NixOS: A Purely Functional Linux Distribution [*]

Eelco Dolstra
Delft University of Technology
e.dolstra@tudelft.nl

Andres Löh
Utrecht University
andres@cs.uu.nl

Abstract

Existing package and system configuration management tools suffer from an *imperative model*, where system administration actions such as upgrading packages or changes to system configuration files are stateful: they destructively update the state of the system. This leads to many problems, such as the inability to roll back changes easily, to run multiple versions of a package side-by-side, to reproduce a configuration deterministically on another machine, or to reliably upgrade a system. In this paper we show that we can overcome these problems by moving to a *purely functional system configuration model*. This means that all static parts of a system (such as software packages, configuration files and system startup scripts) are built by pure functions and are immutable, stored in a way analogously to a heap in a purely function language. We have implemented this model in *NixOS*, a non-trivial Linux distribution that uses the *Nix package manager* to build the entire system configuration from a purely functional specification.

1. Introduction

Current operating systems are managed in an *imperative* way. With this we mean that configuration management actions such as upgrading software packages, making changes to system options, or adding additional system services are done in a stateful way: by performing destructive updates to files. This model leads to many problems: the "DLL hell" (where upgrading an application may suddenly break another because of changes to shared components (Anderson 2000)), the inability to roll back changes easily or to reproduce a configuration reliably, difficulty in running multiple versions of a package side-by-side, the system being in an inconsistent state during updates, and so on.

In this paper we show that it is possible to manage systems in a radically different way by moving to a *purely functional* model. In this model, the static artifacts of a running system — software packages, configuration files, system startup scripts, etc. — are generated by functions in a purely functional specification language. Just like values in a purely functional language, these artifacts never change after they have been built; rather, the system is updated to a new configuration by changing the specification and rebuilding the system from it. This allows a system to be built deterministically, and therefore reproducibly. It allows the user to roll back the system to previous configurations, since previous configurations are not overwritten. Perhaps most importantly, statelessness makes configuration actions predictable: they do not mysteriously fail because of some unknown aspect of the state of the system.

We have previously shown how *package management* — the installation and management of software packages — can be done in a purely functional way, in contrast to the imperative models of conventional tools such as RPM (Foster-Johnson 2003). This concept was implemented in the Nix package manager (Dolstra et al. 2004; Dolstra 2006), summarised in Section 3. In this paper we extend this approach from package management to system configuration management. That is, not just software packages are built from purely functional specifications, but also all other static parts of a system, such as the configuration files that typically live in /etc under Unix.

We demonstrate the feasibility of this approach by means of *NixOS*, a Linux distribution that uses Nix to construct and update the whole system from a declarative specification. Every artifact is stored under an immutable path such as /nix/store/cj22mw17...-linux-2.6.23.17 that includes a cryptographic hash of all inputs involved in building it. There is no /usr, /lib or /opt, and only a very minimal /bin and /etc.

NixOS's purely functional approach to configuration management gives several advantages to users and administrators. Upgrading a system is much more deterministic: it does not unexpectedly fail depending on the previous state of the system. Thus, upgrading is as reliable as installing from scratch, which is generally not the case with other operating systems. This also makes it easy to reproduce a configuration on a different machine. The entire system configuration can be trivially rolled back. The system is not in an inconsistent state during upgrades; upgrades are atomic. Unprivileged users can securely install different versions of software packages without interfering with each other.

The contributions of this paper are as follows:

- We show how a full-featured Linux distribution (NixOS) can be built and configured in a declarative way using principles borrowed from purely functional languages (Section 5).
- We discuss in detail the lazy purely functional Nix expression language that describes how to build system configurations, and why purity and laziness are essential (Section 4).
- We measure the extent to which NixOS and Nix build actions are pure (Section 6).

2. Imperative package management

Most package management tools can be viewed as having an *imperative model*. That is, deployment actions performed by these tools are stateful; they destructively update files on the system.

[*] Note to reviewers: An overview of an early prototype of NixOS appeared in (Dolstra and Hemel 2007). This paper provides a much more detailed exposition of NixOS (Section 5), a reflection on Nix language features (Section 4) and empirical results on the purity of Nix expressions (Section 6).

[Copyright notice will appear here once 'preprint' option is removed.]

For instance, most Unix package managers, such as the Red Hat Package Manager (RPM) (Foster-Johnson 2003), Debian's apt and Gentoo's Portage, store the files belonging to each package in the conventional Unix file system hierarchy, e.g. directories such as /bin. Packages are upgraded to newer versions by overwriting the old versions. If shared components are not completely backwards-compatible, then upgrading a package can break other packages. Upgrade actions or uninstallations cannot easily be rolled back. The Windows registry is similarly stateful: it is often impossible to have two versions of a Windows application installed on the same system because they would overwrite each other's registry entries.

Thus, the filesystem (e.g., /bin and /etc on Unix, or C:\Windows\System32 on Windows) and the registry are used like mutable global variables in a programming language. This means that there is no referential transparency. For instance, a package may have a filesystem reference to some other package, e.g. /usr/bin/perl. But this reference does not point to a fixed value; the referent can be updated at any point, making it hard to give any assurances about the behaviour of the packages that refer to it. For instance, upgrading some application might trigger an upgrade of /usr/bin/perl, which might cause other applications to break. This is known as the "DLL hell" or "dependency hell".

Statefulness also makes it hard to support multiple versions of a package. If two packages depend on conflicting versions of /usr/bin/perl, we cannot satisfy both at the same time. Instead, we would have to arrange for the versions to be placed under different filenames, which requires explicit actions from the packager or the administrator (e.g., install a Perl version under /usr/local/perl-5.8). This also applies to configuration files. For instance, running two instances of an Apache web server might require cloning and tweaking configuration files, control scripts, and so on.

Likewise, building a package is a stateful operation. In RPM, the building of a binary package is described by a *spec file* that lists the build actions that must be performed to construct the package, along with metadata such as its dependencies. However, the dependency specification has two problems.

First, it is hard to guarantee that the dependency specification is *complete*. If, for instance, the package calls the python program, it will build and run fine on the build machine if it has the python package installed, even if that package is not listed as a dependency. However, on the end user machine python may be missing, and the package will fail unexpectedly.

Second, RPM dependency specifications are *nominal*, that is, they specify just the name (and possibly some version constraints) of each dependency, e.g.

```
Requires: python >= 2.4
```

A package with this dependency will build on any system where python with a sufficiently high version is registered in RPM's database; however, the resulting binary RPM has no record of precisely *what* instance of python was used at build time, and therefore there is no way to deterministically reproduce it from source. Indeed, it is not clear how to bootstrap a set of source RPMs: one quickly runs into circular build-time dependencies and incomplete dependency specifications.

Package managers like RPM are even more stateful when it comes to non-software artifacts such as configuration files in /etc. When a configuration file is installed for the first time, it is treated as a normal package file. On upgrades, however, it cannot be simply overwritten with the new version, since the user may have made modifications to it. There are many *ad hoc* (package-specific) solutions to this problem: the file can be ignored, hoping that the old one still works; it can be overwritten (making a backup of the old version), hoping that the user's changes are inessential; or a *post-install script* (delivered as part of the package's meta-data) can attempt to merge the changes. Indeed, post-install scripts are frequently used to perform arbitrary, strongly stateful, actions.

Even worse, configuration changes are typically not under the control of a system configuration management tool like RPM at all. Users might manually edit configuration files in /etc or change some registry settings, or run tools that make such changes for them — but either way there is no trace of such actions. The fact that a running system is thus the result of many *ad hoc* stateful actions makes it hard to reproduce (part of) a configuration on another machine, and to roll back configuration changes.

Of course, configuration files could be placed under revision control, but that does not solve everything. For instance, configuration data such as files in /etc are typically related to the software packages installed on the system. Thus, rolling back the Apache configuration file may also require rolling back the Apache package (for instance, if we upgraded Apache, made some related changes to the configuration, and now wish to undo that change). Furthermore, changes to configuration files are often *actualised* in very different ways: a change to the Unix filesystem table /etc/fstab might be actualised by running mount -a to mount any new filesystems added to that file, while a change to the Apache web server configuration file httpd.conf requires running a command like /etc/init.d/apache reload. Thus any change to a configuration file, including a rollback, may require specific knowledge about additional commands that need to be performed.

In summary, all this statefulness comes at a high price to users:

- It is difficult to allow *multiple versions* of a package on the system at the same time, or to run multiple instantiations of a service (such as a web server).
- There is *no traceability*: the configuration of the system is a result of a sequence of stateful transformations that are not under the control of a configuration management system. This makes it hard to reproduce a configuration elsewhere. The lack of traceability specifically makes *rollbacks* much harder.
- Since configurations are the result of stateful transformations, there is *little predictability*. For instance, upgrading a Linux or Windows installation, as opposed to doing a clean reinstall, tends to be much more error-prone, precisely because the upgrade depends on the previous state of the system, while a clean reinstall has no such dependency.

