JFP **20** (5 & 6): 577–615, 2011. © Cambridge University Press 2010
doi:10.1017/S0956796810000195 First published online 15 October 2010

NixOS: A purely functional Linux distribution

EELCO DOLSTRA
Department of Software Technology, Delft University of Technology, Postbus 5031, 2600 GA Delft, The Netherlands
(*e-mail:* e.dolstra@tudelft.nl)

ANDRES LÖH
Department of Information and Computing Sciences, Utrecht University, Postbus 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* andres@cs.uu.nl)

NICOLAS PIERRON
EPITA Research and Development Laboratory, 14-16 rue Voltaire, 94276 Le Kremlin-Bicêtre cedex, France
(*e-mail:* nicolas.b.pierron@gmail.com)

Abstract

Existing package and system configuration management tools suffer from an *imperative model*, where system administration actions such as package upgrades 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 deploy 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 analogous to a heap in a purely functional 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 modular, 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: "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 that specification. This allows a system to be built deterministically, and therefore reproducibly. It allows the user to roll back the system to previous configurations, since these 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 Red Hat Package Manager (RPM) (Foster-Johnson 2003). This concept was implemented in the Nix package manager (Dolstra *et al.* 2004; Dolstra 2006), summarised in Section 3. Nix uses functional programming principles at two levels: it has a purely functional build specification language that relies heavily on laziness; and it stores packages as immutable values in what is essentially a purely functional data structure.

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 immutable paths such as

/nix/store/c6gddm8rkc05w6pmijrwp6zb76ph3prs-firefox-3.5.7

that include a cryptographic hash of all inputs involved in building it. On NixOS, there are no standard, "stateful" Unix system directories such as /usr or /lib, and there is only a 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 rolled back trivially. 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.

In Section 2, we argue that many of the problems in existing system configuration management tools are a result of statefulness. In Section 3, we give an overview of our previous work on the purely functional Nix package manager. Then we come to the contributions of this paper:

- We show how a full-featured Linux distribution, NixOS, can be built and configured in a declarative way using principles borrowed from purely functional

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

languages, giving rise to properties such as atomicity of upgrades and the ability to roll back (Section 4).
- We present a *module system* for NixOS that allows separation of concerns and convenient extensibility (Section 5).
- We measure the extent to which NixOS and Nix build actions are pure, discuss the compromises to purity that were needed to build NixOS, and reflect on the strengths and weaknesses of the purely functional model (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. For instance, most Unix package managers, such as the 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. Upgrades or uninstallations cannot easily be rolled back unless the package manager makes a backup of the overwritten files. 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 – that is, it contains a path to a file in another 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 one application might trigger an upgrade of the Perl interpreter in /usr/bin/perl, which might cause other applications to break. This is known as "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. Imagine that we have a running Apache web server and we wish to test a new version (without interfering with the currently running server). To do this, we install the new version in a different location from the old version. However, to then run it, we need to clone the configuration file httpd.conf because it contains the path to the Apache installation; and then we need to clone the Apache start script because it contains the paths of Apache and the configuration file. Thus the change to one part of the dependency graph of packages and configuration files ripples upwards, requiring manual changes to other nodes in the graph.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

Furthermore, stateful configuration changes such as upgrades are not *atomic* (or *transactional*). Since most packages consist of multiple files, an upgrade entails overwriting each file in sequence. This means that during the upgrade, the package is in an inconsistent state: some files belong to the old version, and some to the new. If the user tries to run the package during this time window, the results are undefined; it may even crash. Also, if the upgrade is interrupted for whatever reason, the package may be left in a permanently broken state. This extends to the level of upgrading entire systems. For instance, a power failure during a major upgrade of a Linux or Mac OS X installation is likely to leave the system in a broken, perhaps unbootable state.

Similarly, building a package is a stateful operation. In RPM, for instance, 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 $\geqslant$ 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. 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. (See Hart & D'Amelia 2002 for additional problems with RPM, such as sensitivity to the order in which packages are installed.)

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; the user can be presented with the delta between the original and modified version and asked to manually re-apply the changes to the new version; or a *post-install script* (delivered as part of the package's meta-data) can attempt to merge the changes automatically. 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

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

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 all of the problems. 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 file, and now wish to undo the upgrade). Furthermore, changes to configuration files are often *effected* in very different ways: a change to the Unix filesystem table /etc/fstab might be effected by running mount -a to mount any new filesystems added to that file, while a change to the Apache 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 the 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 tends to be much more likely to cause errors than doing a clean reinstall, precisely because the upgrade depends on the previous state of the system, while a clean reinstall has no such dependency.
- Upgrades are not *atomic*, which makes them inherently dangerous operations.

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 (Hudak 1989) such as Haskell (Peyton Jones 2004). 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.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
expressions
e  ::= x                        identifier
     | nat | str                literal
     | [e*]                     list
     | rec? {b*}                (optionally recursive) attribute set
     | let b* in e              local declarations
     | e.x                      attribute selection
     | x : e                    plain λ-abstraction
     | {fs?} : e                λ-abstraction pattern-matching an attribute set
     | e e                      function application
     | if e then e else e       conditional
     | with e1; e2              add attributes from set e1 to lexical scope of e2
     | (e)                      grouping

bindings
b  ::= ap = e;                  allows concise nested attribute sets, e.g. x.y.z = true
     | inherit x*;              copy value of attribute x from lexical scope

attribute path
ap ::= x.ap | x

formals
fs ::= x, fs | x | ...          "..." means ignore unmatched attributes

string literals
str ::= "(char | ${e})*"        strings; interpolation using ${e}
     | ''(char | ${e})*''       strings with common indentation removed
     | uri                      Uniform Resource Identifiers (Berners-Lee et al., 1998)
```

Fig. 1. Informal partial syntax overview of the Nix expression language.

Nix expressions Packages are built from *Nix expressions*, a dynamically typed, lazy, purely functional language. (For convenience, Figure 1 summarises the language constructs used in this paper. These are described below.) 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 (source files or other derivations). A package is built by recursively building the dependencies, then invoking the build script with the given environment variables.

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

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
{ stdenv, fetchurl, ghc, X11, xmessage }: 1

stdenv.mkDerivation 2 (rec {
  name = "xmonad-0.5";

  src = fetchurl {
    url = "http://hackage.haskell.org/.../${name}.tar.gz";
    sha256 = "1i74az7w7nbirw6n6lcm44vf05hjq1yyhnsssc779yh0n00lbk6g";
  };

  buildInputs = [ ghc X11 ]; 3

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

  buildPhase = ''
    ./Setup build
  '';

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

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

Fig. 2. xmonad.nix: Nix expression for xmonad.

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" code. 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*. The function mkDerivation captures this commonality, allowing packages to be written succinctly. For instance, Figure 3 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 sequence to be overridden, e.g., configurePhase 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

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
{ stdenv, fetchurl, pkgconfig, libXaw, libXt }:

stdenv.mkDerivation {
  name = "xmessage-1.0.2";
  src = fetchurl {
    url = mirror://xorg/individual/app/xmessage-1.0.2.tar.bz2;
    sha256 = "1hy3n227iyrm323hnrdld8knj9h82fz6s7x6bw899axcjdp03d02";
  };
  buildInputs = [ pkgconfig libXaw libXt ];
}
```

Fig. 3. xmessage.nix: Nix expression for xmessage.

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/). Therefore, it 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. (Indeed, the Nix Packages collection provides a function to build Cabal-based packages.)

All attributes in the call to mkDerivation are passed as environment variables to the build script (except the meta attribute [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 attribute buildInputs is bound to a list containing ghc and X11 as elements (in Nix expressions, list elements are separated by space) [3]. The paths to these dependencies are thus available to the build script via the buildInputs environment variable, which in turn is used by stdenv to generically set-up other environment variables such as the C include file search path and the linker search path. Similarly, the path to the xmessage package is substituted in the source file XMonad/Core.hs [5]. (Note that we can conveniently splice arbitrary Nix expresssions into strings using *string interpolations* of the form ${e}.) Finally, the function fetchurl returns a derivation that downloads the specified file and verifies its content against the given cryptographic hash (Schneier 1996); thus, the environment variable src will hold the location of the xmonad source distribution, which the stdenv build script will unpack. (The function fetchurl may seem impure, but because the output is practically 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 set to the path given in out [6].

