Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

NixOS

Andres Löh und Eelco Dolstra

Department of Information and Computing Sciences
Utrecht University

July 18, 2008

Überblick

Nix – ein funktionaler Paketmanager

- Imperativ vs. funktional
- Der Nix-Store
- Nix-Ausdrücke

NixOS

- Nixpkgs
- Systemkonfiguration

Nix – ein funktionaler Paketmanager

Klassisches (imperatives) Paketmanagement

- Viele Einzelpakete.
- Komplexe Abhängigkeiten.
- Zu einem bestimmten Zeitpunkt sind auf einem System selektierte Pakete installiert und können verwendet werden.
- Aufgabe des Paketmanagers: Konsistenz gewährleisten.
- Alle Benutzer verwenden die gleiche Konfiguration.
- Gemeinsames Dateisystem.
- Auffinden von Abhängigkeiten durch Suchen in Standard-Pfaden.

Warum imperativ?

Die Systemkonfiguration ist wie eine (veränderbare) Variable:

- Installieren, Updaten oder Entfernen von Paketen verändert die Konfiguration unwiderruflich.
- Paket-spezifische Dateien werden bei Updates gelöscht oder überschrieben.

Die Folgen des imperativen Modells

Das Ändern eines Pakets kann andere Pakete beeinträchtigen.

Beispiel:

- Webbrowser “Firefox” hängt zur Laufzeit von der Verschlüsselungsbibliothek “openssl” ab.
- Das Update des Versionskontrollsystems “subversion” erfordert eine neuere Version von openssl.
- Das Ersetzen der “openssl”-Bibliothek kann dazu führen, daß Firefox nicht mehr funktioniert.

Die Folgen des imperativen Modells

Das Ändern eines Pakets kann andere Pakete beeinträchtigen.

Beispiel:

- Webbrowser “Firefox” hängt zur Laufzeit von der Verschlüsselungsbibliothek “openssl” ab.
- Das Update des Versionskontrollsystems “subversion” erfordert eine neuere Version von openssl.
- Das Ersetzen der “openssl”-Bibliothek kann dazu führen, daß Firefox nicht mehr funktioniert.
- Evtl. ist der Paketmanager intelligent genug, auch eine neue Version von Firefox zu installieren.
- Firefox hat Erweiterungen, und nicht alle Erweiterungen sind notwendigerweise kompatibel mit der neuen Version.

Weitere Nachteile

- Oft können Abhängigkeiten nur unvollkommen spezifiziert werden (z.B. durch Versionsbereiche).
- Was ist zu tun mit Konfigurationsdateien?
  - Überschreiben mit neuer Version: Verlust alter Anpassungen.
  - Alte Version behalten: neue Optionen werden übersehen, evtl. ist auch das Format verändert.
  - Verschmelzen der alten mit der neuen Version, aber wie? Erfordert zumindest manuelles Eingreifen.
- Schwierig:
  - Mehrere Varianten desselben Paketes auf demselben System.
  - Eine bestimmte Konfiguration reproduzieren.
  - Ein inkonsistentes System retten.
  - Während eines Updates weiterarbeiten.

Funktionales Sprachen

Reine funktionale Programmiersprachen haben die folgenden Eigenschaft:

- Funktionen sind Funktionen im mathematischen Sinn. Die Anwendung einer Funktion `f` auf ein Argument `x`

  ```
  f x
  ```

  hat **immer** dasselbe Resultat. Es gibt keine "Seiteneffekte".

Funktionales Sprachen

Reine funktionale Programmiersprachen haben die folgenden Eigenschaft:

- Funktionen sind Funktionen im mathematischen Sinn. Die Anwendung einer Funktion `f` auf ein Argument `x`

  ```
  f x
  ```

  hat **immer** dasselbe Resultat. Es gibt keine "Seiteneffekte".
- Insbesondere können Daten nicht destruktiv verändert werden. Es ist z.B. nicht möglich, in eine Liste ein neues Element einzufügen und die alte Liste zu ersetzen

  ```
  list := insert new list
  ```

  Statt dessen können wir nur eine neue Liste erzeugen

  ```
  list' = insert new list
  ```

  Beide Listen sind weiterhin verfügbar.

