Purely Functional System Configuration Management

Software Technology Colloquium, UU

Eelco Dolstra[1] Armijn Hemel[2]

[1]Universiteit Utrecht, Faculty of Science,
Department of Information and Computing Sciences

[2]Loohuis Consulting

April 19, 2007

Universiteit Utrecht

Overview

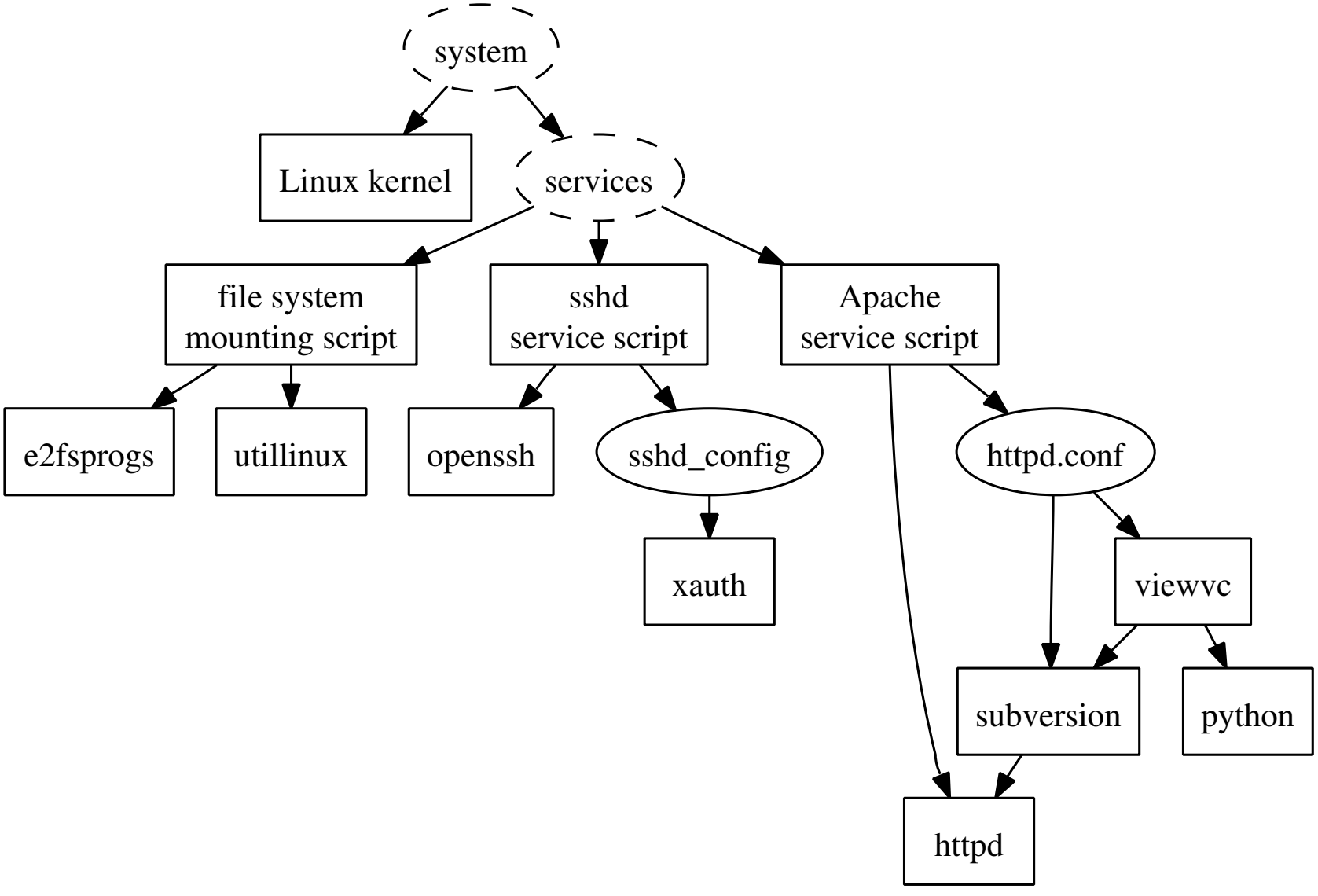
- Operating systems are installed and managed using tools that have an *imperative model*
- This causes lots of problems: upgrading is unreliable, rollbacks are hard, etc.
- This paper shows that it is possible to implement a system with a *purely functional model*
- Implemented in a Linux distribution called *NixOS*

Introduction

What is a system configuration?