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
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 = ...;
  ...
}
```

Fig. 4. all-packages.nix: Function calls to instantiate packages.

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: they will not overwrite each other, because Nix ensures that each instance is called with a different out environment variable, as we will see below.

Figure 4 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 2 8. Thus, xmonad is a concrete derivation that can be built. Similarly, the arguments are concrete derivations, e.g., xmessage is the result of a call to the function in Figure 3.

The user can now install xmonad by running the following command[1]:

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

This builds the xmonad derivation and all its dependencies, and ensures 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 nix-env operations include -e to uninstall a package, and --rollback to undo the previous nix-env action. There is also a command to build a derivation without installing it:

```
$ nix-build all-packages.nix -A xmonad
```

After a successful build, nix-build leaves a symlink ./result in the current directory that points to the output of the xmonad derivation.

[1] Command invocations from the Unix shell are denoted in this paper using the $ prefix.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

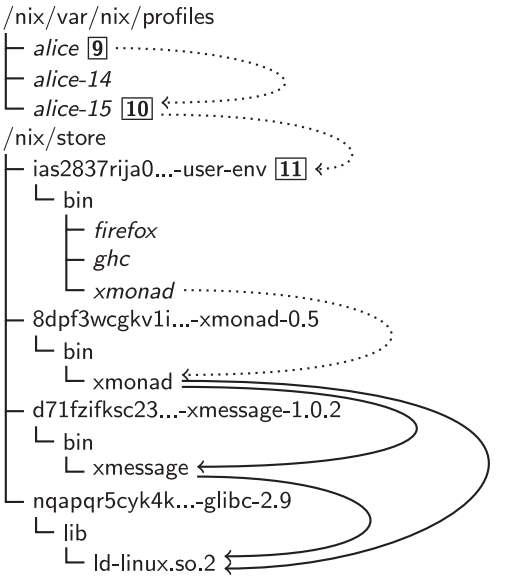

Fig. 5. The Nix store, containing xmonad and some of its dependencies, and profiles.

Nix is available at http://nixos.org/, along with the Nix Packages collection (Nixpkgs), a large collection of Nix expressions for nearly 2500 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., must not overwrite each other.

Nix accomplishes this by storing packages as immutable elements in 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 2 in Figure 2, 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.

Figure 5 shows 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. Similarly, xmonad contains the full path

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

to the xmessage program because we compiled it into the executable (at point [5] in Figure 2).

Figure 5 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. For instance, the user environment at [11] contains symlinks to firefox, ghc, and xmonad. A symlink /nix/var/nix/profiles/*user-number* [10] is created to point to that user environment, and finally the symlink /nix/var/nix/profiles/*user* [9] is atomically updated to point at *that* symlink. This makes the new set of packages appear in an atomic action in the user's PATH. Rollbacks are trivial: the command nix-env --rollback simply flips the symlink back to its previous target (e.g., to /nix/var/nix/profiles/alice-14).

Advantages. The purely functional approach has several advantages for package management (Dolstra 2006):

- 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. E.g., if the output of a build that had the path /nix/store/8dpf3wcg...-xmonad-0.5 as a build-time dependency contains the string 8dpf3wcg..., then we know that it has a potential runtime dependency on this instance of xmonad. 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-2.9, and the installation of some other package causes glibc-2.10 to be installed, it will not break xmonad: it will continue to link against glibc-2.9.
- Knowledge of the runtime dependency graph allows unused packages to be garbage-collected automatically. The roots of the collector are the profile symlinks in Figure 5.
- Since actions such as upgrading or downgrading to a different version are non-destructive, the previous versions are not modified. 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.
- 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 tell Nix that compressed archives of pre-build store paths are available for download from a remote

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

web server (such as the Nixpkgs distribution site). Then, when Nix needs to build some path p, it checks whether a pre-built archive of p is available remotely. If so, it is downloaded instead of building from source. 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. Not only can these relations be expressed easily 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 NixOS

In Section 3 we saw that the purely functional approach of the Nix package manager solves many of the 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. With NixOS, however, we have developed a Linux distribution built entirely on Nix.

4.1 Going all the way

NixOS 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, they can just as easily build any file or directory tree so long as it can be built and used 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 6 generates a simple configuration file (sshd_config) for the OpenSSH (Secure Shell) server program in the Nix store. That is, evaluating this expression will generate a file /nix/store/*hash*-sshd_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 returns different file fragments depending on configuration options in the attribute set config specified by the user (discussed below). Namely, if the option services.sshd.forwardX11 is enabled, the SSH option X11Forwarding should be set to yes, and furthermore SSH must be able to find the xauth program (to import X11 authentication credentials from the remote machine). This means that this configuration file *has an optional dependency on the* xauth

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
pkgs.writeText "sshd_config" ''
  UsePAM yes
  ${if config.services.sshd.forwardX11 then ''
    X11Forwarding yes
    XAuthLocation ${pkgs.xorg.xauth}/bin/xauth
  '' else ''
    X11Forwarding no
  ''}
''
```

Fig. 6. Nix expression to build sshd_config.

package: depending on a user configuration option, the configuration file either does or does not reference xauth. This kind of dependency between a configuration file and a software package is generally not handled by conventional package managers, since they do not 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 sshd_config refers to it – an important constraint on the integrity of the system.

When we build this expression, a possible result is

```
UsePAM yes
X11Forwarding yes
XAuthLocation /nix/store/j1gcgw...-xauth-1.0.2/bin/xauth
```

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, setting the option services.xserver.enable will cause a dependency on the X11 server (large), while services.xserver.desktopManager.kde4.enable will bring in the K Desktop Environment (very large).

4.2 Using NixOS

The rest of the system – the kernel, boot scripts, configuration files, and so on – are built in a similar manner. In Section 5, we will discuss in detail the implementation and organisation of the Nix expressions that constitute NixOS. First, however, we will give a high-level view of the system from the user's perspective, and along the way point out the advantages to end-users and system administrators that flow from the purely functional approach.

In an installed NixOS system, the Nix expressions that constitute NixOS and Nixpkgs reside in /etc/nixos/nixos and /etc/nixos/nixpkgs, respectively. There is a single top-level expression, /etc/nixos/nixos/default.nix containing an attribute system that, when evaluated, builds the entire NixOS system.

Reconfiguring. The user specifies the desired configuration of the system in a Nix expression, /etc/nixos/configuration.nix, that contains a nested attribute set

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
{ config, pkgs, ... }: 9

{
  boot.loader.grub.device = "/dev/sda"; 10
  boot.kernelModules = [ "fuse" "kvm-intel" ]; 11

  fileSystems = 12
    [ { mountPoint = "/";
        device = "/dev/disk/by-label/nixos";
      }
      { mountPoint = "/home";
        device = "/dev/disk/by-label/home";
      }
    ];

  swapDevices = [ { device = "/dev/disk/by-label/swap"; } ]; 13

  environment.systemPackages = [ pkgs.firefox ]; 14

  services.sshd.enable = true; 15
  services.sshd.forwardX11 = true; 16

  services.xserver.enable = true; 17
  services.xserver.videoDriver = "nvidia";
  services.xserver.desktopManager.kde4.enable = true;
}
```

Fig. 7. /etc/nixos/configuration.nix: NixOS configuration specification.

specifying all configuration pertaining to the user's system. This file is imported by the NixOS expressions and affects the resulting derivations. Figure 7 shows an example configuration file. It defines a simple, but typical workstation. It specifies that we want to run the SSH daemon to allow remote logins 15, and that it should support forwarding of X11 connections 16. As we saw in Figure 6, the value of attributes such as services.sshd.forwardX11 influences the generation of configuration files such as sshd_config. The configuration also specifies the filesystems to be mounted 12; the swap device 13; the disk on which the *GRUB boot loader* (see Section 5.2) is to be installed 10; that we need certain Linux kernel modules that are not loaded by default, such as the kvm-intel module to support hardware virtualisation 11; that Mozilla Firefox should be built and made available to users 14; and that we want a graphical environment, namely the KDE desktop environment 17.

(The reader may notice that configuration.nix is a *function* 9. This is mostly to conveniently pass the Nix Packages collection, allowing the user to refer to packages in the collection, such as pkgs.firefox. In Section 5 we shall see that configuration.nix is actually a *NixOS module*, and as such follows the calling convention for modules.)

The user can *realise* changes to configuration.nix by running this command:

```
$ nixos-rebuild switch
```

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

This 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*. As we shall see in Section 5, the activation script is a shell script built as part of the system derivation that takes care of the "imperative" aspects of actually switching to the new configuration, such as restarting system services. For instance, if the user changed the value of services.sshd.forwardX11, then the SSH daemon must be restarted to make it use the new sshd_config file.

A variant command, nixos-rebuild boot, also builds the new configuration, but does not activate it until the machine is next rebooted.

Upgrading. The same nixos-rebuild mechanism is also used to *upgrade* the system. NixOS 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/*, and then running nixos-rebuild.

A crucial advantage of NixOS is that reconfiguring or upgrading the system is *atomic* (or *transactional*), apart from the execution of the activation script. That is, during the build phase of nixos-rebuild, the currently active system configuration is in no way affected: none of the files of the current configuration in the Nix store are overwritten in place. The only moment when an inconsistency can be observed is during the activation phase; for instance, the system might temporarily run a PostgreSQL database server from the old configuration and an Apache web server from the new configuration, which can cause a problem. Even so, since the switch to the new configuration in the profile /nix/var/nix/profiles/system is atomic (as we saw in Section 3), after a system crash or reset, the system will start either all of the old configuration or all of the new configuration. That is, nixos-rebuild boot is entirely atomic, while nixos-rebuild switch has slightly weaker semantics.

Rollbacks. The profile /nix/var/nix/profiles/system (see Section 3) is used to enable rollbacks to previous configurations. One of the great strengths of NixOS is that upgrades are not destructive: the previous system configurations are still available in the Nix store. For example, to undo the result of a previous nixos-rebuild switch action, we simply do

```
$ nixos-rebuild --rollback switch
```

This command runs (nix-env --rollback) to flip the /nix/var/nix/profiles/system symlink from the new configuration back to the previous configuration, and then runs the activation script of the previous – now current – configuration.

The system profile is also 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 8). This enables another way to roll back to previous configurations: the user can recover trivially from bad upgrades (such as those that render the system unbootable) simply by selecting an older configuration in the boot menu.

Without user action, old configurations are kept indefinitely. If disk space becomes a problem, it is possible to get rid of the old configurations:

```
$ nix-env -p /nix/var/nix/profiles/system --delete-generations old
```

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

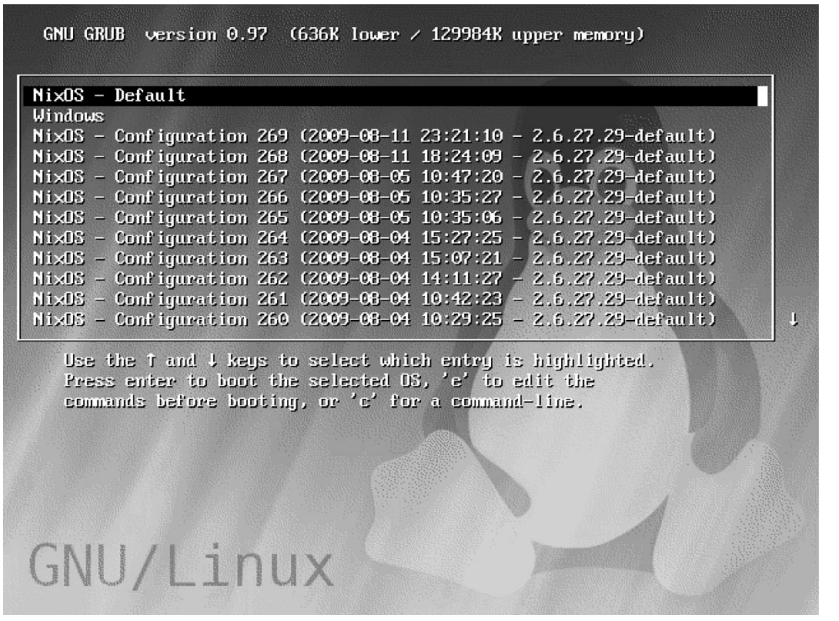

Fig. 8. The GRUB boot menu for a NixOS machine.

which deletes all symlinks to configurations other than the current one. The user can then run Nix's garbage collector (nix-store --gc). This deletes all paths in the Nix store not reachable (under the runtime dependency relation) from one of the garbage collector roots in /nix/var/nix/profiles.

Testing new configurations. The fact that configuration changes are non-destructive makes *testing* such changes a lot less threatening. The command nixos-rebuild test builds and activates a new configuration, but does not install it in the system profile or update the GRUB boot menu. Thus, rebooting or resetting the system will automatically cause a rollback to the current configuration in the system profile.

It is also possible to simply *build* the new configuration without activating it using the command nixos-rebuild build. This is often useful during development. Indeed, as Nix's security model allows any user to securely perform build actions in the Nix store, this command needs no root privileges.

A final, very attractive method to test configuration changes before committing to them in production use is the command nixos-rebuild build-vm. This command uses a different top-level derivation than the system attribute to build a *virtual machine* that runs the configuration specified in configuration.nix. It leaves a symlink result in the current directory pointing at a script that starts the virtual machine. Thus, the virtual machine can be started by running ./result/bin/run-nixos-vm. This pop up a window showing the display of the virtual machine. The configuration in the virtual machine is exactly the same as what we would get on the host machine if we were to run nixos-rebuild switch, except that the VM does not share the filesystems of the host, starting instead with an initially empty virtual disk. The fact

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

that we can so easily reproduce a system's configuration on either a virtual machine or a real machine is a direct consequence of the purely functional approach. As we shall see in Section 5.3, the generation of these virtual machines is highly time and space-efficient, making it very easy for users to test potentially dangerous upgrades.

Installation. To newly install NixOS on a machine involves the following steps. To prepare for the installation, the user boots the system from the NixOS installation CD, partitions the target disk, creates a root filesystem for NixOS and mounts it on /mnt. The user must then declaratively specify the desired configuration of the system in /mnt/etc/nixos/configuration.nix (using a text editor). The actual installation is initiated by running the command nixos-install, which creates a Nix store in /mnt/nix, copies a small bootstrap environment to that Nix store, performs a chroot operation (Stevens & Rago 2005) to /mnt, runs nixos-rebuild boot, and finally installs the GRUB bootloader to make the system bootable.

Obviously, tasks such as manually creating filesystems or writing a NixOS configuration expression are not for atechnical computer users. However, there is no reason why this installation process cannot be hidden behind a convenient (graphical) installer. The same applies to the reconfiguration of an installed system: the manipulation of configuration.nix and running of nixos-rebuild can be used as the back-end technology of a user-friendly configuration tool.

5 Implementation

In this section, we describe the implementation of NixOS in greater detail. In particular, we show how NixOS decomposes a Linux system into interrelated *modules* and *options* that together compute the top-level derivation that builds the entire operating system.

5.1 NixOS modules

We have seen in the previous sections that Nix can build software packages from purely functional specifications, and that we can apply the same approach to building the other static parts of a Linux system, such as configuration files and boot scripts. NixOS thus consists of a large set of Nix expressions that build the various parts constituting a Linux system: static configuration files, boot scripts, and so on. These build upon the software packages already provided by Nixpkgs. Even so, there are hundreds of derivations involved in building a usable Linux installation. To make building the system manageable and extensible, the Nix expressions are organised into *NixOS modules*. The essential feature of NixOS modules is that they allow *separation of concerns*: each module defines a single, "logical" part of the system (e.g., some system service, support for a certain kind of hardware device, or an aspect such as the system path), even though its *implementation* might cross-cut many "physical" parts (e.g., the derivations that build the boot scripts).

The goal of NixOS modules is to define the full *system configuration*: a nested attribute set named config whose members (called *options*) specify various aspects

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

of what the system should do. These include end-user configuration items such as the options services.sshd.enable or fileSystems in Figure 7, but also, crucially, values that are *computed* from the values of other options. This ultimately results in the option system.build.toplevel, whose value is a derivation that builds the entire system through its dependencies.

Example. Figure 9 shows the NixOS module that defines the OpenSSH daemon system service. A NixOS module defines a nested attribute set config [19] containing *option definitions*, that is, values of options. (The expression mkIf *b as* essentially evaluates to *as* if *b* is true, and the empty attribute set otherwise. We will return to this in Section 6.3.) These option definitions can be used by other modules. For instance, the sshd module *uses* (among others) the option services.sshd.enable [20], and *defines* the options users.extraUsers, jobs and networking.firewall.allowedTCPPorts, which are used by other modules. For instance, the firewall module uses the latter value in the definition of the derivation that produces the firewall set-up script. Thus, definitions across many modules contribute to the derivation that generates the firewall setup script[2].

Each option must be *declared* in some module before it can be *defined* by others. Modules declare options in their options attribute, along with possible default values, documentation and other meta-information. Thus, the sshd module at [18] declares the options services.sshd.enable and services.sshd.forwardX11.

The most important option defined by the sshd module is jobs, which specifies declaratively how the service is to be started. The option jobs is used in another module to generate concrete configuration files for *Upstart* (http://upstart.ubuntu.com/), the system "init" process responsible for starting and monitoring system services. For instance, the sshd module defines a job named sshd [22], with some shell code to be run before the job is started (preStart) and the actual command to start the service (exec).

To support the ability to upgrade and roll back cleanly, NixOS services must be self-initialising and idempotent, as the sshd service illustrates. At start up, it creates the environment needed by the sshd daemon, such as the server host key /etc/ssh/ssh_host_dsa_key. This removes the need for post-install scripts used in other package managers (Cosmo *et al.* 2008) and allows switching between configurations.

Computing the system configuration. The general structure of a NixOS module is as follows:

```
{ config, pkgs, ... }:
{ options = { nested attribute set of option declarations using mkOption };
  config = { nested attribute set of option definitions };
}
```

[2] Prior to the development of NixOS's module system, the Nix expression defining the firewall derivation had to use the option services.sshd.enable to decide whether to include port 22. This hurt extensibility and violated the principle of separation of concerns.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
{ config, pkgs, ... }: with pkgs.lib;

{

  options [18] = {
    services.sshd.enable = mkOption {
      default = false;
      description = "Whether to enable the Secure Shell daemon.";
    };
    services.sshd.forwardX11 = mkOption {
      default = true;
      description = "Whether to allow X11 connections to be forwarded.";
    };
  };

  config [19] = mkIf config.services.sshd.enable [20] {

    users.extraUsers = singleton
      { name = "sshd"; [21]
        uid = 2;
        description = "SSH privilege separation user";
        home = "/var/empty";
      };

    jobs.sshd = [22]
      { description = "OpenSSH server";

        startOn = "started network-interfaces";

        preStart =
          ''
            mkdir -m 0755 -p /etc/ssh
            if ! test -f /etc/ssh/ssh_host_dsa_key; then
                ${pkgs.openssh}/bin/ssh-keygen ...
            fi
          '';

        daemonType = "fork";

        exec =
          let sshdConfig = pkgs.writeText "sshd_config" (elided; see Figure 6);
          in "${pkgs.openssh}/sbin/sshd -D ... -f ${sshdConfig} [23]";
      };

    networking.firewall.allowedTCPPorts = [ 22 ];
  };
}
```

Fig. 9. sshd.nix: NixOS module for the OpenSSH server daemon.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

That is, a module is a function that *takes* the full system configuration in the argument config and, for convenience, the Nix Packages collection in pkgs; and *returns* a nested set of option declarations in options and a nested set of option definitions in config. (The configuration in Figure 7 uses an abbreviated module style for modules that define, but do not declare options.)

The config attribute sets returned by each module collectively define the full system configuration, which is computed by *merging* them together. Multiple definitions of an option are combined using that option's *merge function*. Each option declaration can define a specific merge function, which has a default based on the type of the option. For instance, lists are by default concatenated: if in addition to the sshd module, some other NixOS module has a configuration value

```
networking.firewall.allowedTCPPorts = [ 80 ];
```

then the value of this option in the final system configuration will be [22 80] or [80 22], depending on the module order.

The full configuration resulting from the merge is passed as an *input* back into each module through the config function argument[3]. This allows modules to refer to values defined in other modules. Note that, while the computation of the full system configuration is circular, thanks to laziness there is no problem as long as no individual options have a circular dependency on each other.

5.2 Composing a Linux system

As we saw, modules such as sshd.nix map high-level options such as services.sshd.enable to lower-level implementation options such as jobs. However, these are still far removed from actual Nix derivations that allow us to build a system. (For instance, the sshd module itself specifies no derivations other than the call to pkgs.writeText to generate the sshd_config file in the Nix store.) It is up to other modules to take options such as jobs and map them onto still lower-level options, until we end up with the option system.build.toplevel whose value builds the entire system.

To see how we get from modules such as sshd to a running system, it is necessary to understand the major parts of a Linux system and their interdependencies. Figure 10 shows the dependency graph of a small subset of the derivations that build NixOS. The top-level derivation is the option system.build.toplevel, the evaluation of which causes the entire system to be built – hundreds of derivations that collectively build a NixOS operating system instance. Each arrow $a \rightarrow b$ means that the output of derivation a is an input to derivation b. Italic nodes denote derivations that build configuration files; bold nodes build Upstart jobs; 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, the distinction between different node

[3] Note that there are two identifiers named config: one, the function argument config, is the final system configuration, while the other, the result attribute config, is this module's contribution to the final configuration. Because the resulting attribute set is not recursive (using rec), references to config such as config.services.sshd.enable at 20 in Figure 9 refer to the final system configuration.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

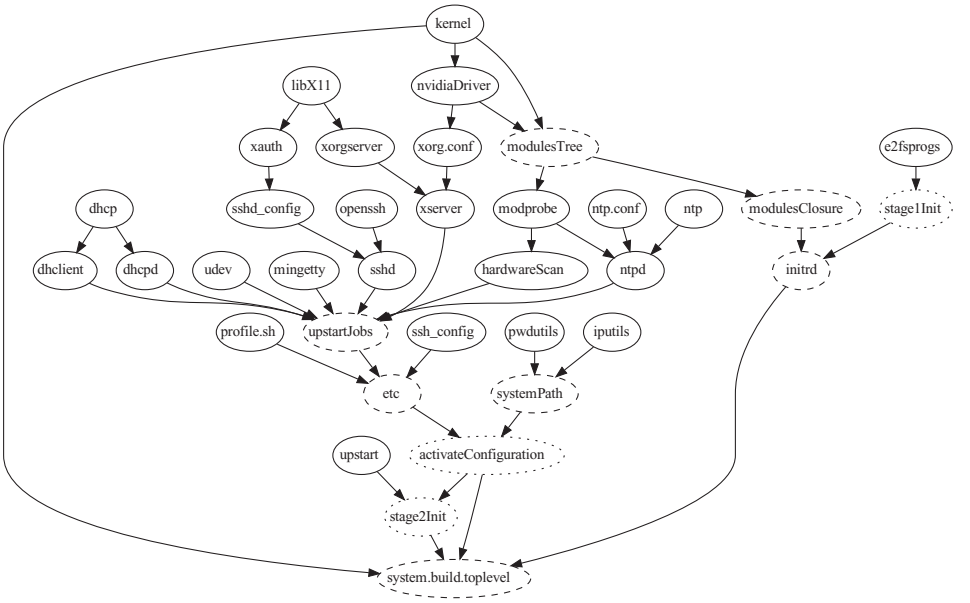

Fig. 10. A small part of the dependency graph of the derivations that build NixOS

types is entirely conceptual. As far as Nix is concerned, all nodes are derivations: pure, atomic build actions.

To illustrate the flow of values from high-level modules such as sshd to the final system.build.toplevel option, we will discuss the most important parts of the system in more detail.

Upstart jobs. A running system consists primarily of a set of *services*, such as the X server, the DHCP client, the SSH daemon, the Apache webserver, and many more. On NixOS, these are started and monitored by Upstart, which provides the init process that starts all other processes. Upstart reads specifications of system "jobs" from /etc/init and runs the commands for each job when appropriate. Jobs can have dependencies on each other, e.g., the SSH daemon should be started after the job that initialises networking has succeeded.

