Under consideration for publication in J. Functional Programming

NixOS: A Purely Functional Linux Distribution

EELCO DOLSTRA
Delft University of Technology, The Netherlands
(*e-mail:* e.dolstra@tudelft.nl)

ANDRES LÖH
Utrecht University, The Netherlands
(*e-mail:* andres@cs.uu.nl)

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

Contents

1 Introduction

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

In this article 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 previous configurations are not overwritten. Perhaps most importantly, statelessness makes configuration actions predictable: they do not mysteriously fail because of some unknown aspect of the state of the system.

We have previously shown how *package management* – the installation and management of software packages – can be done in a purely functional way, in contrast to the imperative models of conventional tools such as RPM (Foster-Johnson, 2003). This concept was implemented in the Nix package manager (Dolstra *et al.*, 2004; Dolstra, 2006), summarised in Section 3. In this article 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/cj22mw17...-linux-2.6.23.17` 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. We then come to the contributions of this article:

- We discuss in detail the lazy purely functional *Nix expression language*, used to build packages and system configurations. We show why purity and laziness are essential features in this domain (Section 4).
- We show how a full-featured Linux distribution (NixOS) can be built and configured in a declarative way using principles borrowed from purely functional languages (Section 5).
- We present a *module system* for NixOS that allows separation of concerns and convenient extensibility (Section 6).
- We measure the extent to which NixOS and Nix build actions are pure (Section 7).

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

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

Statefulness also makes it hard to support multiple versions of a package. If two packages depend on conflicting versions of /usr/bin/perl, we cannot satisfy both at the same time. Instead, we would have to arrange for the versions to be placed under different filenames, which requires explicit actions from the packager or the administrator (e.g., install a Perl version under /usr/local/perl-5.8). This also applies to configuration files. 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 than 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.

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.

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

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

Second, RPM dependency specifications are *nominal*, that is, they specify just the name (and possibly some version constraints) of each dependency, e.g. Requires: python $>=$ 2.4. A package with this dependency will build on any system where python with a sufficiently high version is registered in RPM's database; however, the resulting binary RPM has no record of precisely *what* instance of python was used at build time, and therefore there is no way to deterministically reproduce it. 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 fact that a running system is thus the result of many *ad hoc* stateful actions makes it hard to reproduce (part of) a configuration on another machine, and to roll back configuration changes.

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

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

- It is difficult to allow *multiple versions* of a package on the system at the same time, or to run multiple instantiations of a service (such as a web server).
- There is *no traceability*: the configuration of the system is a result of a sequence of stateful transformations that are not under the control of a configuration management system. This makes it hard to reproduce a configuration elsewhere. The lack of traceability specifically makes *rollbacks* much harder.
- Since configurations are the result of stateful transformations, there is *little predictability*. For instance, upgrading a Linux or Windows installation 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 such as Haskell (Hudak, 1989). In such languages, the result of a function depends only on its inputs, and variables are immutable. This suggests that the deployment problems above go away if we can somehow move to a *purely functional* way to store software packages and configuration data.

3 Purely functional package management

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

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

Figure 1 shows the Nix expression for the xmonad package, a tiling X11 window manager written in Haskell (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 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*. mkDerivation is a function that captures this commonality, allowing packages to be written succinctly. For instance, Figure 2 shows the function that builds the xmessage package. The xmonad package, being Haskell-based, does not build in this standard way, so mkDerivation allows the various phases of the package build 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 emphasise that mkDerivation is just a function that abstracts over a common build pattern, not a language construct: functions that abstract over other build patterns can easily be written. For instance, the xmonad package makes use of Haskell's Cabal build system (http://haskell.org/cabal/). It therefore makes sense to extract the Cabal build commands (all the invocations of ./Setup) into a Cabal-specific wrapper around mkDerivation, thereby allowing the expression specific to xmonad to be reduced to the same conciseness as the xmessage example. Due to the functional nature of the Nix expression language, nearly all recurring patterns can be captured in functions in this way. (Indeed, the Nix Packages collection provides a function to build Cabal-based packages.)