- Software packages
- Configuration files
- System scripts
- ...

Example of a configuration



Imperative model

System configuration tools have an *imperative model*: configuration actions are *stateful*.

Examples

- Package management tools such as RPM, apt, Windows installers perform *destructive updates*: they overwrite existing files.
- Configuration tools such as Cfengine (declaratively) specify imperative updates to configuration files.
- Windows installers overwrite registry entries.

Why is statefulness bad?

- No *traceability*
  - Configuration is the result of a sequence of (sometimes manual) imperative actions over time
  - Hard to reproduce a configuration
- No *predictability* (determinism)
  - If an action depends on an ill-defined initial state, then the result is probably ill-defined
  - This is why upgrading is riskier than a full re-install
- Configuration actions clobber the previous configuration
  - No rollbacks
  - Hard to safely test a configuration

Analogous to imperative languages

Imperative languages such as C and Java have analogous problems. E.g., cannot reason about the result of function calls due to global variables or I/O.

There is a better way!

- *Purely functional languages* like Haskell
  - No mutable variables, data structures
  - Function result only depends on function arguments
  - $x = y \Rightarrow f(x) = f(y)$
  - *Referential transparency*
- No referential transparency in existing system CM tools
- So we need purely function system configuration management!

There is a better way!

Goal: purely functional system configuration management

- All static parts of a configuration should be *immutable*
- Configurations should be built by *pure functions*

Nix: Purely functional package management

- Deployment system developed at Utrecht University: `http://nix.cs.uu.nl/`
- *Purely functional* package management:
  - Packages builds only depend on declared inputs
  - Packages never change after they have been built