As we saw, NixOS modules define system jobs through the jobs option (e.g. 22 in Figure 9). The module upstart.nix maps these to derivations that build Upstart job files, such as sshd or xserver in the graph. Note that the jobs option abstracts over the concrete Upstart syntax. This allows us to switch to a different "init" system, or to build test scripts for system services to allow them to be run outside of Upstart.

Figure 11 shows the output of the generated Upstart job for sshd. The path to the sshd configuration file (built by the derivation in Figure 6) is substituted in the Upstart job at 24 as a result of the string interpolation at 23. This means that the Upstart job has a runtime dependency on the sshd configuration file, discovered by

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
# Upstart job 'sshd'.  This is a generated file.  Do not edit.
description "OpenSSH server"

start on started network-interfaces

start script
  mkdir -m 0755 -p /etc/ssh
  if ! test -f /etc/ssh/ssh_host_dsa_key; then
      /nix/store/giiav2gdivfi...-openssh-5.2p1/bin/ssh-keygen ...
  fi
end script

exec /nix/store/giiav2gdivfi...-openssh-5.2p1/sbin/sshd -D ... \
  -f /nix/store/cs9fh614q8g0...-sshd_config 24

expect fork
respawn
```

Fig. 11. /nix/store/qjyn954j5jab...-upstart-sshd/etc/init/sshd.conf: Generated Upstart job for the OpenSSH server daemon

scanning for hashes. As a consequence, the latter will not be garbage collected as long as the former is not garbage.

Furthermore, it means that the sshd service does not need a "global" /etc/ssh/sshd_config, in contrast to conventional Linux distributions. While this is not a big deal for the sshd service, it is very important for other services: for instance, it makes it easy 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/ssh/sshd_config would preclude this.

Cross-cutting configuration files. By contrast, 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 is not 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 by the DHCP client. However, others are static (i.e., only change as a result of explicit reconfigurations) but still cross-cutting, such as the static host name resolution list in /etc/hosts, the list of well-known TCP and UDP ports in /etc/services, or the configuration files of the PAM authentication system in /etc/pam.d/. These are built by Nix expressions, stored in the Nix store and symlinked from /etc in the activation script (described below).

Modules can define static files that should appear in /etc through the option environment.etc. This is a list of attribute sets specifying files in the Nix store along with the intended locations in /etc. The etc derivation in Figure 10 combines the files in environment.etc into a tree of symlinks. Later, theactivation script copies

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

these symlinks into /etc. For instance, the option definition

```
environment.etc =
  [ { source = pkgs.writeText "hosts" "127.0.0.1 localhost";
      target = "hosts";
    }
  ];
```

causes the activation script to create a symlink /etc/hosts to a file in the Nix store (say, /nix/store/04vi02p4k8sp...-hosts) containing the line 127.0.0.1 localhost.

Note that the use of "global" files such as those in /etc dilutes the purely functional approach, as the values of the symlinks in /etc are stateful. We discuss the extent of this problem in Section 6.2.

System path. Modules can define packages that should appear in the search path of all users through the option environment.systemPackages. For instance, the definition environment.systemPackages = [pkgs.firefox] in Figure 7 makes the programs in the Firefox package available to all users. The systemPath derivation in Figure 10 creates a tree of symlinks to all files in the store paths in environment.systemPackages. This enables declarative package management. By contrast, manual package management actions such as nix-env -i (Section 3) are stateful and do not use a declarative specification to define the set of installed packages.

Activation script. 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 or state directories such as /var/log must be created, and so on. This is done by the *activation script*, built by the derivation activateConfiguration. As we saw in Section 4.2, the activation script is run by the command nixos-rebuild switch after building the system configuration in the Nix store. Thus, it can perform stateful initialisation that cannot be done as part of purely functional build actions. Modules can specify script fragments to be included in the activation script by defining the option system.activationScripts.

As an example, consider the creation of user accounts required by system services. For instance, the SSH daemon requires the existence of an account named sshd for its privilege separation feature. Modules can declaratively specify user accounts through the option users.extraUsers. Thus, the sshd.nix module in Figure 9 defines at point 21 that an account named sshd must exist with user ID 2 and home directory /var/empty. This option is implemented in another module (users-groups.nix) by defining a script fragment in the option system.activationScripts that makes the necessary calls to shell commands such as useradd to idempotently create or update the required accounts.

Kernel. The Linux kernel consists of the kernel proper (the node kernel in Figure 10) along with external kernel modules – drivers and other kernel extensions in the Linux model – such as the NVIDIA graphics drivers (nvidiaDrivers). These are

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

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 upgrade problem is that the user upgrades the kernel and then discovers that the X11 windowing system will not start anymore because the NVIDIA drivers were not rebuilt. This is not possible here, because system upgrades are done exclusively by rebuilding the system.build.toplevel derivation; therefore, *all* its dependencies must build successfully. (Of course the external modules might not be compatible with the new kernel, but such a problem usually manifests itself at build time.) This illustrates that *the declarative model of NixOS ensures that the running system is consistent with the specification of the intended configuration.*

The small wrapper component modulesTree is an example of how we circumvent the impure models of various tools and applications. The command to load Linux kernel modules, modprobe, expects all modules to live in a single directory tree (/lib/modules/*kernel-version* on conventional Linux distributions). Thus, modules from various packages are expected to be installed in an existing directory, which is therefore stateful. The modulesTree wrapper allows the modprobe command to work in a purely functional discipline: it combines (through symlinks) the kernel modules from the kernel package with those from the external module packages in a single directory tree, which is then used by modprobe.

Boot scripts and initial ramdisk. NixOS, as most 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*, and starts the kernel. The initial ramdisk is a compressed filesystem image whose sole purpose is to mount the root filesystem of the running system. After loading the initial ramdisk, the kernel starts the *stage 1 init script*: the file /init in the initial ramdisk. It loads any drivers needed to mount the root filesystem, runs the filesystem checker fsck if needed, mounts the root filesystem, performs a chroot operation to make it the actual root of the filesystem hierarchy, and passes control to the *stage 2 init script*. This script then performs remaining boot-time initialisation (such as clearing the contents of the runtime state directory /var/run), runs the activation script, and finally starts Upstart to boot and manage all system services. These services include initialisation tasks such as bringing up the network, activating swap devices, and mounting remaining filesystems.

The initial ramdisk is built by the derivation initrd. Because it must include all kernel modules necessary to mount the root filesystem, it is not fixed, but depends on the hardware configuration of the user. For instance, it must include kernel modules for the hardware containing the filesystem (e.g. drivers for SCSI or USB) and the filesystem itself (e.g., the ext3 filesystem).

Top-level derivation Finally, the value of the option system.build.toplevel is a derivation that simply creates symlinks to its inputs, e.g., $out/kernel links to

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

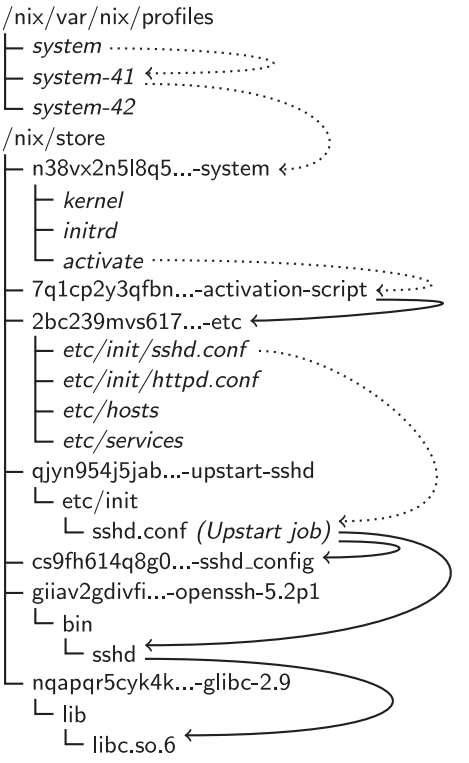

Fig. 12. Outputs of derivations related to the OpenSSH server daemon, as well as the system profile

the kernel image, $out/activate links to the activation script, and so on. These are sufficient to allow the command nixos-rebuild switch to activate the newly built configuration (by running the activation script) and to update the GRUB boot menu (using the symlinks to the kernel and initial ramdisk).

Figure 12 shows a small part of the file hierarchy structure in the Nix store that results from building a NixOS system configuration with services.sshd.enable and services.sshd.forwardX11 both enabled. As in Figure 5, italicised names and dotted arrows denote symbolic links, while solid arrows denote other kinds of dependencies. For example, the output of the top-level derivation (/nix/store/n38vx2n5l8q5...-system) has a symlink to the activation script, which in turn contains the path of the output of the etc derivation. Recall that the etc derivation contains symlinks to the files that must be placed in /etc at activation time. One of those is /etc/init/sshd.conf, which is the Upstart job for the sshd service. As seen in Figure 11, this file contains build-time generated references to the openssh package and the sshd_config configuration file. Finally, the openssh package has a dependency on the GNU C library (the glibc package). The Nix package manager keeps track of all these dependencies. This means that none of the paths in Figure 12 will be garbage collected as long as they are reachable from any of the symlinks in /nix/var/nix/profiles.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

The Nix expression /etc/nixos/nixos/default.nix contains a call to the function that imports and evaluates all NixOS modules and merges the resulting config attribute sets together to form the full system configuration. Thus, the following sequence of commands:

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

builds the entire configuration (including all software dependencies, configuration files and Upstart jobs), leaving a symlink result in the current directory; and then runs the activation script.[4] When switching to the new configuration, the activation script stops any Upstart jobs that have disappeared in the new configuration, starts new jobs, restarts jobs that have changed, and leaves all other jobs untouched. It can determine precisely which jobs have changed by comparing the store paths of the Upstart job files: if they are different, then by definition some input in the dependency graph of the job changed, and a restart may be needed.

5.3 Building disk images and virtual machines

Normally, building a NixOS configuration entails evaluating the option system.build.toplevel, which allows the configuration to be installed in the system profile in /nix/var/nix/profiles, activated and booted from the GRUB boot menu. However, there are specialised modules that define other "top-level" options to build a system in rather different ways. Notably, these are the generation of CD-ROM images containing NixOS, and the generation of NixOS virtual machines.

ISO images. For instance, the module installer/cd-dvd/iso-image.nix defines a derivation option system.build.isoImage that runs the genisoimage command to generate an ISO-9660 (CD-ROM) filesystem image containing the closure of the output of the derivation system.build.toplevel (i.e., the usual top-level derivation), along with files such as the GRUB boot loader to make the CD bootable. Thus, slightly simplified, one can run the command