These problems are similar to those caused by the lack of referential transparency in imperative programming languages, such as the difficulty in reasoning about the behaviour of functions in the presence of mutable global variables or I/O. Indeed, the absence of such problems is a principal feature of purely functional languages such as Haskell (Hudak 1989). In such languages, the result of a function depends only on its inputs, and variables are immutable. This suggests that the deployment problems above go away if we can somehow move to a *purely functional* way to store software packages and configuration data.

3. Purely functional package management

Nix (Dolstra et al. 2004; Dolstra 2006), the package manager underlying NixOS, has such a purely functional model. This means that packages are built by functions whose outputs in principle depend only on their function arguments, and that packages never change after they have been built.

Nix expressions Packages are built from *Nix expressions*, a dynamically typed, lazy, purely functional language. The goal of Nix expressions is to describe graphs of build actions called *derivations*. A derivation consists of a build script, a set of environment variables, and a set of dependencies (other derivations). A package

```
{stdenv, fetchurl, ghc, X11, xmessage}: $\boxed{1}$

let version = "0.5"; in

stdenv.mkDerivation $\boxed{2}$ (rec {

  name = "xmonad-${version}";

  src = fetchurl {
    url = "http://hackage.haskell.org/.../${name}.tar.gz";
    sha256 = "cfcc4501b000fa740ed35a5be87dc012...";
  };

  buildInputs = [ghc X11]; $\boxed{3}$

  configurePhase = '' $\boxed{4}$
    substituteInPlace XMonad/Core.hs --replace \
      '"xmessage"' '"${xmessage}/bin/xmessage"' $\boxed{5}$
    ghc --make Setup.lhs
    ./Setup configure --prefix="$out" $\boxed{6}$
  '';

  buildPhase = ''
    ./Setup build
  '';

  installPhase = ''
    ./Setup copy
    ./Setup register --gen-script
  '';

  meta = { $\boxed{7}$
    description = "A tiling window manager for X";
  };
})
```

Figure 1. xmonad.nix: Nix expression for xmonad

is built by recursively building the dependencies, then invoking the build script with the given environment variables.

Figure 1 shows the Nix expression for the xmonad package, a tiling X11 window manager written in Haskell. This expression is a *function* that takes a set of arguments (declared at point $\boxed{1}$) and returns a derivation (at $\boxed{2}$). The most important data type in the Nix expression language is the *attribute set*, a set of name/value pairs, written as $\{$ name$_1$ $=$ value$_1$; ...; name$_n$ $=$ value$_n$; $\}$. The keyword rec defines a recursive attribute set, i.e., the attributes can refer to each other. The syntax $\{$arg$_1$, ..., arg$_n\}$: body defines an anonymous function that must be called with an attribute set containing the specified named attributes.

The arguments of the xmonad function denote dependencies (stdenv, ghc, X11, and xmessage), as well as a helper function (fetchurl). stdenv is a package that provides a standard Unix build environment, containing tools such as a C compiler and basic Unix shell tools. X11 is a package providing the most essential X11 client libraries.

The function stdenv.mkDerivation is a helper function that makes writing build actions easier. Build actions generally have a great deal of “boiler plate” actions. For instance, most Unix packages are built by a standard sequence of commands: tar xf *sources* to unpack the sources, ./configure --prefix=*prefix* to detect system characteristics and generate makefiles, make to compile, and make install to install the package in the directory *prefix*. mkDerivation is a function that captures this commonality, allowing packages to be written succinctly. For instance, Figure 2 shows the function that builds the xmessage package. The xmonad package, being Haskell-based, does not build in this standard way, so mkDerivation allows the various phases of the package build

```
{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];
};
```

Figure 2. xmessage.nix: Nix expression for xmessage

sequence to be overridden, e.g. configurePhase $\boxed{4}$ specifies shell commands that configure the package. On the other hand, unpacking xmonad’s source follows the convention, so it is not necessary to specify an unpackPhase — the default functionality provided by mkDerivation suffices. We emphasise that mkDerivation is just a function that abstracts over a common build pattern, not a language construct: functions that abstract over other build patterns can easily be written. For instance, the xmonad package makes use of Haskell’s Cabal build system (http://haskell.org/cabal/). It therefore makes sense to extract the Cabal build commands (all the invocations of ./Setup) into a Cabal-specific wrapper around mkDerivation, thereby allowing the expression specific to xmonad to be reduced to the same conciseness as the xmessage example. Due to the functional nature of the Nix expression language, nearly all recurring patterns can be captured in functions in this way.

All attributes in the call to mkDerivation are passed as environment variables to the build script (except the meta attribute $\boxed{7}$, which is filtered out by the mkDerivation function). For instance, the environment variable name will be set to xmonad-0.5. For attributes that specify other derivations, the paths of the packages they build are substituted. For instance, the buildInputs environment variable (which is used by stdenv to generically set up other environment variables such as the C include file search path and the linker search path) will contain the paths to the ghc and X11 dependencies $\boxed{3}$. Likewise, the function fetchurl returns a derivation that downloads the specified file and verifies its content against the given cryptographic hash; thus, the environment variable src will hold the location of the xmonad source distribution, which the stdenv build script will unpack. (fetchurl may seem an impure function, but because the output is guaranteed to have a specific content by the cryptographic hash, which Nix verifies, it is pure as far as the purely functional deployment model is concerned.)

When invoking a build script to perform a derivation, Nix will set the special environment variable out to the intended location in the filesystem where the package is to be stored. Thus, xmonad is configured with a build prefix equal to the path given in out $\boxed{6}$.

Since the expression that builds xmonad is a function, it must be called with concrete values for its arguments to obtain an actual derivation. This is the essence of the purely functional package management paradigm: the function can be called any number of times with different arguments to obtain different instances of xmonad. Just as multiple calls to a function in a purely functional language cannot “interfere” with each other, these instances are independent: for instance, they will not overwrite each other.

Figure 3 shows an attribute set containing several concrete derivations resulting from calls to package build functions. The attribute xmonad is bound to the result of a call to the function in Figure 1 $\boxed{8}$. Thus, xmonad is a concrete derivation that can be built. Likewise, the arguments are concrete derivations, e.g. xmessage is the result of a call to the function in Figure 2.

The end user can now install xmonad by doing

```
$ nix-env -f all-packages.nix -i -A xmonad
```

```
rec {
  xmonad = import .../xmonad.nix { [8]
    inherit stdenv fetchurl ghc X11 xmessage;
  };

  xmessage = import .../xmessage.nix { ... };

  ghc = ghc68;

  ghc68 = import ../development/compilers/ghc-6.8 {
    inherit fetchurl stdenv readline perl gmp ncurses m4;
    ghc = ghcboot;
  };

  ghcboot = ...;
  stdenv = ...;
  ...
}
```

Figure 3. all-packages.nix: Function calls to instantiate packages

This will build the xmonad derivation and all its dependencies and ensure that the xmonad binary appears in the user's PATH, in a way described below. Derivations that have been built previously are not built again. Other operations include nix-env -e to uninstall a package, and nix-env --rollback to undo the previous nix-env action.

Nix is available at http://nixos.org/, along with the Nix Packages collection (Nixpkgs), a large collection of Nix expressions for around 1350 packages.

The Nix store Build scripts only need to know that they should install the package in the location specified by the environment variable out, which Nix computes before invoking each package's build script. So how does Nix store packages? It must store them in a purely functional way: different instances of a package should not interfere, i.e., should not overwrite each other.

Nix accomplishes this by storing packages as immutable children of the *Nix store*, the directory /nix/store, with a filename containing a cryptographic hash of all inputs used to build the package. For instance, a particular instance of xmonad might be stored under

/nix/store/8dpf3wcgkv1ixghzjhljj9xbcd9k6z9r-xmonad-0.5/

where 8dpf3wcgkv1ixghzjhljj9xbcd9k6z9r is a base-32 representation of a 160-bit hash. The inputs used to compute the hash are all attributes of the derivation (e.g., the attributes at $\boxed{2}$ in Figure 1, plus some default attributes added by the mkDerivation function). These typically include the sources, the build commands, the compilers used by the build, library dependencies, and so on. This is recursive: for instance, the sources of the compiler also affect the hash. Nix computes this path and passes it through the environment variable out to the build script.

Figure 4 shows a part of a Nix store containing xmonad and some of its build time and runtime dependencies. The solid arrows denote the existence of *references* between files, i.e., the fact that a file in the store contains the path of another store object. For instance, as is required for dynamically linked ELF executables (TIS Committee 1995), the xmonad executable contains the full path to the dynamic linker ld-linux.so.2 in the Glibc package.

