration files

- ▶ Most configuration files are package-specific and generated in the Nix store (for instance, the sshd configuration is not contained in /etc).
- ▶ Other configuration files are used by multiple packages (/etc/hosts) and are therefore symlinked to /etc.
- ➤ A select few configuration files (/etc/passwd) are not maintained via the Nix store, but Nix ensures the presence of certain entries.



Changing the configuration

- A configuration contains an activation script that starts/stops services, creates links in /etc, ensures the presence of user accounts, ...
- ▶ To change the system configuration, one must
 - edit configuration.nix
 - ▶ call nixos-rebuild switch
- ► Then,
 - attribute system of the top-level Nix expression is evaluated
 - the activation script is run
 - the resulting derivation is installed in a special system profile
 - ▶ the boot menu is regenerated from the system profile

Conclusions

- ▶ It works too much state is just making things complicated.
- ▶ Purity and laziness are essential for the Nix expression language.
- ► Experiments show that the current way of enforcing purity is sufficient.



Future

- Enforce purity more strictly.
- ► Static nominal type system:
 - generating GUIs from type information
 - QuickChecking Nix expressions
- Why restrict Nix expressions to describe only one system? Networks!