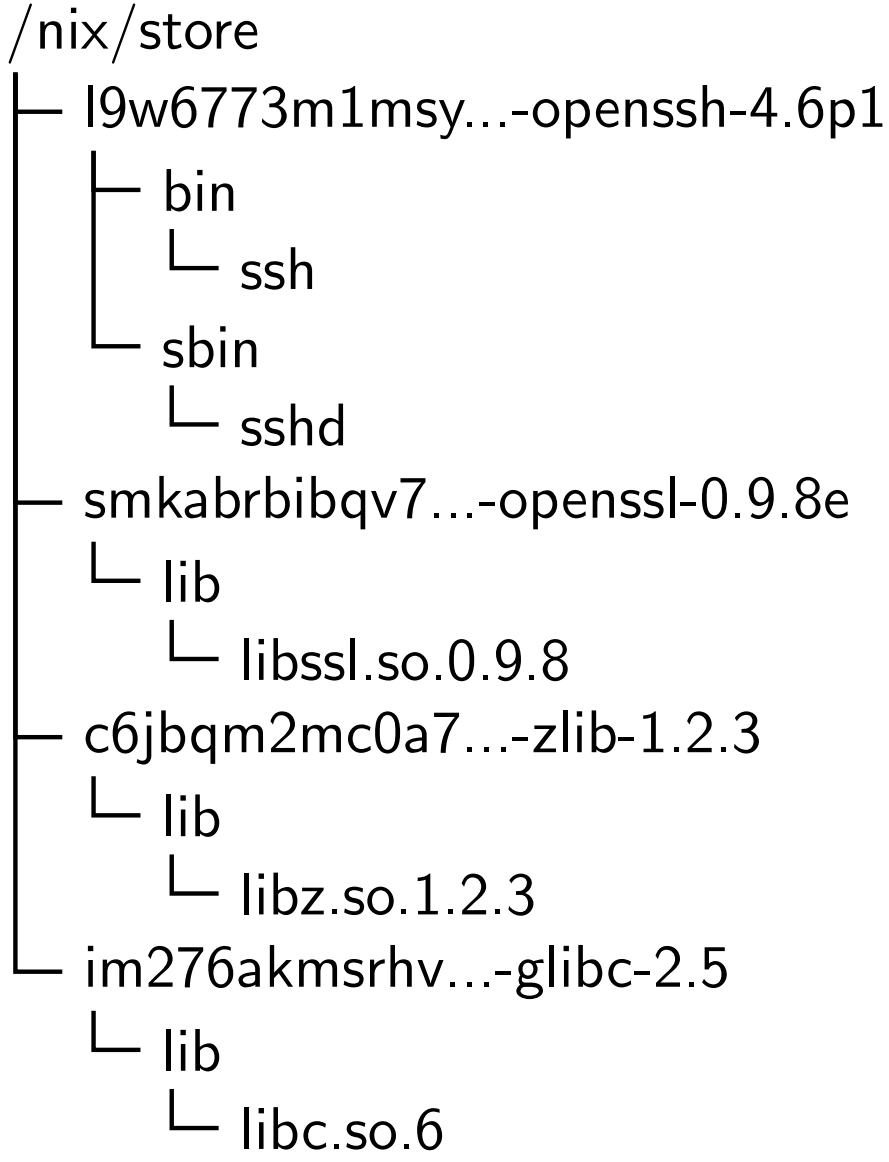
Nix store

Central idea: store all packages in isolation from each other:

`/nix/store/axrzx0rh0ivw...-firefox-2.0.0.3`

Paths contain a 160-bit **cryptographic hash** of **all** inputs used to build the package:

- Sources
- Libraries
- Compilers
- Build scripts
- ...



Nix expressions

Nix expressions describe how to build packages.

```
{stdenv, fetchurl, openssl, zlib}:

stdenv.mkDerivation {
  name = "openssh-4.6p1";
  src = fetchurl {
    url = http://.../openssh-4.6p1.tar.gz;
    sha256 = "0fpjlr3bfind0y94bk442x2p...";
  };
  buildCommand = "
    tar xjf $src
    ./configure --prefix=$out --with-openssl=${openssl}
    make; make install ";
}
```

Nix expressions

Nix expressions describe how to build packages.

```
{stdenv, fetchurl, openssl, zlib}:

stdenv.mkDerivation {
  name = "openssh-4.6p1";
  src = fetchurl {
    url = http://.../openssh-4.6p1.tar.gz;
    sha256 = "0fpjlr3bfind0y94bk442x2p...";
  };
  buildCommand = "
    tar xjf $src
    ./configure --prefix=$out --with-openssl=${openssl}
    make; make install ";
}
```

Function arguments

Nix expressions

Nix expressions describe how to build packages.

```
{stdenv, fetchurl, openssl, zlib}:

stdenv.mkDerivation {
  name = "openssh-4.6p1";
  src = fetchurl {
    url = http://.../openssh-
    sha256 = "0fpjlr3bfind0y94bk442x2p...";
  };
  buildCommand = "
    tar xjf $src
    ./configure --prefix=$out --with-openssl=${openssl}
    make; make install ";
}
```

Function arguments

Build attributes

Nix expressions

system/all-packages.nix

```
openssh = import ../tools/networking/openssh {
  inherit fetchurl stdenv openssl zlib;
};

openssl = import ../development/libraries/openssl {
  inherit fetchurl stdenv;
};

zlib = import ../development/libraries/zlib {
  inherit fetchurl stdenv;
};

fetchurl = ...;

stdenv = ...;
```

Nix expressions

system/all-packages.nix

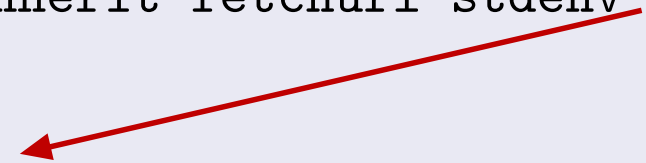
```
openssh = import ../tools/networking/openssh {
  inherit fetchurl stdenv openssl zlib;
};

openssl = import ../development/libraries/openssl {
  inherit fetchurl stdenv;
};

zlib = import ../development/libraries/zlib {
  inherit fetchurl stdenv;
};

fetchurl = ...;

stdenv = ...;
```



Taking it all the way

- Since we can build packages...
- ...why not build all the other stuff that goes into a configuration?
- It's all the same, really. As long as it's pure, we can build it!