Figure 4 also shows the notion of *profiles*, which enable atomic upgrades and rollbacks and allow per-user package management. Names that are *slanted* denote symlinks, with dotted arrows denoting symlink targets. The user has the directory /nix/var/nix/profiles/*user*/bin in her path. When a package is installed through nix-env -i, a pseudo-package called a *user environment* is automatically built that consists of symlinks to all installed packages for that user $\boxed{11}$. A symlink /nix/var/nix/profiles/*user-number* $\boxed{10}$ is created to point to that user environment, and finally the symlink /nix/var/nix/profiles/*user* $\boxed{9}$ is atomically updated to point at the former symlink. This makes the new set of packages appear in an atomic action in the user's PATH. A rollback is trivially done by flipping the symlink back (e.g. to /nix/var/nix/profiles/alice-14).

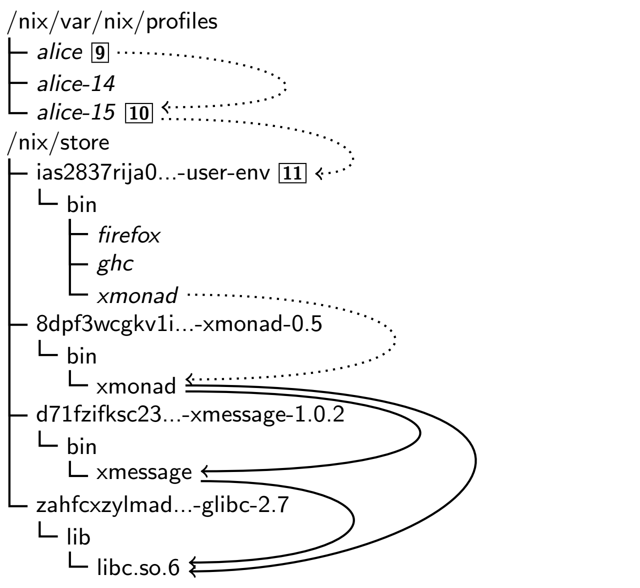

/nix/var/nix/profiles
alice 9
alice-14
alice-15 10
/nix/store
ias2837rija0...-user-env 11
bin
firefox
ghc
xmonad
8dpf3wcgkv1i...-xmonad-0.5
bin
xmonad
d71fzifksc23...-xmessage-1.0.2
bin
xmessage
zahfcxzylmad...-glibc-2.7
lib
libc.so.6

Figure 4. The Nix store, containing xmonad and some of its dependencies, and profiles

Advantages The purely functional approach has several advantages for package management:

- Nix prevents undeclared build time dependencies through the use of the store: since store paths are not in any default search paths, they will not be found.
- Nix detects runtime dependencies automatically by scanning for references, i.e., the hash parts of filenames. This is analogous to how the mark phase of a conservative garbage collector detects pointers in languages that do not have a formally defined pointer graph structure (Boehm 1993). Since the full dependency graph is known, we can reliably deploy a package to another machine by copying its *closure* under the references relation.
- Since packages are immutable, there is no DLL hell. For instance, if xmonad uses glibc-2.7, and the installation of some other package causes glibc-2.8 to be installed, it will not break xmonad: it will continue to link against glibc-2.7.
- Knowledge of the runtime dependency graph allows unused packages to be garbage collected automatically – however, in contrast to memory management in programming languages, the garbage collector is only run on user request. The roots of the collector are the profile symlinks in Figure 4.
- Since actions such as upgrading or downgrading to a different version are non-destructive, the previous versions are not touched. Therefore, it is possible to roll back to previous versions (unless the garbage collector has explicitly been invoked in the meantime). Also, since upgrades and rollbacks are *atomic*, the user never sees a half-updated package; the system is never in an inconsistent state at any point during the upgrade.

```
expressions
e ::= x                                  identifier
    | nat | str | uri | path             literal
    | [e*]                               list
    | rec? {b*}                          attribute set
    | let b* in e
    | e.x                                selection
    | x : e | {fs?} : e                  function
    | e e                                application
    | if e then e else e                 conditional
    | with e; e                          inclusion
    | (e)                                group
    | e ● e                              operator
    | !e                                 negation

bindings
b ::= x = e;
    | inherit x* | inherit (e) x*

formls
fs ::= x, fs | x, | x

operators
● ::= == | != | && | || | -> | // | ~ | +
```

Figure 5. Syntax of the Nix expression language

- While the Nix model is in essence a *source deployment model* (since Nix expressions describe how to build software from source), purity allows transparent binary deployment as an optimisation. One can register with Nix that store paths are available for download at a remote machine (such as the Nixpkgs distribution site). Then, when Nix needs to build some path p, it checks whether p is available remotely. If so, it is downloaded in lieu of building. Otherwise, it is built normally. Thus, binary deployment is a simple optimisation of source deployment.
- A purely functional build language allows build-time variability in packages to be expressed in a natural way, and allows abstracting over common build patterns.
- Having a declarative specification of the system pays off particularly when performing upgrades that invalidate a large number of packages. For instance, an upgrade of GCC or the core C libraries might essentially require nearly the entire system to be rebuilt. On a smaller, but not less annoying scale, every upgrade of GHC requires all Haskell libraries to be rebuilt. These relations can not only easily be expressed in Nix – the upgrade path is also extremely painless, as the old versions remain usable in parallel for as long as desired. The system will always be consistent and the different versions do not interfere.

4. The Nix expression language

In this section, we formalise the syntax and semantics of the Nix expression language, and motivate why certain language features are essential to the language.

4.1 Syntax

The Nix expression language is a dynamically typed purely functional language. The syntax of the language is shown in Figure 5. Next to the constructs shown, the language has a considerable number of built-in functions that syntactically are normal identifiers. We focus our description on the features that make the language special and well-suited for its task of describing packages.

There are a few base types: Booleans, natural numbers, strings, and paths. More complex types can be built using lists, attribute sets, and functions.

String literals String literals can be enclosed in double quotes (`"string"`) or two single quotes (`''string''`). Both forms of strings can span multiple lines and allow *interpolation*: arbitrary expressions can be inserted into the middle of strings by writing `${expr}` (such as at $\boxed{5}$ in Figure 1). Interpolations are desugared to string concatenations.

The second form, surrounded by two single quotes, is known as an *indented string*: Indentation is removed from an indented string in a clever way. Indented strings are very convenient for including shell scripts in a Nix expression. Shell language itself makes frequent use of quoting (with both single and double quotes) and escaping via `\`, but rarely contains the `''` combination. Therefore, large chunks of shell scripts can be included in a Nix expression almost literally using indented strings.

As a third form of string literals, the Nix language supports URI literals. URIs according to RFC 2396 can be specified directly, without using quotes, as a convenience feature: the programmer gets free syntax checking of URIs.

Paths Next to strings and URIs, Nix also handles path literals that are absolute or relative Unix paths and are given without surrounding quotes. Paths are treated slightly different from strings during evaluation.

Lists Lists can be specified by enclosing a space-separated sequence of elements within square brackets. No special elimination constructs are part of the language, but there are built-in functions for accessing the head and tail of a list. Lists can be heterogeneous, e.g., `[1 true [./file "endo lives"]]` is a valid list.

Attribute sets The central type of the Nix language is attribute sets. Attribute sets essentially are records. They are introduced by listing a number of bindings. Bindings either associate an attribute name with an expression, or introduce names from the surrounding scope by means of `inherit`. By specifying a source, `inherit` can also introduce names from another attribute set. An attribute x can be selected from an attribute set a by writing a.x.

Local bindings can be made by means of a `let-in` construct. An expression of the form `with` e_1; e_2 where e_1 describes an attribute set adds the attributes of e_1 to the scope while evaluating e_2. This is often used in the form `with import` path; e_2 where path refers to a file containing an attribute set – the construct then simulates opening and importing a module.

Functions Anonymous functions are introduced using a colon to separate the argument from the body. The identity function can be written as `x : x`, the constant function in curried form as `x : y : x`. For functions taking attribute sets as arguments, there is a special syntax that allows to pattern match on specific attributes by name (for instance used in the xmonad expression in Figure 1). Like in ML or Haskell, function application is denoted by juxtaposing two expressions.

Derivations The most important built-in function is `derivation`. Usually, a Nix expression will not directly invoke `derivation`, but rather call a wrapper function such as stdenv.mkDerivation that fills in boilerplate code. The function takes an attribute set and interprets it as a build action for a package. It returns the original attribute set, extended with a few additional attributes. One of the new attributes is outPath, the path in the Nix store where the complete output of the build action is stored.