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

let version = "0.5"; in

stdenv.mkDerivation [2] (rec {

  name = "xmonad-${version}";

  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. 1. xmonad.nix: Nix expression for xmonad

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 buildInputs environment variable (which is used by stdenv to generically set up other environment variables such as the C include file search path and the linker search path) will contain the paths to the ghc and X11 dependencies [3]. Likewise, the path to the xmessage package is substituted in the source file XMonad/Core.hs [5]. 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. (fetchurl may seem an impure function, but because the output is practically guaranteed to

```
{ 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. 2. xmessage.nix: Nix expression for xmessage

have a specific content by the cryptographic hash, which Nix verifies, it is pure as far as the purely functional deployment model is concerned.)

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

Since the expression that builds xmonad is a function, it must be called with concrete values for its arguments to obtain an actual derivation. This is the essence of the purely functional package management paradigm: the function can be called any number of times with different arguments to obtain different instances of xmonad. Just as multiple calls to a function in a purely functional language cannot "interfere" with each other, these instances are independent: 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 3 shows an attribute set containing several concrete derivations resulting from calls to package build functions. The attribute xmonad is bound to the result of a call to the function in Figure 1 [8]. Thus, xmonad is a concrete derivation that can be built. Likewise, the arguments are concrete derivations, e.g. xmessage is the result of a call to the function in Figure 2.

The end user can now install xmonad by doing[1]

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

which 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 to the output of the xmonad derivation.

Nix is available at http://nixos.org/, along with the Nix Packages collection (Nixpkgs), a large collection of Nix expressions for nearly 2000 packages.

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

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

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

Nix accomplishes this by storing packages as immutable 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 1, plus some default attributes added by the mkDerivation function). These typically include the sources, the build commands, the compilers used by the build, library dependencies, and so on. This is recursive: for instance, the sources of the compiler also affect the hash. Nix computes this path and passes it through the environment variable out to the build script.

Figure 4 shows a part of a Nix store containing xmonad and some of its build time and runtime dependencies. The solid arrows denote the existence of *references* between files, i.e., the fact that a file in the store contains the path of another store object. For instance, as is required for dynamically linked ELF executables (TIS Committee, 1995), the xmonad executable contains the full path to the dynamic linker ld-linux.so.2 in the Glibc package. Likewise, xmonad contains the full path to the xmessage program because we compiled it into the executable (at point 5 in Figure 1).

Figure 4 also shows the notion of *profiles*, which enable atomic upgrades and rollbacks and allow per-user package management. Names that are *slanted* denote symlinks, with dotted arrows denoting symlink targets. The user has the directory /nix/var/nix/profiles/

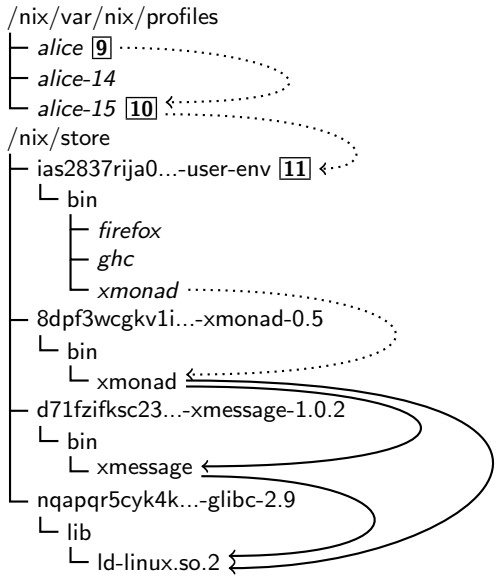

Fig. 4. The Nix store, containing xmonad and some of its dependencies, and profiles

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 4.
- Since actions such as upgrading or downgrading to a different version are non-destructive, the previous versions are not touched. Therefore, it is possible to roll back to previous versions (unless the garbage collector has explicitly been invoked in the meantime). Also, since upgrades and rollbacks are *atomic*, the user never sees a half-updated package; the system is never in an inconsistent state at any point during the upgrade.
- 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 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. These relations can not only easily be expressed in Nix – the upgrade path is also extremely painless, as the old versions remain usable in parallel for as long as desired. The system will always be consistent and the different versions do not interfere.

4 The Nix expression language

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

4.1 Syntax

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

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

String literals String literals can be enclosed in double quotes (`"string"`) or two single quotes (`''string''`). Both forms of strings can span multiple lines and allow *interpola-*

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

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

attribute path
ap ::= x.ap | x

formals
fs ::= x, fs | x | ...

operators
•  ::= == | != | && | || | -> | // | +
```