Funktionales Paketmanagement

- Pakete werden mittels Ausdrücken in der funktionalen Nix-Sprache beschriben.
- Das Auswerten eines Ausdrucks entspricht dem Bauen bzw. Bereitstellen eines Pakets.
- Pakete werden im "Nix-Store" abgelegt.
- Jedes Paket bekommt seinen eigenen, komplett abgeschotteten Bereich im Nix-Store.
- Pakete sind unveränderlich, sobald sie gebaut wurden.
- Der Nix-Store dient auch als Cache: wenn ein Ausdruck augewertet wird, der schon früher ausgewertet wurde, wird das entsprechende Paket nicht neu installiert, sondern der bereits vorhandene Eintrag im Nix-Store wiederverwendet.

Der Nix-Store

- Ähnelt einem Heap, der zur dynamischen Speicherallokation von Programmiersprachen verwendet wird. Aber statt Datenstrukturen liegen Softwarepakete auf dem Heap.
- Alle Pakete sind komplett isoliert voneinander.
- Installieren und Updaten von Paketen kann neue Einträge im Nix-Store erzeugen, aber beeinflußt nie die Einträge, die schon existieren.
- Es ist einfach, mehrere Varianten eines Paketes auf demselben System zu installieren.

Mehr über den Nix-Store

- Der Nix-Store befindet sich unterhalb eines festen Platzes im Dateisystem, `/nix/store`.
- Jeder Eintrag im Nix-Store bekommt ein eigenes Unterverzeichnis. Der Name des Verzeichnisses enthält einen Hashcode, in dessen Berechnung die komplette Beschreibung des Paketes inklusive aller Abhängigkeiten eingeht.

Example

```
/nix/store/rb4sqlpdnlcinsqr7pfbisdlpngc5jax-ghc-6.8.2
```

Vorteile (kryptographischer) Hashes

- Der Hash beinhaltet alles, was ein Paket beeinflussen könnte. Dadurch bekommen wir isolierte Pakete, aber auch die automatische Wiederverwendung bereits erstellter Pakete.
- Hashes sind viel detaillierter als nur ein Name und eine Versionsangabe.
  `/nix/store/q5cq4g7rpm4vgk49qkmvlks4ijrz90n6-ghc-6.8.2`
  `/nix/store/rb4sqlpdnlcinsqr7pfbisdlpngc5jax-ghc-6.8.2`
- Hashes in den Verzeichnisnamen bewirken, daß Programme nicht aus Versehen auf den Nix-Store zugreifen.
- Hashes können verwendet werden, um Pakete automatisch nach Laufzeitabhängigkeiten abzusuchen.

Eine einfache Paketbeschreibung

```
{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];
}
```

Neue Abstraktionen können definiert werden

```
{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"'
  '';
})o
```

Verbinden der einzelnen Pakete

```
rec {

  ...

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

  ...

}
```

Nix-Ausdrücke

- Dynamisch getypte, reine, “lazy” ausgewertete funktionale Sprache.
- Domänen-spezifische Features: URI-Literale, Pfadliterale, mehrzeilige Strings, die die Einrückung erhalten und eingebettete Nix-Ausdrücke erlauben.
- Attributmengen (Records).
- Kein Modulsystem, aber ein Konstrukt zum Import von Attributmengen aus anderen Dateien:
  `with import path ; ...`

Derivations

- Die Funktionen

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

  greifen auf die eingebaute Funktion **derivation** zurück.
- Die Funktion **derivation** is parametrisiert über eine Attributmenge, die eine Paketerstellung beschreibt:

```
{ system  = ...;  # Art des Systems
  name    = ...;  # Name des Pakets
  builder = ...;  # Shell-Skript
  ...             # weitere Attribute
}
```