The attribute set passed to the `derivation` function must define the attributes system, name, and builder. The attribute builder is interpreted as the invocation of a program that produces the

$$\frac{\mathsf{e} \rightarrow^{*} \{\vec{\mathsf{b}}\} \quad \mathsf{x} = \mathsf{e}' \in \vec{\mathsf{b}}}{\mathsf{e.x} \rightarrow \mathsf{e}'} \quad \text{(select)}$$

$$\frac{\vec{\mathsf{b}}_s \equiv \{\mathsf{x}\ \texttt{=}\ \texttt{(rec}\ \{\vec{\mathsf{b}}_1\ \texttt{/}\ \vec{\mathsf{b}}_2\}\texttt{).}\mathsf{x} \mid \mathsf{x} \in \mathrm{dom}\ (\vec{\mathsf{b}}_1 \cup \vec{\mathsf{b}}_2)\}}{\texttt{rec}\ \{\vec{\mathsf{b}}_1\ \texttt{/}\ \vec{\mathsf{b}}_2\} \rightarrow \{\vec{\mathsf{b}}_1[\vec{\mathsf{b}}_s]\texttt{;}\ \vec{\mathsf{b}}_2\}} \quad \text{(rec)}$$

$$\frac{\mathsf{x} \notin \mathrm{dom}\ \vec{\mathsf{b}}}{\texttt{let}\ \vec{\mathsf{b}}\ \texttt{in}\ \mathsf{e} \rightarrow \texttt{(rec}\ \{\vec{\mathsf{b}}; \mathsf{x} = \mathsf{e}\}\texttt{).}\mathsf{x}} \quad \text{(let)}$$

$$\frac{\mathsf{e}_1 \rightarrow^{*} \{\vec{\mathsf{b}}\}}{\texttt{with}\ \mathsf{e}_1\ \texttt{;}\ \mathsf{e}_2 \rightarrow \mathsf{e}_2[\vec{\mathsf{b}}]} \quad \text{(with)}$$

$$\frac{\mathsf{e}_1 \rightarrow^{*} \mathsf{x}\ \texttt{:}\ \mathsf{e}_3}{\mathsf{e}_1\ \mathsf{e}_2 \rightarrow \mathsf{e}_3[\mathsf{x}\ \texttt{=}\ \mathsf{e}_2]} \quad \text{(apply1)}$$

$$\frac{\mathsf{e}_1 \rightarrow^{*} \{\vec{\mathsf{x}}\}\ \texttt{:}\ \mathsf{e}_3 \quad \mathsf{e}_2 \rightarrow^{*} \{\vec{\mathsf{b}}\} \quad \vec{\mathsf{b}} \sim \vec{\mathsf{x}} \rightarrow \vec{\mathsf{b}}'}{\mathsf{e}_1\ \mathsf{e}_2 \rightarrow \mathsf{e}_3[\vec{\mathsf{b}}']} \quad \text{(apply2)}$$

Figure 6. Evaluation rules of the Nix expression language

$$\frac{\mathsf{x}\ \texttt{=}\ \mathsf{e} \in \vec{\mathsf{b}} \quad \vec{\mathsf{b}} \sim \vec{\mathsf{x}} \rightarrow \vec{\mathsf{b}}'}{\vec{\mathsf{b}} \sim \mathsf{x}, \vec{\mathsf{x}} \rightarrow \mathsf{b}\ ;\ \vec{\mathsf{b}}'} \quad \text{(matchnonempty)}$$

$$\frac{}{\vec{\mathsf{b}} \sim \varepsilon \rightarrow \varepsilon} \quad \text{(matchempty)}$$

Figure 7. Matching

package. The program is run in a very restricted build environment, but that environment is influenced by the other attributes in the attribute set passed to `derivation`: each attribute is converted into an environment variable of the same name. What exactly is passed depends on the type of the attribute:

Strings, URIs and natural numbers are passed verbatim. A path causes the referenced file to be copied to the store, and the corresponding store path is passed in the environment variable. That way, sources, configuration files, patches and other inputs to the build will all be part of the Nix store. Another derivation makes that derivation a dependency of the current derivation – the dependency is built first, and its output path is passed in the environment variable. Lists are flattened into space-separated strings. Finally, Booleans are passed as the empty string or 1.

4.2 Semantics

We give an operational semantics for the Nix expression language by providing evaluation rules. The rules have the form $\mathsf{e} \rightarrow \mathsf{e}'$, reducing one expression to another. They always apply to the complete Nix expression under consideration. In other words, reduction proceeds as long as the full Nix expression constitutes a redex. We write $\mathsf{e} \rightarrow^{*} \mathsf{e}'$ to denote that we can go from e to e' in many steps.

Next to syntactic transformation on Nix expressions, evaluation in Nix causes build actions to take place, such as described for the built-in function `derivation` above. Note that we see the build action as part of the evaluation and not as a side effect.

Listing all the reduction rules, especially for all the built-in functions and operators, would go beyond the space we have in this paper. We therefore give a few representative evaluation rules in Figure 6 that deal with functions and attribute sets.

Desugaring of inheritance For non-recursive attribute sets, the binding `inherit x` is syntactic sugar for a binding `x = x`. If a source is specified, for instance `inherit (e) x`, the desugaring is $\mathsf{x}\ \texttt{=}\ \mathsf{e.x}$. Multiple attributes can be inherited in a single statement. This is transformed into multiple bindings.

For recursive attribute sets, we have to be careful, because desugaring x : rec { inherit x; y = x; } to x : rec { x = x; y = x; } would incorrectly introduce infinite recursion. Therefore, recursive attribute sets internally separate recursive attributes from non-recursive (i.e., inherited) attributes (we write this $\texttt{rec}\ \{\vec{\mathsf{b}}_1\ \texttt{/}\ \vec{\mathsf{b}}_2\}$, and the above expression is desugared to x : rec { y = x; / x = x; } where the `x` in the first binding points to the attribute bound in the second binding, but the `x` on the right hand side of the second binding points to the lambda-bound `x`.

For the evaluation rules, we therefore assume that non-recursive attribute sets can be written in the form $\{\vec{\mathsf{b}}\}$ where each b is of the form $\mathsf{x}\ \texttt{=}\ \mathsf{e}$, and that recursive attribute sets are of the form $\texttt{rec}\ \{\vec{\mathsf{b}}_1\ \texttt{/}\ \vec{\mathsf{b}}_2\}$.

Operations on attribute sets The rule (select) describes how attribute selection is reduced: the expression e must evaluate to an attribute set containing bindings $\vec{\mathsf{b}}$. If there is a binding for the selected attribute x, the corresponding expression e' is returned.

The rule (rec) deals with unfolding a recursive attribute set. As explained above, only the bindings $\vec{\mathsf{b}}_1$ are actually considered to be recursive. In these bindings, all references x to names from the attribute set (written $\mathrm{dom}\ (\vec{\mathsf{b}}_1 \cup \vec{\mathsf{b}}_2)$) are substituted with their unfolding, i.e., a selection of the appropriate attribute x from the recursive set $\texttt{rec}\ \{\vec{\mathsf{b}}_1\ \texttt{/}\ \vec{\mathsf{b}}_2\}$ itself. We use a postfixed $[\vec{\mathsf{b}}]$ here to denote the simultaneous substitution where all free occurrences of the attributes bound in $\vec{\mathsf{b}}$ are replaced by their associated expressions. Note that the result of unfolding a recursive attribute set is always a non-recursive attribute set.

In rule (let), a let-statement can be reduced to a recursive attribute set where the body of the let is added as a fresh attribute. The body is then selected from that attribute set.

For a with-statement (with), we first reduce e_1 into an attribute set with bindings $\vec{\mathsf{b}}$. We then add the attributes defined in $\vec{\mathsf{b}}$ to the scope for evaluating e_2 by simply substituting these attributes in e_2.

Function application On encountering a function application, the function e_1 is reduced. There are two rules for function application, corresponding to the two forms of lambda abstraction in the syntax. If the function is of the form $\mathsf{x}\ \texttt{:}\ \mathsf{e}_3$, then rule (apply1) applies, and we substitute the argument e_2 for x in e_3.

If the function is defined via pattern matching on an attribute set (apply2), we also evaluate the argument e_2 to an attribute set with bindings $\vec{\mathsf{b}}$. We then match the bindings $\vec{\mathsf{b}}$ against the formals $\vec{\mathsf{x}}$. The matching results in another set of bindings $\vec{\mathsf{b}}'$ which we then use as a substitution for the body of the function e_3.

Rules for matching are shown in Figure 7. The rules (matchnonempty) and (matchempty) traverse the set of formals $\vec{\mathsf{x}}$. Every formal x is individually matched against the bindings. Matching thus succeeds if $\vec{\mathsf{x}}$ is a subset of $\mathrm{dom}\ \vec{\mathsf{b}}$ and computes the restriction $\vec{\mathsf{b}}$ to the attributes in $\vec{\mathsf{x}}$.

4.3 Design decisions

After having described the Nix expression language and its operational semantics, let us now point out three concepts that we believe are really essential to the language: purity, laziness, and the use of attribute sets as a prominent type.

While many aspects of the language (for instance, the fact that it is dynamically and not statically typed) could certainly be changed, the idea of functional package management crucially requires the language to be pure. Laziness and attribute sets both help significantly in making the system convenient to use.