The result: NixOS

- Linux distribution: didn't write anything ourselves, except for Nix and the necessary glue
- What we have:
  - Hardware support: networking, sound, video
  - RAID, LVM
  - System daemons: SSH, Apache, CUPS, dhcpd, NTP, Cron, Mingetty, ...
  - X11
  - KDE, most of Gnome
  - All the applications and tools in Nixpkgs (850 or so Unix packages)

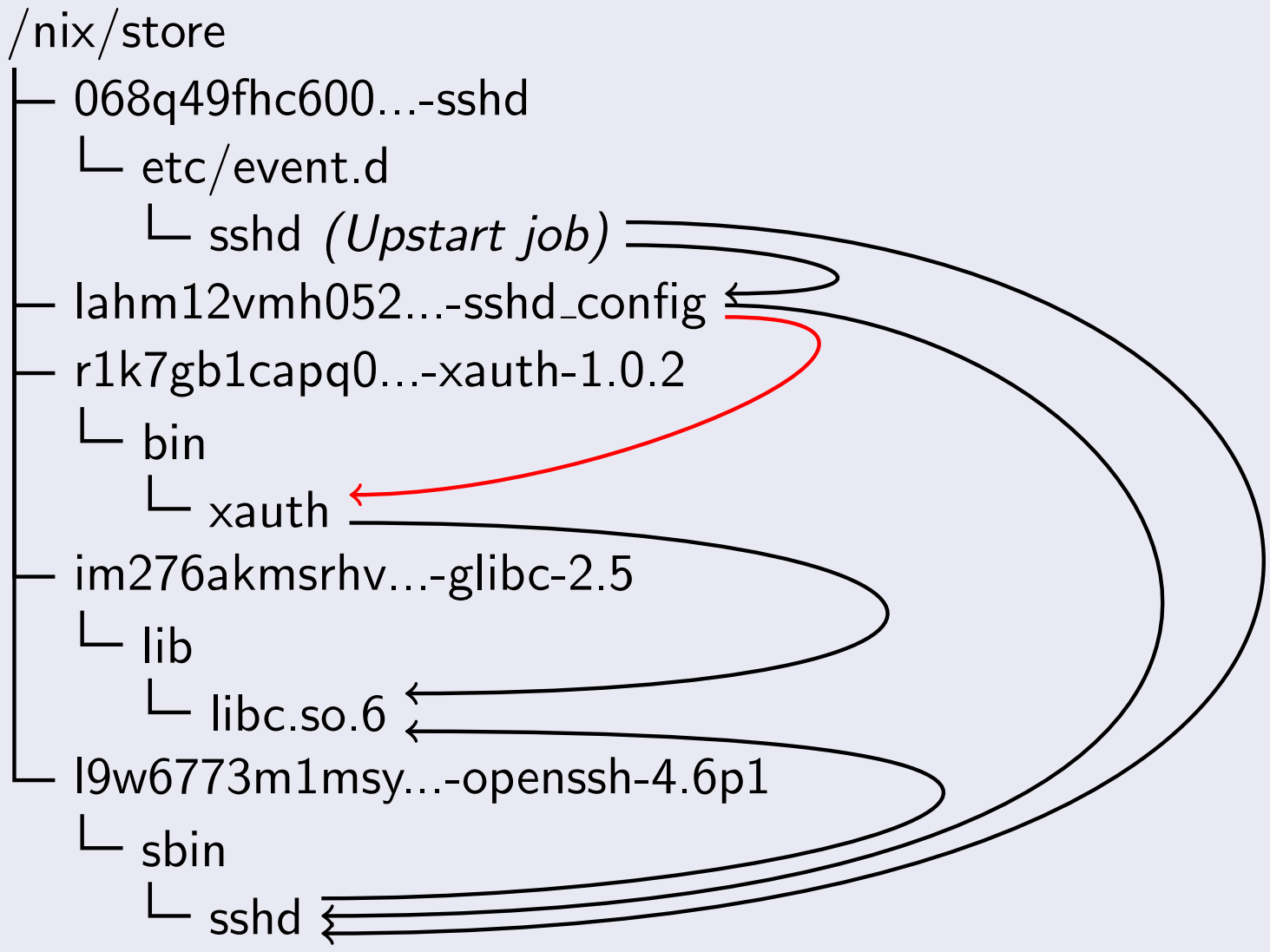
NixOS


MPlayer
The GIMP
Paintbrush
tyros:eelco:~
tyros:eelco:~/Dev/nixpkgs/pkgs
top - 20:29:37 up 22:32, 3 users, load average: 0.51, 0.16, 0.08
Tasks: 124 total, 3 running, 121 sleeping, 0 stopped, 0 zombie
Cpu(s): 7.3%us, 1.3%sy, 0.0%ni, 90.7%id, 0.0%wa, 0.7%hi, 0.0%si, 0.0%st
Mem: 507428k total, 489308k used, 18120k free, 32412k buffers
Swap: 1020088k total, 41244k used, 978844k free, 128256k cached
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
5869 root 15 0 434m 90m 7172 R 6.0 18.2 17:03.23 X
27101 eelco 15 0 80536 7740 3104 R 1.3 1.5 0:18.22 compiz
26739 eelco 15 0 30932 16m 8784 S 0.7 3.4 0:09.05 gnome-terminal
1 root 16 0 1604 652 552 S 0.0 0.1 0:03.53 init
2 root RT 0 0 0 0 S 0.0 0.0 0:00.00 migration/0
3 root 34 19 0 0 0 S 0.0 0.0 0:00.42 ksoftirqd/0
4 root RT 0 0 0 0 S 0.0 0.0 0:00.00 watchdog/0
5 root 10 -5 0 0 0 S 0.0 0.0 0:00.72 events/0
6 root 18 -5 0 0 0 S 0.0 0.0 0:00.01 khelper
7 root 10 -5 0 0 0 S 0.0 0.0 0:00.00 kthread
35 root 10 -5 0 0 0 S 0.0 0.0 0:00.97 kblockd/0
36 root 10 0 0 0 0 S 0.0 0.0 0:00.04 kacpid
158 root 15 -5 0 0 0 S 0.0 0.0 0:00.03 kseriod
179 root 11 -5 0 0 0 S 0.0 0.0 0:02.78 pdflush
16 -5 0 0 0 S 0.0 0.0 0:05.63 kswapd0
0:00.00 aio/0
0:00.00 kpsmoused