Auswerten von derivations

- Beim Auswerten der Funktion `derivation`
  - werden alle Store-Einträge realisiert, die in der übergebenen Attributmenge auftauchen,
  - wird das Verzeichnis (mit Hash) berechnet, das für das neue Paket bestimmt ist
  - wird das Paket in einer klar spezifizierten Umgebung gebaut (falls es nicht bereits existiert)
  - wird der Verzeichnisname des Store-Eintrags zurückgegeben
- Das Auswerten einer derivation ist der einzige (vorgesehene) Weg, an den Verzeichnisnamen eines Store-Eintrags zu kommen.

Die Umgebung

- Zusätzliche Attribute werden als Umgebungsvariablen dem `builder`-Skript mitgegeben.
- Durch das Aufzählen anderer Derivations können deren Store-Verzeichnisse übergeben werden.
- Nicht jedes Paket benötigt ein eigenes `builder`-Skript. Der generische `builder`, der von `stdenv.mkDerivation` verwendet wird,
  - fügt ausführbare Dateien von spezifizierten abhängigen Paketen automatisch zum Suchpfad hinzu
  - fügt Bibliotheken von spezifizierten abhängigen Paketen automatisch zum Suchpfad des Linkers hinzu
  - ...
- Das `builder`-Skript kann nur in ein temporäres Verzeichnis und in das zugewiesene Verzeichnis des Nix-Store schreiben.

Binärpakete

- Verzeichnisnamen von Store-Einträgen werden durch die Ausdrücke bestimmt. Auf verschiedenen Systemen gleicher Art haben führen dieselben Ausdrücke zu denselben Verzeichnisnamen.
- Der Download eines vorkompilierten Binärpakets von einem Server ist eine einfache Optimierung.
- Binär-Patches können verwendet werden, um die Größe der Downloads zu reduzieren.

Closed-source software

- Es ist kein Problem, kommerzielle Programme mittels Nix-Ausdrücken zu beschreiben.
- ELF-Binärdateien enthalten die Pfade von Dateien, die benötigt werden, in lesbarer und modifizierbarer Form.
- Nix enthält ein Hilfsprogramm (`patchelf`), das verwendet werden kann, um Abhängigkeiten von Binärsoftware so umzuschreiben, daß sie auf Store-Einträge verweisen.

Profile

- Ein **Profil** bietet eine Sicht auf eine Menge von Store-Einträgen.
- Viele Profile pro System (systemweit und benutzerspezifisch).
- Jeder Benutzer kann seine eigenen Pakete installieren und wird nicht durch die Entscheidungen anderer Benutzer beeinträchtigt.
- Jedes Profil enthält eine Liste von **Benutzerumgebungen**.
- Jede Benutzerumgebung wird selbst als Store-Eintrag realisiert.
- Wenn ein Benutzer ein neues Paket installiert, erzeugt das eine neue Benutzerumgebung.
- Es ist einfach, zu einer alten Konfiguration zurück zu wechseln.

Benutzerschnittstelle

- `nix-env -i firefox`
  installiert die neueste Version von Firefox.
- `nix-env -i firefox-3.0`
  installiert eine spezifische Version.
- `nix-env -u firefox`
  "ersetzt" die Version von Firefox durch die neueste.
- `nix-env -e firefox`
  "loescht" Firefox aus dem Profil (aber nicht aus dem Store!).

Garbage collection

- Garbage collection löscht unbenutzte Einträge aus dem Nix-Store.
- Erfolgt nur, wenn explizit (vom Administrator) aufgerufen:

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

- Kann die Möglichkeit, zu alten Konfigurationen zurückzuwechseln, beeinträchtigen.