Purity NixOS relies on the fact that a specific Nix expression has one associated value that does not depend on the state of the system. The Nix expression language is different from typical programming languages in that some of the values associated with expressions are actually directory trees with files in it, but that does not change the fact that one (and only one) value is associated with a Nix expression describing a derivation.

This referential transparency gives us the possibility to identify a package with the Nix expression that builds it. When we evaluate the same Nix expression multiple times – whether in a single computation or in different ones – we can reuse the previous result because it cannot have changed. We can even download precomputed results (i.e., precompiled binary packages) from other systems if desired. Purity also implies that different build actions performed at the same time do not influence each other, thus the Nix evaluator can easily schedule builds to run in parallel. Parallel builds are enabled by default, for instance, to distribute Nixpkgs builds on our compile farm among available machines and cores. By contrast, in Make (Feldman 1979), enabling parallel builds is not safe in general because build actions often interfere with each other.

Another aspect of purity is the integrity of the Nix store. Nix maintains full control over the references that packages in the Nix store have to other packages. All such references are fixed, i.e., they cannot be modified, and also the contents of the Nix store are immutable. This allows us to keep packages from randomly picking up untracked dependencies that might lead to unpredictable system failures. It also ensures that we can use garbage collection to identify unused packages and safely remove them.

Because purity is so central to NixOS, we discuss in Section 6.2 how we ensure that build actions are pure.

Laziness Evaluation of a derivation means performing a build action, and build actions can be quite expensive: some packages require a significant amount of sources to be downloaded from the internet, others take several hours to compile. It is therefore of utmost importance that a package is only built if it is necessary.

Only this allows us to write Nix expressions in a convenient style: Packages are described by Nix expressions and these Nix expressions can freely be passed around in a Nix program – as long as we do not access the contents of the package, no evaluation and thus no build will occur. For instance, the whole of the Nix packagages collection is essentially one attribute set where each attribute maps to one package contained in the collection. It would be unthinkable to require the entire collection of packages to be built if only one attribute was selected.

Moreover, Nix expressions typically contain more than just the plain build descriptions – they store meta-information about the packages as well, such as their version number, their homepage, or a description of the package. If we could not access that information without first building the package, the system would not be practicable to use.

Attribute sets Attribute sets are very convenient because they are so versatile. The most important feature that attribute sets provide us with are named arguments. For a language such as Nix, where most of the functions are first-order functions taking a possibly very large number of dependencies to a resulting package, it would be extremely inconvenient if all the arguments had to be specified in a specific order. Package dependencies do not usually have a natural total order, and as a Nix expression evolves, the number of dependencies often changes. Being able to refer to dependencies by name is therefore nearly a must.

Furthermore, attribute sets can be used to simulate a whole number of other powerful language concepts easily: for example, they can be used as both modules and objects, without having to add a huge number of additional language concepts.

Attribute sets also give us a form of subtyping. We can, for instance, bundle together a number of packages in a set and pass it to a function that expects only a subset of those packages.

```
pkgs.writeText "ssh_config" ''
  SendEnv LANG LC_ALL ...
  ${if config.services.sshd.forwardX11 then ''
    ForwardX11 yes
    XAuthLocation ${pkgs.xorg.xauth}/bin/xauth
  '' else ''
    ForwardX11 no
  ''}
''
```

Figure 8. Nix expression to build ssh_config

The reader may wonder if the Nix expression language could have been implemented as an embedded DSL (e.g., in Haskell). For instance, Sloane (2002) embedded a similar build language (based on Odin) in Haskell. This is certainly possible, but it would be painful to use: for instance, features such as string interpolation or functions over attribute sets would not be available.

5. NixOS

In Section 3 we saw that the purely functional approach of the Nix package manager solves many problems that plague "imperative" package management systems. We have previously used Nix as a package manager under existing Linux distributions and other operating systems, such as Mac OS X, FreeBSD and Windows to deploy software alongside the "native" package managers of those systems. In this section we describe NixOS, a Linux distribution entirely built on Nix. It uses Nix not just for package management but also to build all other static parts of the system. This extension follows naturally: while Nix derivations typically build whole packages in an atomic action, they can build any file or directory tree so long as it can be built in a pure way. For instance, most configuration files in /etc in a typical Unix system do not change dynamically at runtime and can therefore be built by a derivation.

For example, the expression in Figure 8 generates a simple configuration file (ssh_config) for the SSH client program in the Nix store (i.e., /nix/store/*hash*-ssh_config). The helper function pkgs.writeText returns a derivation that builds a single file in the Nix store with the specified name and contents. The contents in this case is a string containing an interpolation that return different file fragments depending on configuration options set by the user (discussed below). Namely, if the option services.sshd.forwardX11 is enabled, the SSH option ForwardX11 should be set to yes, and furthermore SSH must be able to find the xauth program (to forward X11 authentication credentials to the remote machine). This means that this configuration file *has an optional dependency on the* xauth *package*: depending on a user configuration option, the configuration file either does or doesn't reference xauth. This kind of dependency between a configuration file and a software package is generally not handled by conventional package managers, since they don't deal with configuration files in the same formalism as packages: packages can have dependencies, but configuration files cannot. In Nix, they are all derivations and therefore treated in the same way. As a concrete consequence, the xauth package won't be garbage-collected if ssh_config refers to it — an important constraint on the integrity of the system.

When we build this expression, a possible result might be

```
SendEnv LANG LC_ALL ...
ForwardX11 yes
XAuthLocation /nix/store/j1gcgw7a...-xauth-1.0.2/bin/xauth
```

It is worth noting that due to laziness, xauth will be built if and only if config.services.sshd.forwardX11 is true. Thus, the *laziness of the Nix expression language directly translates to laziness in building packages*. This is a crucial feature: while xauth is a tiny package, many NixOS configuration options trigger huge dependencies. For instance, the option services.xserver.enable = true will cause a dependency on the X11 server (large), while services.xserver.sessionType = "kde" will bring in the K Desktop Environment (very large).

Building a system, declaratively NixOS thus consists of a set of Nix expressions that return derivations that build the various parts that constitute a Linux system: static configuration files, boot scripts, and so on. These build upon the software packages already provided by Nixpkgs. Figure 9 shows a graph of a small subset of the derivations that build NixOS. Italic nodes denote derivations that build configuration files; bold nodes build Upstart service descriptions (see below); dotted nodes are scripts; dashed nodes are various helper derivations that compose logical parts of the system; and all other nodes are software packages. However, this distinction is entirely conceptual. As far as Nix is concerned, they are all derivations: pure, atomic build actions.

Among the points of interest in the graph are the following:

- The Linux kernel (kernel), along with external kernel modules (drivers and other kernel extensions in the Linux model) such as the NVIDIA graphics drivers (nvidiaDrivers) and the Intel Wireless drivers (iwlwifi) that depend on it. These are part of Nixpkgs. A very nice consequence of the purely functional approach is that an upgrade of the kernel (i.e., a change to the Nix expression that builds the kernel) will trigger a rebuild of all dependent external kernel modules. This is in contrast to many other Linux distributions, where a common scenario is that one upgrades the kernel and then discovers that the X server will not start anymore because the NVIDIA drivers were not rebuilt. This is not possible here. (Of course the external modules might not be compatible with the new kernel, but such a problem usually manifests itself at build time; and one cannot activate a new configuration unless *all* of it builds successfully.)

  A small wrapper component (modulesTree) combines (through symlinks) the kernel modules from the kernel package with those from the external module packages in a single directory tree. This is necessary because the kernel module loading command modprobe expects all modules to live in a single directory tree. This is an example of how we circumvent the impure models of various tools and applications: modprobe espouses an impure model where modules from various packages are statefully installed in an existing directory. modulesTree turns this into a purely functional discipline.

- The derivation initrd builds the *initial ramdisk* necessary for booting the system. NixOS, as most modern Linux distributions, has the following boot process. The machine's BIOS loads the boot loader Grub (http://www.gnu.org/software/grub/). Grub then lets the user choose the operating system to boot, and in the case of Linux, loads the kernel and the initial ramdisk. The latter contains a small root filesystem containing everything necessary to allow the real filesystem to be mounted. Notably, it must contain any kernel modules for the hardware containing the filesystem (e.g. drivers for SCSI or USB), the filesystem itself (e.g. ext3) and perhaps other modules such as network drivers for remote booting. The initrd thus is not fixed, but depends on the hardware configuration of the user. It therefore contains a copy of the closure under the module dependency relation of the set of modules needed (e.g. ["ext3" "ata_piix" "sd_mod" "sd_mod"] for a Dell Latitude D630).