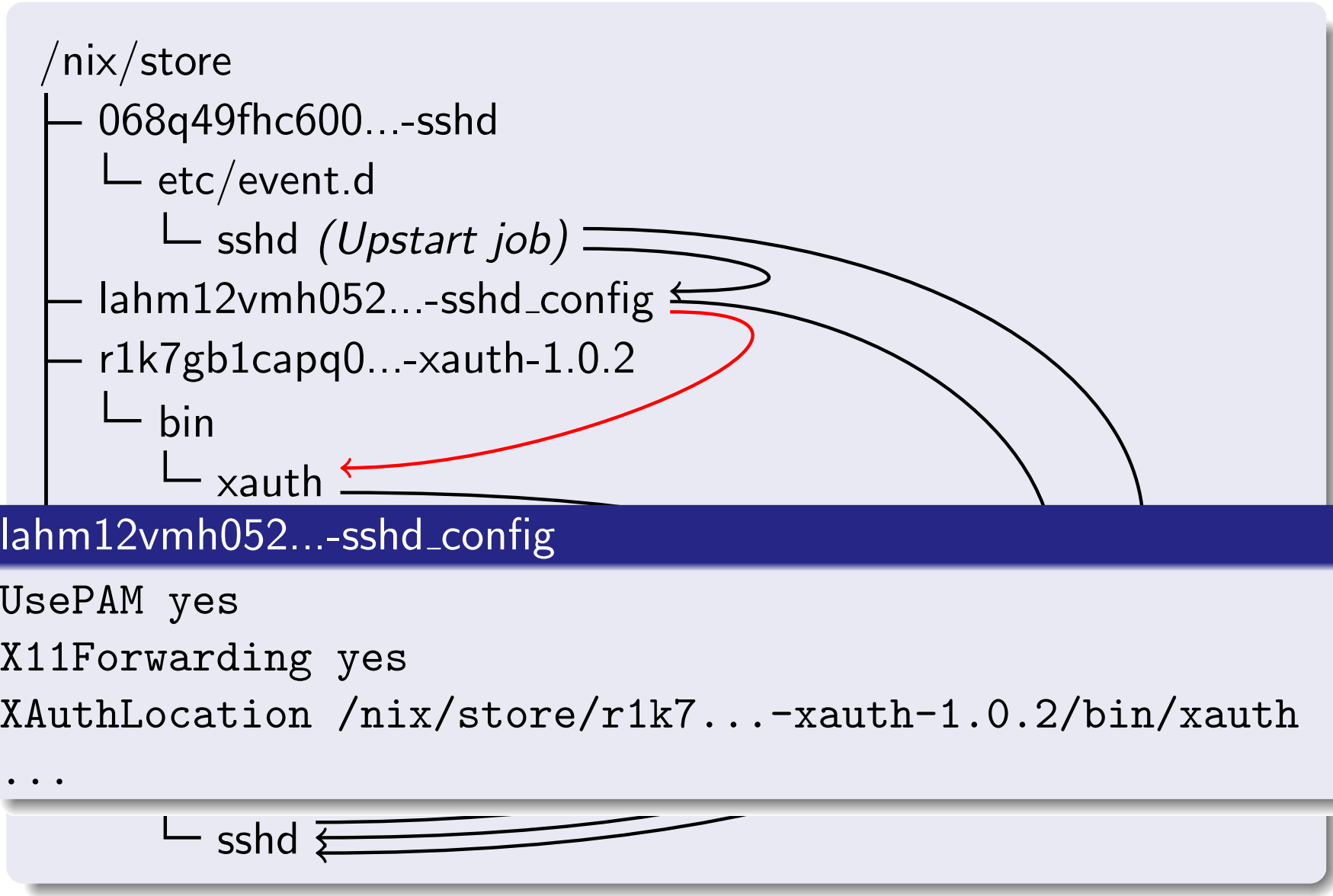
What Nix expressions did we need?

- NixOS is currently 49 Nix expressions, about 6200 lines.
- Building Upstart jobs for starting system daemons / services.
- Building /etc configuration files.
- Building the *initial ramdisk* for booting.
- Building the Grub boot menu.
- Building the *activation script*.
- Building additional boot scripts.
- Building the ISO image (purely functional!).
- Building system management scripts: *nixos-rebuild*.

Example


/nix/store
068q49fhc600...-sshd
etc/event.d
sshd (Upstart job)
lahm12vmh052...-sshd_config
r1k7gb1capq0...-xauth-1.0.2
bin
xauth
im276akmsrhv...-glibc-2.5
lib
libc.so.6
l9w6773m1msy...-openssh-4.6p1
sbin
sshd

Example



lahm12vmh052...-sshd_config

```
UsePAM yes
X11Forwarding yes
XAuthLocation /nix/store/r1k7...-xauth-1.0.2/bin/xauth
...
```

Example

Nix expression for sshd_config

```
{writeText, forwardX11, xauth}:

writeText "sshd_config" "
  UsePAM yes

  ${if forwardX11 then "
    X11Forwarding yes
    XAuthLocation ${xauth}/bin/xauth
  " else "
    X11Forwarding no
  "}
"
```