Fig. 5. Syntax of the Nix expression language

tion: arbitrary expressions can be inserted into the middle of strings by writing ${expr} (such as at [5] in Figure 1). Interpolations are desugared to string concatenations.

The second form, surrounded by two single quotes, is known as an *indented string*. In this form, a number of whitespace characters is removed from the start of each line equal to the indentation of the *least-indented* line. For instance, four spaces are removed from the start of each line in the string at [4] in Figure 1. Indented strings allow multi-line strings to naturally follow the indentation of the Nix expression. They are very convenient for including shell scripts in a Nix expression. Shell language itself makes frequent use of quoting (with both single and double quotes) and escaping via \, but rarely contains the '' combination. Therefore, large chunks of shell scripts can be included in a Nix expression almost literally using indented strings.

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

Paths Next to strings and URIs, Nix also handles path literals that are absolute or relative Unix paths and are given without surrounding quotes. Paths are treated slightly differently from strings during evaluation. When used in a derivation attribute, the path is copied to the Nix store, and the resulting path in the store is passed to the derivation.

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

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

There is some syntactic sugar for writing nested attribute sets, which occur quite frequently in NixOS: a nested set such as `{ a = { b = { c = 1; }; d = 2; }; }` can also be written as `{ a.b.c = 1; a.d = 2; }`.

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

Functions Anonymous functions are introduced using a colon to separate the argument from the body. The identity function can be written as `x : x`, the constant function in curried form as `x : y : x`. For functions taking attribute sets as arguments, there is a special syntax { *formals* } : e that allows pattern-matching on specific attributes by name (for instance used in the `xmonad` expression in Figure 1). Each formal parameter is the name of an attribute that must be present in the attribute set passed to the function. The list of formal parameters can end with the special ... (ellipsis) literal to denote that any additional attributes will be ignored; otherwise, such attributes cause a runtime error. Like in ML or Haskell, function application is denoted by juxtaposing two expressions.

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

The attribute set passed to the `derivation` function must define the attributes `system`, `name`, and `builder`. The attribute `builder` is interpreted as the invocation of a program that produces the package. The program is run in a very restricted build environment, but that environment is influenced by the other attributes in the attribute set passed to `derivation`: each attribute is converted into an environment variable of the same name. What exactly is passed depends on the type of the attribute. Strings, URIs and natural numbers are passed verbatim. A path causes the referenced file to be copied to the store, and the corresponding store path is passed in the environment variable. That way, sources, configuration files, patches and other inputs to the build will all be part of the Nix store. If the attribute is another derivation, then it becomes a dependency of the current derivation

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

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

$$\frac{\mathsf{x} \notin \mathrm{dom}\ (\vec{\mathsf{b}}_1 \cup \vec{\mathsf{b}}_2)}{\mathtt{let}\ \vec{\mathsf{b}}_1\ /\ \vec{\mathsf{b}}_2\ \mathtt{in}\ \mathsf{e} \rightarrow (\mathtt{rec}\ \{\vec{\mathsf{b}}_1; \mathsf{x} = \mathsf{e}\ /\ \vec{\mathsf{b}}_2\})\mathsf{.x}} \quad \text{(let)}$$

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

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

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

Fig. 6. Evaluation rules of the Nix expression language

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

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

Fig. 7. Matching

– the dependency is built first, and its output path is passed in the environment variable. Lists are flattened into space-separated strings. Finally, Booleans are passed as the empty string (for false) or 1 (for true).