  Furthermore, the initial ramdisk contains a boot script called stage1Init, which loads the kernel modules, runs the filesystem checker fsck if needed, mounts the root filesystem, and passes control to the final boot script stage2Init.

```
{ntp, modprobe, glibc, writeText, servers}: [12]

let
  stateDir = "/var/lib/ntp";
  ntpUser = "ntp";
  config = writeText "ntp.conf" '' [13]
    driftfile ${stateDir}/ntp.drift
    ${toString (map (srv: "server $srv\n") servers)}
  '';
in

{ [14]
  name = "ntpd";

  users = [ [15]
    { name = ntpUser;
      uid = (import ../system/ids.nix).uids.ntp;
      description = "NTP daemon user";
    }
  ];

  job = '' [16]
    description "NTP daemon"

    start on ip-up
    stop on ip-down

    start script [17]
        mkdir -m 0755 -p ${stateDir}
        chown ${ntpUser} ${stateDir}
        # Needed to run ntpd as an unprivileged user.
        ${modprobe}/sbin/modprobe capability || true
    end script

    respawn ${ntp}/bin/ntpd -n -c ${config} [18] \
        -u ${ntpUser}:nogroup -i ${stateDir}
  '';
}
```

Figure 10. ntp.nix: Nix expression that generates the NTP Upstart job

- A running system consists primarily of a set of *services*, such as the X server, the DHCP client, the SSH daemon, and many more. On NixOS, these are started and monitored by Upstart (http://upstart.ubuntu.com/), which provides the init process that starts all other processes. Upstart reads specifications of system services from /etc/event.d and runs the commands for each service when appropriate. Services can have dependencies on eath other, e.g., the SSH daemon should be started when networking comes up.

  The Upstart service descriptions in /etc/event.d are built by Nix expressions. Figure 10 shows the expression that builds the service that runs the Network Time Protocol (NTP) client, which synchronises the clock against those of a set of NTP servers. It is a function $\boxed{12}$ that takes as inputs the software dependencies (such as the ntp package) and a configuration option servers specifying the hostnames or IP addresses of the NTP servers to use. It returns an attribute set containing various properties of the service, such as its name, a set of user accounts $\boxed{15}$ that must be created for this service (by the activation script, discussed below), and the text of the Upstart service in the format expected by Upstart $\boxed{16}$.

  There are two points of interest in the NTP service that are representative of NixOS in general. First, it is self-initialising and idempotent: at startup $\boxed{17}$, it creates the environment needed by the ntpd daemon, such as the directory /var/lib/ntp. This

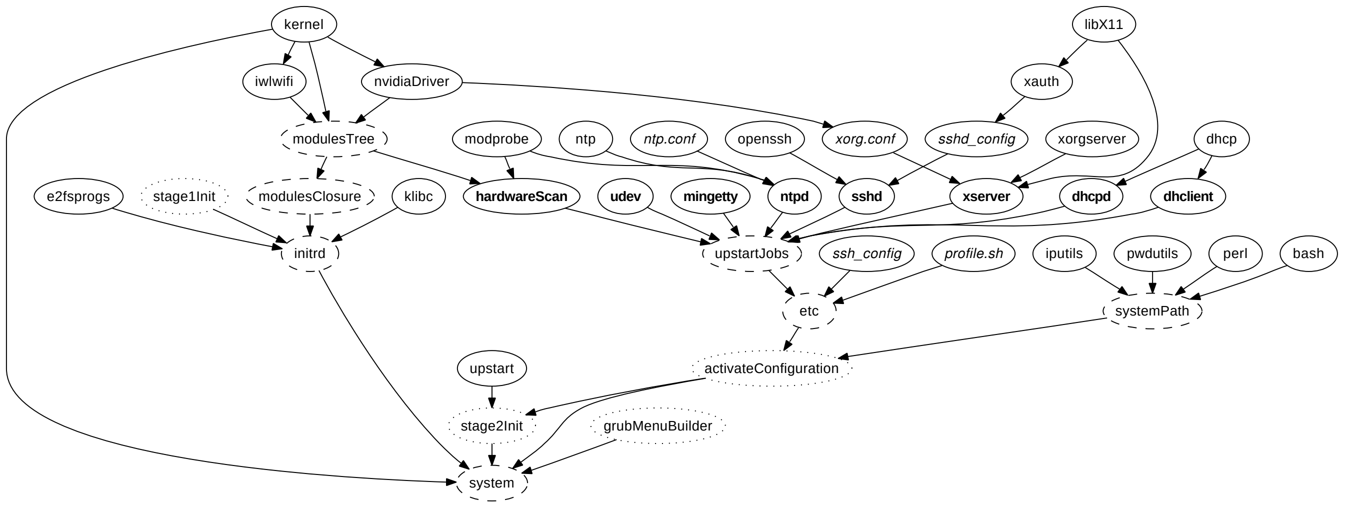


Figure 9. A small part of the dependency graph of the derivations that build NixOS

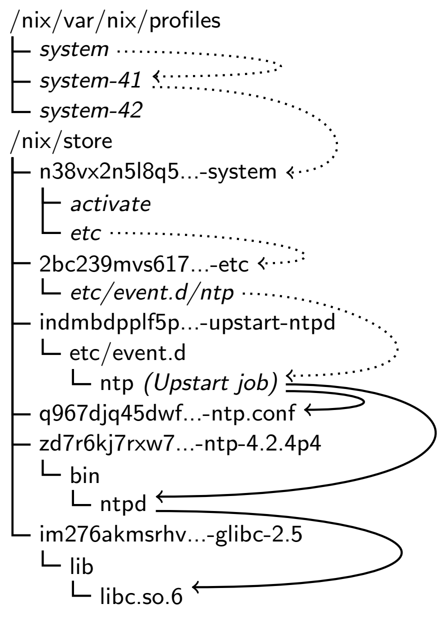

```
/nix/var/nix/profiles
├ system
├ system-41
└ system-42
/nix/store
├ n38vx2n5l8q5...-system
│ ├ activate
│ └ etc
├ 2bc239mvs617...-etc
│ └ etc/event.d/ntp
├ indmbdpplf5p...-upstart-ntpd
│ └ etc/event.d
│   └ ntp (Upstart job)
├ q967djq45dwf...-ntp.conf
├ zd7r6kj7rxw7...-ntp-4.2.4p4
│ └ bin
│   └ ntpd
└ im276akmsrhv...-glibc-2.5
  └ lib
    └ libc.so.6
```

Figure 11. Outputs of derivations related to the NTP service, as well as the system profile

obviates the need for post-install scripts used in other package managers and allows switching between configurations.

Second, the configuration file for the daemon is not stored in /etc/ntp.conf, but in the Nix store, built by the writeText helper function $\boxed{13}$. The path of the generated ntp.conf in the store is directly substituted in the generated Upstart service file through the string interpolation at $\boxed{18}$. While this is not a big deal for the NTP service, it is very important for other services: for instance, it makes it trivial to run multiple web servers side-by-side, such as production and test instances, simply by instantiating the appropriate function multiple times with different arguments. The use of "global variables" such as /etc/ntp.conf would preclude this. Figure 11 shows the outputs of the derivations for the NTP service.

- By contrast to ntp.conf, some files *are* still needed in /etc, mostly because they are "cross-cutting" configuration files that cannot be feasibly passed as build-time dependencies to every derivation. Some constitute *mutable state* that aren't dealt with in the functional model at all, such as the system password file /etc/passwd or the DNS configuration file /etc/resolv.conf, which must be modified at runtime. However, others are static (only change as a result of explicit reconfigurations) but still crosscutting, such as the static host name resolution list /etc/hosts. These are built by Nix expressions and stored in the Nix store. The etc derivation in Figure 9 takes as inputs the files that are needed in /etc and combines them into a tree of symlinks. Later, the activation script (described below) copies these symlinks into /etc.
- systemPath is a set of symlinks to selected packages and is placed in the users' PATH environment variable. Since it is constructed by a Nix expression, it enables declarative package management. By contrast, manual package management actions like nix-env -i (Section 3) are stateful and don't use a declarative specification to define the set of installed packages.
- Building a NixOS configuration is entirely pure; it only computes values (files and directories) in the Nix store. But to *activate* a configuration requires actions to be performed; system services must be started or stopped, user accounts for system services must be created, symlinks in /etc must be created (see above), and so on. This is done by the *activation script* (built by derivation activateConfiguration).
- The stage 2 initialisation script (stage2Init) performs boot-time initialisation and runs the activation script.
- Finally, the system derivation itself simply creates symlinks to its inputs, e.g. $out/kernel links to the kernel image.

Thus, given the top-level Nix expression for NixOS (installed in /etc/nixos/nixos/default.nix), the commands

```
$ nix-build /etc/nixos/nixos -A system
$ ./result/activate
```

build the entire configuration, including all software dependencies, configuration files, Upstart jobs, etc., leaving a symlink result in the current directory; and then run the activation script. When switching to the new configuration, the activation script stops any Upstart

```
{
  boot = {
    grubDevice = "/dev/sda";
    kernelModules = ["fuse" "kvm-intel"];
  };
  fileSystems = [
    { mountPoint = "/";
      device = "/dev/sda1";
    }
  ];
  services = {
    sshd = {
      enable = true;
      forwardX11 = true;
    };
    xserver = {
      enable = true;
      videoDriver = "nvidia";
      sessionType = "kde";
    };
  };
}
```

Figure 12. configuration.nix: The NixOS system configuration specification

services that have disappeared in the new configuration, starts new services, restarts services that have changed, and leaves all other services untouched. It can determine precisely which services have changed by comparing the store paths of the Upstart service files: if they are different, then by definition some input in the dependency graph of the service changed, and a restart may be needed.