```
$ nix-build /etc/nixos/nixos -A system.build.isoImage
```

to generate a NixOS CD-ROM image that can be burned onto a CD. This is a so-called "live CD": the NixOS system on the CD behaves like a normal installation except that the filesystem is not persistent across reboots. A ramdisk stacked on top of the CD filesystem using a union filesystem (Pendry & McKusick 1995) is used to make the CD appear writable.

This module is used to generate the NixOS installation CDs in a modular way. For instance, there is a module installation-cd-graphical.nix that builds a large, full-featured CD containing X11 and the KDE desktop environment. It does this by importing iso-image.nix, and then defining options such as

[4] This sequence of commands is essentially what the user command nixos-rebuild test does; nixos-rebuild switch in addition regenerates the GRUB boot menu.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

```
{ services.xserver.enable = true;
  services.xserver.desktopManager.kde4.enable = true;
}
```

By contrast, the module installation-cd-minimal.nix specifies a small CD image without a GUI and related features.

Cheap virtual machines. The module virtualisation/qemu-vm.nix defines a derivation option system.build.vm that produces a shell script that starts a virtual machine running the given NixOS system configuration. This is used by the command nixos-rebuild build-vm (Section 4.2) to allow users to safely test configurations in a VM. The VM is implemented using QEMU/KVM (http://www.linux-kvm.org/), which has the ability to start a Linux kernel directly from the host filesystem (i.e., without the need for a virtual disk image containing that kernel). Essentially, the content of the start script is

```
${pkgs.qemu_kvm}/bin/qemu-system-x86_64 -smb / \
  -kernel ${config.boot.kernelPackages.kernel} \
  -initrd ${config.system.build.initialRamdisk} \
  -append "init=${config.system.build.bootStage2} ..."
```

The module also defines the fileSystems option as

```
[ { mountPoint = "/hostfs"; device = "//10.0.2.4/qemu";
    fsType = "cifs"; }
  { mountPoint = "/nix/store"; device = "/hostfs/nix/store";
    options = "bind"; }
]
```

to make the initial ramdisk mount the Nix store on the host machine through the CIFS network filesystem. This is a feature provided by QEMU/KVM, which provides a virtual Ethernet card to the guest. The option -smb / causes QEMU/KVM to automatically start a CIFS server attached to IP address 10.0.2.4 in the virtual network.

The fact that the virtual machine shares the Nix store of the host makes the generation of VMs very efficient: we do not have to generate a disk image containing the closure of system.build.toplevel, which would typically be hundreds of megabytes large. As a result, nixos-rebuild build-vm can build VMs for test purposes very quickly.

Virtual machine images. While the QEMU-based approach does not build virtual machine images, it is not hard to do so with the appropriate tools. For instance, NixOS provides a module that builds a bootable disk image for Amazon's Elastic Compute Cloud (http://aws.amazon.com/ec2/). It defines a derivation option system.build.amazonImage that creates and formats a disk image and fills it with the closure of the store path created by system.build.toplevel. It also defines the network configuration necessary to make the machine reachable from outside the cloud. Thus, we can take an arbitrary NixOS system configuration module (e.g., the one in Figure 7) and combine it with the Amazon EC2 module to generate a cloud machine that implements the specified configuration.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

6 Evaluation and discussion

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, how pure Nix derivations actually are, and what compromises to purity were necessary. We also discuss the security implications of purity, the importance of laziness in the Nix expression language, and the type system of the language.

6.1 Status

NixOS is not a proof-of-concept: it is in production use on a number of desktop and server machines. Sources and ISO images of NixOS for i686 and x86_64 platforms are available at http://nixos.org/. As of May 2010, the Nix Packages collection (on which NixOS builds) contains Nix expressions for around 2481 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 KDE desktop environment 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).

To provide some sense of the size of a typical configuration (a laptop running X11, KDE and Apache, among other things[5]): the build graph rooted at the top-level system attribute in /etc/nixos/nixos/default.nix consists of 1036 derivations (or 582 excluding fetchurl derivations) and 167 miscellaneous source files. The closure of the output of the system derivation (i.e., its runtime dependencies) consists of 487 store paths with a total size of 1991 MiB, on a filesystem with a 2 KiB block size. (To place this roughly in context, a 32-bit install of Windows 7 requires 16 GiB of disk space.) Thus more than half of the derivations are build-time-only dependencies, such as source distributions, compilers, and parser generators. The evaluation of system imports 560 Nix expression files in the NixOS and Nixpkgs trees, and takes 1.5 seconds of CPU time on an Intel Core 2 Duo 7700 running 32-bit NixOS.

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" such as /bin that prevent multiple versions of packages and services from existing 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

[5] Revision 21775 of the configuration at https://svn.nixos.org/repos/nix/configurations/trunk/misc/eelco-dutibo.nix.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

an instance of the Bash shell in the Nix store. This symlink is created by the activation script and is needed because 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 the 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 do not 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 required libraries.

Configuration data. NixOS has far 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 sshd -f ${sshdConfig} in Figure 9). What remains are cross-cutting configuration files, which, as discussed in Section 4, are built purely but then symlinked from /etc by the configuration's activation script. The configuration mentioned in Section 6.1 has 88 such symlinks (including 37 Upstart jobs in /etc/init and 18 symlinks for services of the PAM authentication framework).

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

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

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 bound statically: the implementation of the OpenGL graphics library to be used at runtime (libGL.so), which is hardware specific. We build applications that need OpenGL against Mesa (an OpenGL implementation that provides a software renderer), but add the path to the actual OpenGL implementation selected in the user's system configuration to the LD_LIBRARY_PATH environment variable.

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 this 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 seven 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. (As with conservative garbage collectors, there is a possibility that dependencies are not found if they are represented in a non-standard way, such as in a UTF-16 encoding or as a result of compression. However, we have not encountered this in practice.) 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

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

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 if /var/run/dbus did not exist. Nix can optionally perform builds in a chroot-environment (Stevens & Rago 2005), where directories such as /var do not exist, but this is somewhat less portable.

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[6] 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.

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.

Cost of purity. The purely functional model of NixOS comes at a substantial cost in terms of disk space usage. Since derivations never overwrite each other, we can end up with many versions or variants of the same package in the Nix store. In particular, a change to a package, no matter how trivial, causes a rebuild or redownload of all dependent packages. The worst case is a change to a fundamental package such as Glibc, which triggers a rebuild or redownload of the entire system. On the other hand, upgrades to derivations near the "top" of the dependency graph tend to be cheap. This is exactly analogous to purely functional data structures in a language such as Haskell (since the store *is* a purely functional data structure): updating the first element of a list is cheap, while updating the last element is expensive.

[6] To be precise, the i686-linux derivations from build-for-release.nix in revision 11312 of https://svn.nixos.org/repos/nix/nixpkgs/branches/purity-test.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

As a result, even if we garbage collect after every upgrade, NixOS may require twice the amount of disk space of a conventional system to perform an upgrade; e.g., almost 4 GiB for the configuration in Section 6.1, rather than 2 GiB. It is worth pointing out, however, that other systems have rollback facilities that are similarly expensive; for instance, Windows *restore points* consume gigabytes of data in typical configurations.

Nix has a tool (nix-store --optimise) to optimise disk space usage by searching the store for identical files and replacing them with hard links to a single copy. For instance, given a rebuild of the configuration in Section 6.1 after changing a single character in a derivation at the root of the dependency graph, this tool found that out of the 89150 files in the new configuration, 77702 were identical to a file in the old configuration, leading to a savings of 1139 MiB, or 57.3% of the total (on a filesystem with a 2 KiB block size).

Note that the proliferation of multiple versions of packages only happens *between* builds of the system configuration; each such build itself generally contains only one version of each dependency. Thus, after garbage collecting all non-current configurations, we end up with only one version of each package. (The only exception is if the configuration explicitly specifies multiple versions or variants of a package.) On the other hand, user environments created with nix-env typically contain many different versions of dependencies, because these tend to accumulate over time – there is, after all, no reason to update a package in a profile if it works properly. For instance, the main profile on the first author's laptop after around 2.5 years and 237 nix-env actions had a closure containing 714 store paths and 275 unique packages. Thus 429 paths were variants of other paths; e.g., nine versions of libX11, eight versions of GTK+, and six versions of Glibc. This proliferation can be prevented by using the "declarative" style of package management through the option environment.systemPackages (Section 5.2).

6.3 Laziness

The use of lazy evaluation for the Nix expression language was a conscious design decision. Here, we discuss how the use of laziness leads to advantages both in the specification of individual derivations and in building the NixOS system configuration.

Laziness in derivations. Evaluation of a derivation means performing a build action, and build actions can be quite expensive: some packages require a significant amount of data 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.

The choice for lazy evaluation allows us to write Nix expressions in a convenient and elegant 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 example, a derivation for a package can check a Boolean parameter to determine

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

whether to actually pass a certain library to the build inputs or not. An example of such a design pattern is shown in Figure 6.

At the call site of a function, we can supply all potential dependencies without having to worry that unneeded dependencies might be evaluated. For instance, the whole of the Nix packages collection is essentially one attribute set, where each attribute maps to one package contained in the collection (cf. Figure 4). It is very convenient that at this point, we do not have to deal with the administration of what exactly will be needed where.

Another benefit is that we can easily store meta information with packages, such as name, version number, homepage, license, description, and maintainer. Without any extra effort, we can access such meta-information without having to build the whole package.

Laziness in computing the system configuration. Recall from Section 5.1 that the full system configuration config, computed by combining the config attribute from each NixOS module, is passed as an *input* to each module in the config function argument. That is, *the input of a module depends on its own output* – a circular definition. This feature allows modules to reflect on the full system configuration, using option definitions from other modules regardless of where they are defined. Thanks to the laziness of the Nix expression language, this kind of circularity is fine as long as there is no circularity between the definitions of individual options.

However, circularity does lead to a problem: in some common cases, we are not lazy enough. Recall that the sshd module (Figure 9) is optionally enabled or disabled through the option services.sshd.enable. Naively, we might write the module as follows:

```
{ config, pkgs, ... } :
{ config =
    if config.services.sshd.enable
    then { users.extraUsers = ...; ... }
    else { };
}
```

Unfortunately, this leads to an infinite recursion (detected by the Nix expression evaluator through *blackholing* Peyton Jones 1992; Dolstra 2008). To evaluate the function argument config, we need to recursively merge each module's config attribute, which therefore need to be evaluated as well; but to evaluate the config attribute of the module above, we need to evaluate the condition of the if, which requires us to evaluate the function argument config, which is circular.

The solution is the function mkIf (used in Figure 9 at [20]), which *delays* the evaluation of the conditional by "pushing it down" to the individual option definitions. Essentially, the mkIf transforms the attribute set into this

```
{ users.extraUsers =
    if config.services.sshd.enable then ... else ignore;
  ...
}
```

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

where the ellipses denote the original values of the option definitions, and ignore is a special value that is filtered out when the definitions are merged.

6.4 Types

The Nix expression language is dynamically typed. In contrast to the choice of evaluation strategy, the decision for dynamic typing was less conscious, and mainly motivated by keeping the language design simple.

The use of dynamic types has influenced the way Nix expressions are currently being written. By now, there are quite a few library functions that depend on run-time type analysis and meta-programming techniques, both of which are facilitated by the lack of a static type system (and will also make adding such a system at a later stage harder): the Nix expression language has built-in functions for run-time type testing, and offers functions that can check for the presence of a particular attribute in an attribute set, or even turn an attribute set into a list of string-value pairs. One significant example, where meta-programming is used is the NixOS module system: as explained in Section 5.1, all modules are composed in an intricate way that makes use of reflection.

A benefit of dynamic types is that it becomes somewhat easier to support migration between library interfaces. If a library function changes behaviour, but a user-developed Nix function still makes use of an outdated interface, we can dynamically check for it, issue a deprecation warning, and then convert it automatically to the new interface, all without special built-in language support.

However, we are also 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 build time. Furthermore, we believe that a suitable type system can be an asset to end users, as (graphical) user interfaces for system configuration management could be derived from the types.

6.5 Security issues

Some Unix packages need to install programs with *setuid* or *setgid* permissions. These cause a program to be executed under the identity of the owner of the program rather than under the identity of the caller (Stevens & Rago 2005). For instance, the passwd program must be installed *setuid* root to allow it to modify the system password file. However, such binaries cannot be allowed in the Nix store due to the security implications of the purely functional model: if a security bug is discovered in a *setuid* binary such as passwd (e.g., a privilege escalation bug), it *must* be overwritten or deleted, since otherwise the security hole remains in the system. This is in contrast to security bugs in normal programs, which we merely need to refrain from using; their presence in the store is not a problem. However,

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

the purely functional model disallows overwriting of files, and only deletes files if they can be safely garbage collected. Fortunately, because Nix executes derivations under a non-privileged user account, they *cannot* install *setuid* binaries.

Since programs in the Nix store cannot have these permissions, we push the issue of supporting *setuid* programs to activation time, where we are allowed to have statefulness: we let the activation script create trivial wrapper programs with the desired permissions in /var/setuid-wrappers that call the wrapped program. For instance, /var/setuid-wrappers/passwd is a *setuid* root wrapper that calls the real passwd program, e.g., /nix/store/sbci75s3c7im...-pwdutils-3.1.3/bin/passwd. Since only the root user has sufficient permission to run the activation script succesfully, unprivileged users cannot use this mechanism to create wrappers: they can build a NixOS configuration (e.g., using nixos-rebuild build), but cannot activate it.

A NixOS option defines the programs for which wrappers must be created. This has the security advantage of making explicit the set of potentially dangerous, privileged programs. The configuration described in Section 6.1 creates 16 *setuid* wrappers.

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. Tucker and Krishnamurthi (2001) proposed modelling packages and system configurations using the *unit* system of MzScheme, allowing functional abstraction, explicit composition and multiple instantiations of packages and system services. 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.

The system configuration management literature sometimes distinguishes between configuration management tools with *convergent* and with *congruent* behaviour (Traugott & Brown 2002). In a convergent model, the tool tends to make the system configuration *converge* towards a desired state as a result of iterative changes to a system specification. In a congruent model, the tool guarantees that the actual configuration of the system matches the specification of the desired configuration. Disregarding mutable state, NixOS has a congruent model: after a nixos-rebuild, the system is in a state determined by the NixOS system configuration specification, and independent from the previous state of the system.

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

An example of the convergent approach is Cfengine (Burgess 1995), 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; if it doesn't, add it"). Cfengine is convergent in that the desired state is implicit: when faced with an actual system configuration that differs from the desired configuration, the system administrator updates the Cfengine configuration with actions that attempt to *modify* the state of the system towards the desired state. This is an iterative process. However, these actions transform a possibly unknown state of the system, and can therefore have all the problems of statefulness. Furthermore, since actions are specified with respect to fixed configuration file locations (e.g., /etc/sendmail.mc), it is not straightforward to enable multiple configurations to coexist in the same system.

ISconf (Traugott & Brown 2002) is an example of a tool that aims at having a congruent model. It attempts to achieve congruence by keeping a list of all actions that have been performed on a system. This allows the configuration to be reproduced on an empty machine by replaying the history of actions. However, maintaining this history may be impractical; e.g., we need to keep around all old versions of actions because they may have contributed to the current state of the system in the past (Kanies 2003). NixOS does not have this problem because it does not rely on the previous state.