4.2 Semantics

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

Next to syntactic transformation on Nix expressions, evaluation in Nix causes build actions to take place, such as described for the built-in function `derivation` above. Note that we see the build action as part of the evaluation, not a side-effect, as the Nix store can be seen as an on-disk extension of the "heap" of our language.

Listing all the reduction rules, especially for all the built-in functions and operators, would go beyond the scope of this article. We therefore give a few representative evaluation rules in Figure 6 that deal with functions and attribute sets. These rules are implemented more or less directly in the Nix expression evaluator, using *maximal laziness* to obtain an efficient evaluator (Dolstra, 2008).

Desugaring of inheritance For non-recursive attribute sets, the binding `inherit x` is syntactic sugar for a binding `x = x`. If a source is specified, for instance `inherit (e) x`,

the desugaring is x = e.x. Multiple attributes can be inherited in a single statement. This is transformed into multiple bindings.

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

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

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

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

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

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

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

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

4.3 Design decisions

After having described the Nix expression language and its operational semantics, let us now point out three concepts that we believe are essential to the language: purity, laziness,

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

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

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

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

Because purity is so central to NixOS, we discuss in Section 7.2 how we ensure that build actions are pure.

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

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

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

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

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

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

The reader may wonder if the Nix expression language could have been implemented as an embedded DSL (e.g., in Haskell). For instance, Sloane (2002) embedded a similar build language (based on Odin (Clemm, 1986)) in Haskell. This is certainly possible, but it would be painful to use: for instance, features such as string interpolation or functions over attribute sets would not be available.

5 NixOS

In Section 3 we saw that the purely functional approach of the Nix package manager solves many problems that plague "imperative" package management systems. We have previously used Nix as a package manager under existing Linux distributions and other operating systems, such as Mac OS X, FreeBSD and Windows to deploy software alongside the "native" package managers of those systems. In this section we introduce NixOS, a Linux distribution entirely built on Nix.

5.1 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 in an atomic action, they can just as easily build any file or directory tree so long as it can be built in a pure way. For instance, most configuration files in /etc in a typical Unix system do not change dynamically at runtime and can therefore be built by a derivation.

For example, the expression in Figure 8 generates a simple configuration file (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 set 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

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

Fig. 8. Nix expression to build sshd_config

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 *package*: depending on a user configuration option, the configuration file either does or doesn't reference xauth. This kind of dependency between a configuration file and a software package is generally not handled by conventional package managers, since they don't deal with configuration files in the same formalism as packages: packages can have dependencies, but configuration files cannot. In Nix, they are all derivations and therefore treated in the same way. As a concrete consequence, the xauth package won't be garbage-collected if sshd_config refers to it – an important constraint on the integrity of the system.

When we build this expression, a possible result might be

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

It is worth noting that due to laziness, xauth will be built if and only if config.services.sshd.forwardX11 is true. Thus, the *laziness of the Nix expression language directly translates to laziness in building packages*. This is a crucial feature: while xauth is a tiny package, many NixOS configuration options trigger huge dependencies. For instance, the option services.xserver.enable = true will cause a dependency on the X11 server (large), while services.xserver.desktopManager.kde4.enable = true will bring in the K Desktop Environment (very large).

5.2 Using NixOS

The rest of the system – the kernel, boot scripts, configuration files, and so on – are built in a similar manner. In Section 6, we will discuss the implementation and organisation of the Nix expressions that constitute NixOS in detail. First, however, we will give a high-level, user view of the system 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.

```
{ config, pkgs, ... }: 12

{
  boot.grubDevice = "/dev/sda"; 13
  boot.kernelModules = ["fuse" "kvm-intel"]; 14

  fileSystems = 15
    [ { mountPoint = "/";
        device = "/dev/disk/by-label/nixos";
      }
      { mountPoint = "/home";
        device = "/dev/disk/by-label/home";
      }
    ];

  swapDevices = [ { device = "/dev/disk/by-label/swap"; } ]; 16

  environment.systemPackages = [ pkgs.firefox ]; 17

  services.sshd.enable = true; 18
  services.sshd.forwardX11 = true; 19

  services.xserver.enable = true; 20
  services.xserver.videoDriver = "nvidia";
  services.xserver.desktopManager.kde4.enable = true;
}
```