NixOS

Von Nix zu NixOS

- Im Prinzip setzt Nix nur ein Unix-artiges System voraus, arbeitet also mit beliebigen Linux-Distributionen, BSD, Solaris, MacOS, Windows/Cygwin usw.
- NixOS is eine eigenständige Linux-Distribution, die Nix nicht nur als Paketmanager verwenden, sondern auch für die Konfiguration des Systems selbst.
- Nix-Ausdrucke in NixOS beschreiben:
  - Software
  - den Linux-Kernel und seine Module
  - Konfigurationsdateien
  - Services (sshd, X, Webserver, ...)

Nixpkgs – eine Sammlung von Nix-Paketen

- Nix-Ausdrücke für mehr als 1300 Pakete (werden ständig mehr).
- GCC, X11, KDE, Gnome, Apache, PostgreSQL, GHC, OCaml, ...
- Auch closed-source Software wie zum Beispiel Acrobat Reader und Flash-Player.
- Die Auswahl ist stark bestimmt durch die Präferenzen der gegenwärtigen Entwickler.
- Paketbeschreibungen in unterschiedlicher Allgemeinheit. Bestimmte Standardkonfigurationen werden automatisch auf einer "build farm" getestet und stehen dann als Binärpakete zur Verfügung.

Systemkonfiguration

- Das gesamte System wird beschrieben durch einen Nix-Ausdruck in

  `/etc/nixos/nixos/default.nix`

  Das Auswerten dieses Ausdrucks erstellt die Systemkonfiguration.
- Anpassungen lassen sich einfach vornehmen in

  `/etc/nixos/configuration.nix`

  (das durch `/etc/nixos/nixos/default.nix` gelesen wird).

Beispielkonfiguration

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

Systemkomponenten

- Fast alles wird zum Store-Eintrag: der verwendete Kernel liegt im Store, genauso wie Kernel-Module.
- Paketspezifische Konfigurationsdateien werden automatisch generiert und lagern ebenfalls im Store.
- Konfigurationsdateien, die von mehreren Programmen benötigt werden (z.B. `/etc/hosts`) werden symbolisch verlinkt nach `/etc`.
- Services werden verlinkt nach `/etc/event.d`.
- Einige wenige Konfigurationsdateien (`/etc/passwd`) werden nicht im Store verwaltet. NixOS überprüft nur die Existenz gewisser Einträge.

Ändern der Konfiguration

- Wird die Systembeschreibung ausgewertet, erstellt diese auch ein Aktivierungs-Skript, das nach Bedarf Services startet und beendet, die benötigten Dateien nach `/etc` verlinkt, ...
- Um die Konfiguration zu ändern, muß man
  - `configuration.nix` anpassen, und
  - `nixos-rebuild switch` aufrufen.
- Dann wird
  - der System-Nix-Ausdruck neu ausgewertet,
  - das erstellte Aktivierungs-Skript ausgeführt,
  - die Konfiguration in einem speziellen Profil mit dem Name `system` zugänglich gemacht,
  - das (GRUB) Bootmenu wird aus der History des `system`-Profils generiert.

Vorteile

Alle Vorteile von Nix übertragen sich auf die Systemkonfiguration:

- es ist einfach, eine alte Konfiguration wiederherzustellen,
- das Erstellen einer Konfiguration beeinflußt noch nicht das laufende System,
- man kann die gleiche Konfiguration auf verschiedenen Systemen reproduzieren.

Erkenntnisse

- Es funktioniert tatsächlich. Zustand verkompliziert Dinge nur unnötig.
- Für das, was NixOS tut, ist es essentiell, daß Nix eine reine (und lazy) funktionale Programmiersprache ist.
- Auch wenn wir die “Reinheit” nicht komplett erzwingen, zeigt die Praxis, daß der gegenwärtige Ansatz zufriedenstellend funktioniert.

Möglichkeiten für die Zukunft

- Noch mehr Garantien.
- Ein statisches Typsystem.
- Warum nur ein System beschreiben? Mit Nix-Ausdrücken können auch Netzwerke beschrieben werden!

http://nixos.org