Using NixOS Of course, we cannot expect the average user to reconfigure her system by editing the various Nix expressions in Figure 9, then rebuilding the configuration. Instead, NixOS has a single configuration file, /etc/nixos/configuration.nix, that contains a nested attribute set specifying all configuration pertaining to the user's system. An example of this configuration file is shown in Figure 12. This expression is passed as an argument to the top-level NixOS expression, /etc/nixos/nixos/default.nix, which distributes its attributes to the appropriate subexpressions. For instance, the attribute services.sshd.forwardX11, seen above, is distributed to the function that builds the file ssh_config.

The configuration of the system is updated by editing configuration.nix and then running

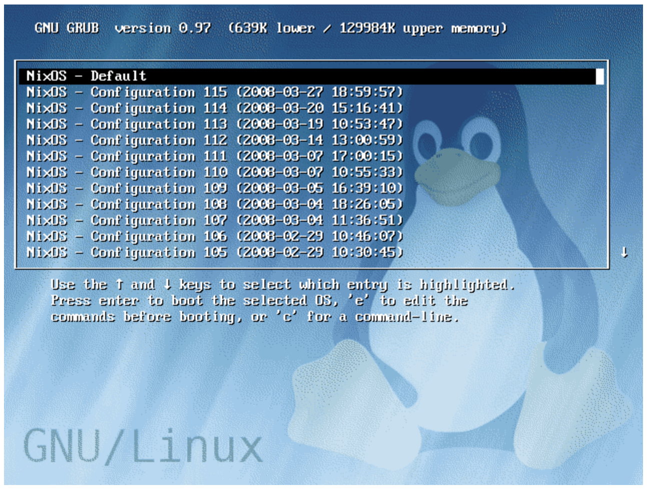
```
$ nixos-rebuild switch
```

which builds the system attribute of the top-level NixOS expression, installs it in the profile /nix/var/nix/profiles/system, and then runs its activation script. The profile (see Section 3) is used to enable rollbacks to previous configurations. Figure 11 shows the symlinks from the profile to the configuration in the Nix store.

Alternatively, one can run the command nixos-rebuild test. This will build and activate the configuration without installing it in the profile. Thus, rebooting the system will automatically cause a rollback to the current configuration in the system profile.

The system profile is used to generate the boot menu of the Grub boot loader. Each non-garbage-collected configuration is made available as a menu entry (see Figure 13). This allows the user to recover trivially from bad upgrades (such as those that render the system unbootable). A system can be rolled back without rebooting by doing nix-env --rollback -p /nix/var/nix/profiles/system and running the activation script.

NixOS itself is currently upgraded simply by updating the Nix expressions in /etc/nixos from the Nix Subversion repository, i.e. by doing svn up /etc/nixos/*.



```
GNU GRUB  version 0.97  (639K lower / 129984K upper memory)

NixOS - Default
NixOS - Configuration 115 (2008-03-27 18:59:57)
NixOS - Configuration 114 (2008-03-20 15:16:41)
NixOS - Configuration 113 (2008-03-19 10:53:47)
NixOS - Configuration 112 (2008-03-14 13:00:59)
NixOS - Configuration 111 (2008-03-07 17:00:15)
NixOS - Configuration 110 (2008-03-07 10:55:33)
NixOS - Configuration 109 (2008-03-05 16:39:10)
NixOS - Configuration 108 (2008-03-04 18:26:05)
NixOS - Configuration 107 (2008-03-04 11:36:51)
NixOS - Configuration 106 (2008-02-29 10:46:07)
NixOS - Configuration 105 (2008-02-29 10:30:45)        ↓

   Use the ↑ and ↓ keys to select which entry is highlighted.
   Press enter to boot the selected OS, 'e' to edit the
   commands before booting, or 'c' for a command-line.
```

Figure 13. The Grub boot menu for a NixOS machine

6. Evaluation

In this section, we reflect upon the extent to which the purely functional model "works", i.e., can be used to implement a useful system, to what extent Nix derivations are pure, and to what extent we need to compromise on purity.

6.1 Status

NixOS is not a proof-of-concept: it is in production use on a number of desktop and server machines. As of March 2008, the Nix Packages collection (on which NixOS builds) contains Nix expressions for some 1350 packages. These range from basic components such as the C library Glibc and the C compiler GCC to end-user applications such as Firefox and OpenOffice.org. NixOS has system services for running X11 (including the 3D window manager Compiz, KDE and parts of Gnome), Apache, PostgreSQL and many more. NixOS is fairly unique among Linux distributions in that it allows non-root users to install software, thanks in part to the purely functional approach, which enables some strong security guarantees (Dolstra 2005, Section 3).

To provide some sense of the size of a typical configuration (a laptop running X11, KDE and Apache, among other things[1]): the build graph rooted at the top-level system attribute in /etc/nixos/nixos/default.nix consists of 707 derivations (408 excluding fetchurl derivations) and 196 miscellaneous source files. The closure of the output of the system derivation (i.e., its runtime dependencies) consists of 340 store paths with a total size of 1053 MiB. Thus 367 of the 707 derivations are build-time-only dependencies, such as source distributions, compilers and parser generators. The evaluation of system imports 290 Nix expression files in the NixOS and Nixpkgs trees, and takes 2.4 seconds of CPU time on a Core 2 Duo 7700.

Sources and ISO images of NixOS for i686 and x86_64 platforms are available at http://nixos.org/.

6.2 Purity

The goal of NixOS was to create a Linux distribution built and configured in a purely functional way. Thus build actions should be deterministic and therefore reproducible, and there should be no "global variables" like /bin that prevent multiple versions of pack-

[1] Revision 11437 of the configuration at https://svn.cs.uu.nl:12443/repos/trace/configurations/trunk/misc/eelco-dutibo.nix.

ages and services to exist side-by-side. There are several aspects to evaluating the extent to which we reached those goals.

Software packages Nix has no /bin, /usr, /lib, /opt or other "stateful" directories containing software packages, with a single exception: there is a symlink /bin/sh to an instance of the Bash shell in the Nix store. This symlink is created by the activation script. /bin/sh is needed because very many shell scripts and commands refer directly to it; indeed, the C library function system() has a hard-coded reference to /bin/sh. To our surprise, /bin/sh is the *only* such compromise that we need in NixOS. Other hard-coded paths in packages (e.g., references to /bin/rm or /usr/bin/perl) are much less common and can easily be patched on a per-package basis. Such paths are uncommon in widely used software because they are not portable in any case (e.g., Perl is typically, but not always installed in /usr/bin/perl). They are relatively more common in Linux-specific packages that we needed to add to Nixpkgs to build NixOS.

An interesting class of packages to support are binary-only packages, such as Adobe Reader and many games. While Nix is primarily a source-based deployment system (with sharing of pre-built binaries as a transparent optimisation, as discussed in Section 3), binary packages can be supported easily: they just have a trivial build action that unpacks the binary distribution to $out. However, such binaries won't work as-is under NixOS, because ELF binaries (which Linux uses) contain a hard-coded path to the dynamic linker used to load the binary (usually /lib/ld-linux.so.2 on the i386 platform), and expect to find dependencies in /lib and /usr/lib. None of these exist on NixOS for purity reasons. To support these programs, we developed a small utility, patchelf, that can change the dynamic linker and RPATH (runtime library search path) fields embedded in executables. Thus, the derivation that builds Adobe Reader uses patchelf to set the acroread program's dynamic linker to /nix/store/...-glibc-.../lib/ld-linux.so.2 and its RPATH to the store paths of GTK and other needed libraries passed as function arguments to the derivation.

Configuration data NixOS has many fewer configuration files in /etc than other Linux distributions. This is because most configuration files concern only a single daemon, which almost always has an option to specify the full path to the configuration file in the Nix store directly (such as ntpd -c ${config} in Figure 10). What remains is cross-cutting configuration files, which, as discussed in Section 5, are built purely but then symlinked in /etc by the configuration's activation script. The configuration above has just 39 such symlinks.

Mutable state NixOS does not have any mechanism to deal directly with mutable state, such as the contents of /var. These are managed by the activation script and the system services in a standard, stateful way. Of course, this is to be expected: the *running* of a system (as opposed to the configuration) is inherently stateful.