Example

Nix expression for the sshd Upstart job

```
{makeJob, openssh, sshdConfig}:

makeJob {
  name = "sshd";
  job = "
    description \"SSH server\"
    start on network-interfaces/started

    start script
      if ! test -f /etc/ssh/ssh_host_dsa_key; then
        ${openssh}/bin/ssh-keygen ...
      fi
      ...
    end script

    respawn ${openssh}/sbin/sshd -D -f ${sshdConfig}
  ";
}
```

The top-level system configuration

/etc/nixos/nixos/system/system.nix

- Top-level Nix expression.
- Calls other expressions to build the Upstart jobs, kernel, initrd, boot scripts...
- Takes as argument a *system configuration* — nested attribute set specifying system parameters.

The system configuration file

/etc/nixos/configuration.nix

```
{
  boot = { grubDevice = "/dev/hda"; };
  fileSystems = [
    { mountPoint = "/";
      device = "/dev/hda1";
    }
  ];
  swapDevices = [ { device = "/dev/hdb1"; } ];
  services = {
    sshd = {
      enable = true;
      forwardX11 = true;
    };
  };
}
```

Building it

Normal operation: switch

```
$ nixos-rebuild switch
```

Builds configuration (by calling system.nix with /etc/nixos/configuration.nix as argument), makes it boot default, activates it. Previous configurations still reachable through boot menu.

Test

```
$ nixos-rebuild test
```

Builds and activates configuration. Reboot reverts.

Building it

Normal operation: switch

```
$ nixos-rebuild switch
```

Builds configuration (by calling system.nix with /etc/nixos/configuration.nix as argument), makes it boot default, activates it. Previous configurations still reachable through boot menu.