The Nix expression language is essentially a functional "Make" (Feldman 1979). Indeed, Make is used in the FreeBSD Ports collection (FreeBSD Project 2009) as a high-level driver for package management. However, without functions, it lacks a clean mechanism to deal with variability in packages; and because Make actions destructively update files, it has all of the problems of statefulness discussed in Section 2. The FreeBSD Ports collection does not use Make to handle non-package parts of the system, such as configuration files in /etc. On the other hand, Vesta (Heydon *et al.* 2001), an integrated configuration management system that provides revision control as well as build management, has a purely functional build language – the System Description Language (Heydon *et al.* 2000). Thus it is quite similar to Nix, and could be used to support purely functional deployment (but to our knowledge, has not been applied to this domain). Similarly, Odin (Clemm 1986, 1995) has a functional language to describe build actions (*queries*). Indeed, a reimplementation of Odin's semantics as an embedded domain-specific language in Haskell is presented by Sloane (2002).

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. NixOS also forms a compelling demonstration of the applicability of lazy, purely functional programming in an unconventional application domain.

Future work. 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

https://doi.org/10.1017/S0956796810000195 Published online by Cambridge University Press

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.

Another issue concerns management of mutable state, which is currently not supported except in an *ad hoc* manner through activation scripts. For instance, if an upgrade requires a database schema to be updated, then a subsequent rollback requires the schema change to be undone. This is clearly a difficult problem. We may also wish to be able to obtain the closure of not only the static parts of a running system, but also its mutable parts, e.g., in order to migrate it to another machine. A partial solution may be to keep state in a mutable variant of the Nix store, say, `/nix/state` (den Breejen 2008).

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 should be invisible and scheduling must be deterministic.

Acknowledgments

This research was supported in part by NWO-JACQUARD project 638.001.201, *TraCE: Transparent Configuration Environments*, and by Philips Healthcare and NWO-JACQUARD project 638.001.208, *PDS: Pull Deployment of Services*. We wish to thank Armijn Hemel, who implemented the first prototype of NixOS. We are grateful to all NixOS contributors: Lluís Batlle, Martin Bravenboer, Wouter den Breejen, Sander van der Burg, Ludovic Courtès, Tobias Hammerschmidt, Yury G. Kudryashov, Marco Maggesi, Michael Raskin, Rob Vermaas, and Marc Weber. Eelco Visser and Merijn de Jonge contributed to the development of Nix. We also wish to thank the anonymous ICFP and JFP reviewers for their comments.

References

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

Beshers, Clifford, Fox, David, & Shaw, Jeremy. (2007). Experience report: using functional programming to manage a Linux distribution. In *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, R. Hinze and N. Ramsey (eds.), New York, NY, USA: ACM, pp. 213–218.

Boehm, Hans-Juergen. 1993 (June). Space efficient conservative garbage collection. *Pages 197–206 of: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.* SIGPLAN Notices, no. 28/6.

den Breejen, Wouter. 2008 (Mar.). *Managing state in a purely functional deployment model*. M.Phil. thesis, Dept. of Information and Computing Sciences, Utrecht University. INF/SCR-2007-053.

Burgess, Mark. (1995). A site configuration engine. *Computing systems*, **8**(3), 309–337.

Clemm, G. (February 1986) *The Odin System — an Object Manager for Extensible Software Environments*. Ph.D. thesis, University of Colorado at Boulder.

Clemm, G. (1995) The Odin system. In *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*. Lecture Notes in Computer Science, no. 1005. Springer-Verlag, pp. 241–262.

Cosmo, R. D., Zacchiroli, S. & Trezentos, P. (2008) Package upgrades in FOSS distributions: details and challenges. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*. New York, NY, USA: ACM, pp. 1–5.

den Breejen, W. (March 2008). *Managing State in a Purely Functional Deployment Model*. M.Phil. thesis, Dept. of Information and Computing Sciences, Utrecht University. INF/SCR-2007-053.

DeTreville, J. (2005) Making system configuration more declarative. In *HotOS X: 10th Workshop on Hot Topics in Operating Systems*. USENIX, pp. 61–66.

Dolstra, E. (November 2005). Secure sharing between untrusted users in a transparent source/binary deployment model. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pp. 154–163.

Dolstra, E. (2006) *The Purely Functional Software Deployment Model*. Ph.D. thesis, Faculty of Science, Utrecht University, The Netherlands.

Dolstra, E. (2008) Maximal laziness — an efficient interpretation technique for purely functional DSLs. In *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*. Electronic Notes in Theoretical Computer Science, vol. 238, no. 5. Elsevier Science Publishers, pp. 81–99.

Dolstra, E., Visser, E. & de Jonge, M. (2004) Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*. IEEE Computer Society, pp. 583–592.

Feldman, S. I. (1979) Make—a program for maintaining computer programs, *Soft.—Prac. Exp.*, **9**(4), 255–265.

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

FreeBSD Project. (2009) *FreeBSD Ports Collection*. Available at: http://www.freebsd.org/ports/

Hallgren, T., Jones, M. P., Leslie, R. & Tolmach, A. (2005) A principled approach to operating system construction in Haskell. In *ICFP '05: Tenth ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 116–128.

Hart, J. & D'Amelia, J. (2002) An analysis of RPM validation drift. In *Proceedings of the 16th Systems Administration Conference (LISA '02)*. USENIX, pp. 155–166.

Heydon, A., Levin, R. & Yu, Y. (2000) Caching function calls using precise dependencies. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*. ACM, pp. 311–320.

Heydon, A., Levin, R., Mann, T. & Yu, Y. (March 2001). *The Vesta Approach to Software Configuration Management*. Tech. Rep. Research Report 168. Compaq Systems Research Center.

Hudak, P. (1989) Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, **21**(3), 359–411.

Kanies, L. (2003) ISconf: Theory, practice and beyond. In *Proceedings of the 17th USENIX Conference on System Administration (LISA '03)*. USENIX, pp. 115–124.

Pendry, J.-S. & McKusick, M. K. (1995) Union mounts in 4.4BSD-Lite. *Proceedings of the USENIX 1995 Technical Conference*. USENIX, pp. 25–33.

Peyton Jones, S. (1992) Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *J. Funct. Program.*, **2**(2), 127–202.

Peyton Jones, S. (ed). (2004) *Haskell 98 Language and Libraries: The revised Report*. Cambridge University Press.

Schneier, B. (1996) *Applied Cryptography*. 2nd ed. John Wiley & Sons.

Sloane, A. M. (2002) 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: IEEE Computer Society, pp. 281–289.

Stevens, W. R. & Rago, S. A. (2005) *Advanced Programming in the UNIX Environment*. 2nd ed. Addison-Wesley.

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

Traugott, S. & Brown, L. (2002) Why order matters: Turing equivalence in automated systems administration. In *Proceedings of the 16th Systems Administration Conference (LISA '02)*. USENIX, pp. 99–120.

Tucker, D. B. & Krishnamurthi, S. (2001) Applying module system research to package management. In *Tenth International Workshop on Software Configuration Management (SCM-10)*.