Runtime dependencies In Nix, we generally try to *fix runtime dependencies at build time.* This means that while a program may execute other programs or load dynamic libraries at runtime, the paths to those dependencies are hard-coded into the program at build time. For instance, for ELF executables, we set the RPATH in the executable such that it will find a statically determined set of library dependencies at runtime, rather than using a dynamic mechanism such as the LD_LIBRARY_PATH environment variable to look up libraries. This is important, because the use of such dynamic mechanisms makes it harder to run applications with conflicting dependencies at the same time (e.g., we might need Firefox linked against GTK 2.8 and Thunderbird linked against GTK 2.10). It also enhances determinism: a program will not suddenly behave differently on another system or under another user account because environment variables happen to be different.

However, there is one case in NixOS and Nixpkgs of a library dependency that *must* be overridable at runtime and cannot be fixed statically: the implementation of OpenGL to be used at runtime (libGL.so), which is hardware-specific. We build applications that need OpenGL against Mesa, but add the impure (stateful) path /var/run/opengl-driver to the RPATH. The activation script symlinks that path to the actual OpenGL implementation selected by the configuration (e.g., nvidiaDriver) to allow programs to use it.

Build actions The Nix *model* is that derivations are pure, that is, two builds of an identical derivation should produce the same result in the Nix store. However, in contemporary operating systems, there is no way to actually enforce that model. Builders can use any impure source of information to produce the output, such as the system time, data downloaded from the network, or the current number of processes in the system as seen in /proc. It is trivial to construct a contrived builder that does such things. But build processes generally do not, and instead are fairly deterministic; impure influences such as the system time generally do not affect the runtime behaviour of the package in question.

There are however frequent exceptions. First, many build processes are greatly affected by environment variables, such as PATH or CFLAGS. Therefore we clear the environment before starting a build (except for the attributes declared by the derivation, of course). We set the HOME environment variable to a non-existent directory, because some derivations (such as Qt) try to read settings from the user's home directory.

Second, almost all packages look for dependencies in impure locations such as /usr/bin and /usr/include. Indeed, the undeclared dependencies caused by this behaviour are what motivated Nix in the first place: by storing packages in isolation from each other, we prevent undeclared build-time dependencies. In five years we haven't had a single instance of a package having an undeclared build-time dependency on another package *in the Nix store*, or having a runtime dependency on another package in the Nix store not detected by the reference scanner. However, with Nix under other Linux distributions or operating systems, there have been numerous instances of packages affected by paths outside the Nix store. We prevent most of those impurities through a wrapper script around GCC and ld that ignores or fails on paths outside of the store. However, this cannot prevent undeclared dependencies such as direct calls to other programs, e.g., a Makefile running /usr/bin/yacc.

Since NixOS has no /bin, /usr and /lib, the effect of such impurities is greatly reduced. However, even in NixOS such impurities can occur. For instance, we recently encountered a problem with the build of the dbus package, which failed when /var/run/dbus didn't exist.

As a final example of impurity, some packages try to install files under a different location than $out. Nix causes such packages to *fail deterministically* by executing builders under unprivileged UIDs that do not have write permission to other store paths than $out, let alone paths such as /bin. These packages must then be patched to make them well-behaved.

To ascertain how well these measures work in preventing impurities in NixOS, we performed two builds of the Nixpkgs collection[2] on two different NixOS machines. This consisted of building 485 non-fetchurl derivations. The output consisted of 165927 files and directories. Of these, there was only one *file name* that differed between the two builds, namely in mono-1.1.4: a directory gac/IBM.Data.DB2/1.0.3008.37160__7c307b91aa13d208 versus 1.0.3008.40191__7c307b91aa13d208. The differing number is likely derived from the system time.

[2] To be precise, the i686-linux derivations from build-for-release.nix in revision 11312 of https://svn.cs.uu.nl:12443/repos/trace/nixpkgs/branches/purity-test

We then compared the contents of each file. There were differences in 5059 files, or 3.4% of all regular files. We inspected the nature of the differences: almost all were caused by timestamps being encoded in files, such as in Unix object file archives or compiled Python code. 1048 compiled Emacs Lisp files differed because the hostname of the build machines were stored in the output. Filtering out these and other file types that are known to contain timestamps, we were left with 644 files, or 0.4%. However, most of these differences (mostly in executables and libraries) are likely to be due to timestamps as well (such as a build process inserting the build time in a C string). This hypothesis is strongly supported by the fact that of those, only 42 (or 0.03%) had different file sizes. None of these content differences have ever caused an observable difference in behaviour.

7. Related work

This paper is about purely functional *configuration management* of operating systems. It is not about implementing an operating system in a (purely) functional language. Hallgren et al. (2005) did the latter in Haskell in their operating system *House*, and a number of systems have been implemented in impure functional languages.

DeTreville (2005) proposed making system configuration declarative. NixOS is a concrete, large-scale realisation of that notion. Beshers et al. (2007) discuss a Linux distribution that not only uses functional programming to implement various system administration tools, but applies a functional mindset to those tasks. Notably, the autobuilder tool builds binary packages for the Linspire Linux distribution in a purely functional way: from a set of source packages it builds immutable binary packages. However, this is not used for the actual package management on end-user systems, nor does it extend to building complete system configurations.

Cfengine (Burgess 1995) is a well-known system configuration management tool. Cfengine updates machines on the basis of a declarative specification of actions (such as "a machine of class X must have the following line in /etc/hosts"). However, these actions transform a possibly unknown state of the system, and can therefore have all the problems of statefulness.

8. Conclusion

We demonstrated that a realistic operating system can be built and configured in a declarative, purely functional way with very few compromises to purity.

Future work We are investigating how to extend the Nix expression language with a type system. In the long term, we do not just want to check as much as possible of the current structural type system statically, but also introduce user-defined datatypes, such that for instance all variants of a single package such as GHC have the same type, and one cannot inadvertently pass a different package as a parameter where really GHC is expected. Currently, such a package fails at evaluation time, which is not much of a problem in the case of important dependencies, because the error will most likely occur prior to distribution. However, since Nix encourages to write descriptions with many parameters such as optional dependencies, it is impossible to test all possible configurations in advance.

Given that we can specify and build configurations for single machines declaratively, the logical next step is to extend our approach to sets of machines, so that the configuration of a network of machines can be specified centrally. This will allow interdependencies between machines (such as a database server on one machine and a front-end webserver on another) to be expressed elegantly. Furthermore, not all machines need to be physical machines: given a declarative specification, it is possible to automatically *instantiate virtual machines* that implement the specification. Apart from easing the deployment of virtual machines, this will enable simulation and debugging of distributed deployments.

Finally, as noted in Section 6.2, our model assumes that builds are pure, but current operating systems cannot enforce this. An interesting idea would be to add support for truly pure builds, e.g., kernel modifications to support the notion of a "pure process": one that is guaranteed to give the same output for some set of inputs. This would mean, for instance, that the time() system call must return a fake value, network access is blocked, files outside of a specified set are invisible, and so on.

Acknowledgments This research was supported in part by NWO/-JACQUARD project 638.001.201, *TraCE: Transparent Configuration Environments*. We wish to thank all contributors to Nixpkgs and NixOS.

References

Rick Anderson. The end of DLL hell. MSDN, http://msdn2.microsoft.com/en-us/library/ms811694.aspx, January 2000.

Clifford Beshers, David Fox, and Jeremy Shaw. Experience report: using functional programming to manage a Linux distribution. In *ICFP'07: Proc. 2007 ACM SIGPLAN Intl. Conf. on Functional Programming*, pages 213–218, New York, NY, USA, 2007. ACM.

Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, number 28/6 in SIGPLAN Notices, pages 197–206, June 1993.

Mark Burgess. Cfengine: a site configuration engine. *Computing Systems*, 8(3), 1995.

John DeTreville. Making system configuration more declarative. In *HotOS X, Tenth Workshop on Hot Topics in Operating Systems*. USENIX, June 2005.

Eelco Dolstra. Secure sharing between untrusted users in a transparent source/binary deployment model. In *20th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE 2005)*, pages 154–163, Long Beach, California, USA, November 2005.

Eelco Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, Utrecht University, The Netherlands, 2006.

Eelco Dolstra and Armijn Hemel. Purely functional system configuration management. In *11th Workshop on Hot Topics in Operating Systems (HotOS XI)*. USENIX, May 2007.

Eelco Dolstra, Eelco Visser, and Merijn de Jonge. Imposing a memory management discipline on software deployment. In *Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.

Stuart I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–65, 1979.

Eric Foster-Johnson. *Red Hat RPM Guide*. John Wiley & Sons, 2003. Also at http://fedora.redhat.com/docs/drafts/rpm-guide-en/.

Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Tenth ACM SIGPLAN Intl. Conf. on Functional Programming*, pages 116–128. ACM Press, 2005.

Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

Anthony M. Sloane. Post-design domain-specific language embedding: A case study in the software engineering domain. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

TIS Committee. Tool Interface Specification (TIS) Executable and Linking Format (ELF) Specification, Version 1.2, May 1995.