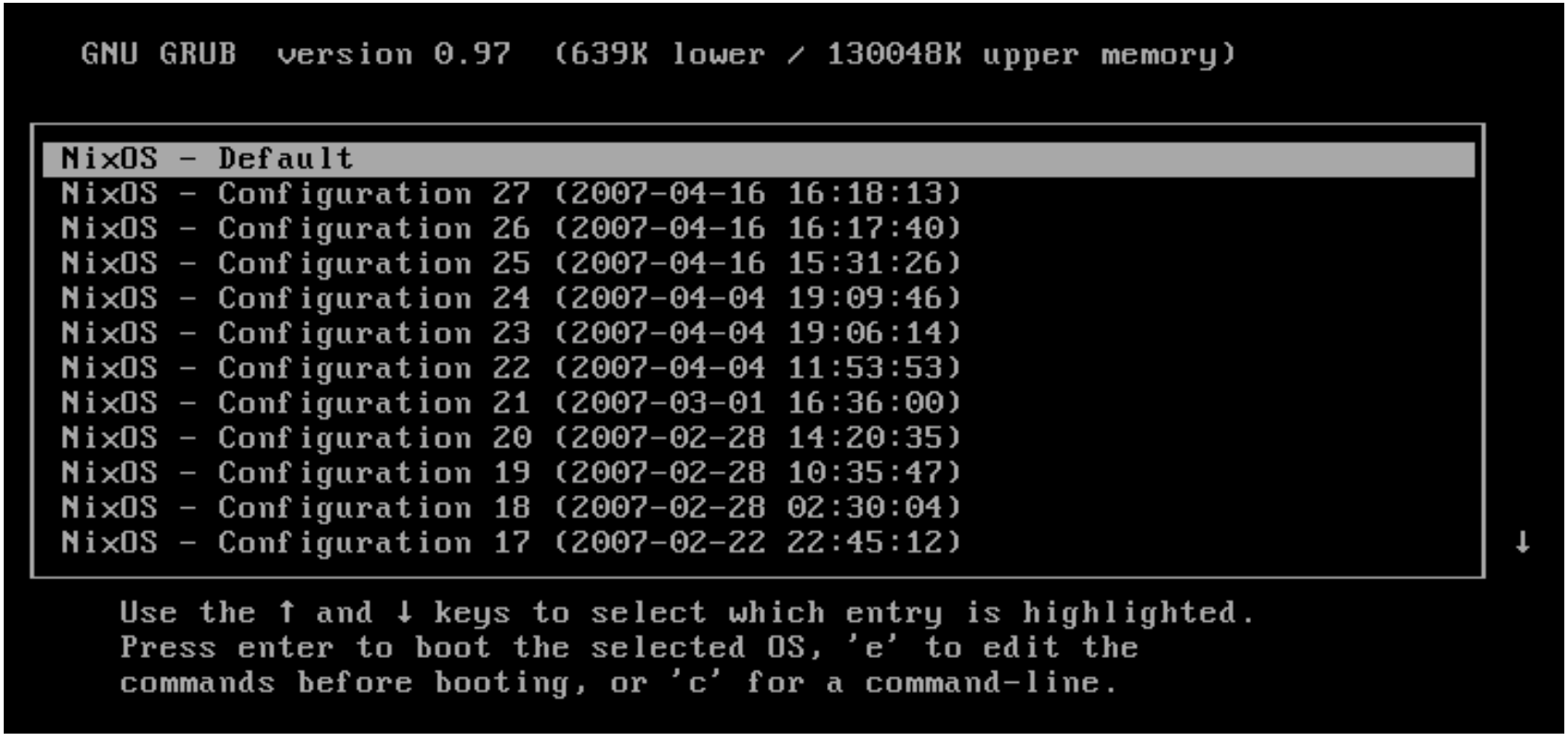
Test

```
$ nixos-rebuild test
```

Builds and activates configuration. Reboot reverts.

Booting it

Grub boot menu is synthesized from the available (non-garbage collected) generations of the system profile.



Evaluation

How pure are we?

- No /bin (with 1 exception), /sbin, /lib, /usr.
  - Sole exception: /bin/sh
- Almost all of /etc resides in the Nix store
  - E.g. sshd_config, Upstart job specifies full store path
  - But some configuration files are cross-cutting (/etc/resolv.conf, /etc/services), so we symlink them in /etc
- Mutable state (/var): don't do anything special with it
  - Nasty: hybrid configuration / state: /etc/passwd

Evaluation

How pure are we?

- No /bin (with 1 exception), /sbin, /lib, /usr.
  - Sole exception: /bin/sh
- Almost all of /etc resides in the Nix store
  - E.g. sshd_config, Upstart job specifies full store path
  - But some configuration files are cross-cutting (/etc/resolv.conf, /etc/services), so we symlink them in /etc
- Mutable state (/var): don't do anything special with it
  - Nasty: hybrid configuration / state: /etc/passwd

Evaluation

How pure are we?

- No /bin (with 1 exception), /sbin, /lib, /usr.
  - Sole exception: /bin/sh
- Almost all of /etc resides in the Nix store
  - E.g. sshd_config, Upstart job specifies full store path
  - But some configuration files are cross-cutting (/etc/resolv.conf, /etc/services), so we symlink them in /etc
- Mutable state (/var): don't do anything special with it
  - Nasty: hybrid configuration / state: /etc/passwd

The downside

- X11/KDE configuration takes up 656 MiB in 236 store paths.
- That's okay, but...
- ...if Glibc changes, than we'll need another 656 MiB.
- But disk space is cheap.
- However, build / download time isn't.
- Analogous to purely functional data structures:
  - x : tail list is cheap
  - init list ++ [x] is expensive
- Binary patching helps a lot.

Conclusion

- NixOS shows that a purely functional system configuration model is feasible and practical.
- Advantages: reproducibility, determinism, predictable upgrading.
- Worth mentioning: multi-user package management, any user can install software, with sharing.
- Disadvantage: can take up to 2x as much disk space.

More information / download

- `http://www.nixos.org/`
- ISO images for x86, x86_64.
- Easy way to play with it: install in a VM or on a USB stick.