Fig. 9. /etc/nixos/configuration.nix: NixOS configuration specification

Reconfiguring The user specifies the desired configuration of the system in a Nix expression, /etc/nixos/configuration.nix, that contains a nested attribute set specifying all configuration pertaining to the user's system. This file is imported by the NixOS expressions and affects the resulting derivations. Figure 9 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 18, and that it should support forwarding of X11 connections 19. As we saw in Figure 8, 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 15; the swap device 16; the disk on which the *Grub boot loader* (see Section 6.1) is to be installed 13; that we need certain Linux kernel modules that are not loaded by default, such as the kvm-intel module to support hardware virtualisation 14; that Mozilla Firefox should be built and made available to users 17; and that we want a graphical environment, namely the KDE desktop environment 20.

(The reader may notice that configuration.nix is a *function* 12. 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 6 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
```

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 6, 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.

It is also possible to build the new system, but delay its activation until the next reboot:

```
$ nixos-rebuild boot
```

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 might 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 applies the nix-env –rollback command 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 10). 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. A system can be rolled back without rebooting by doing nix-env --rollback -p /nix/var/nix/profiles/system and running the activation script.

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
```

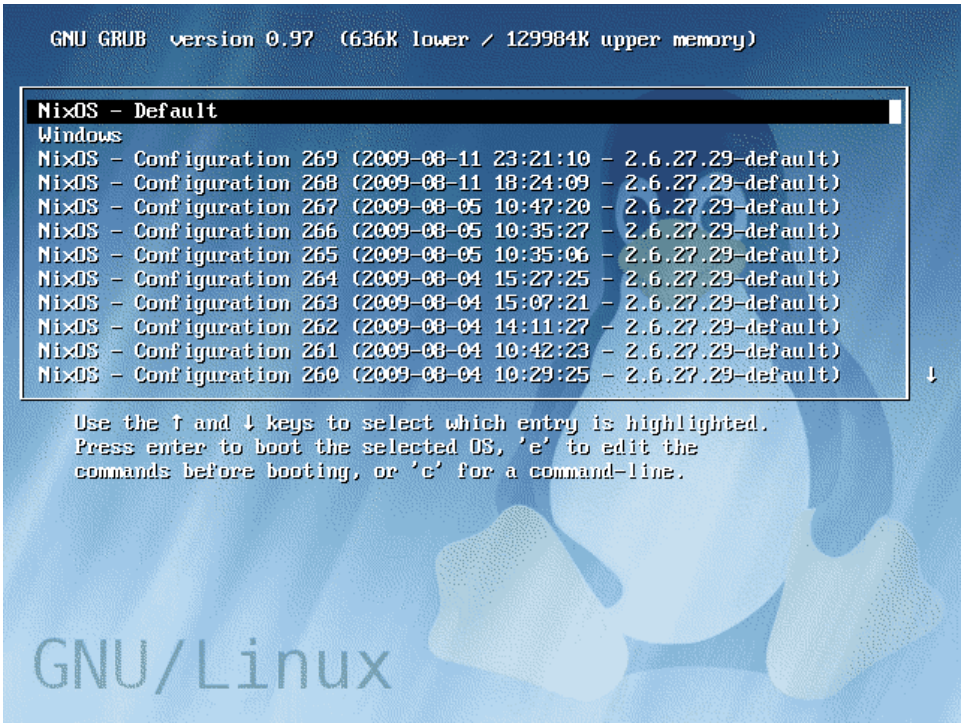

Fig. 10. The Grub boot menu for a NixOS machine

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 following command:

```
$ nixos-rebuild test
```

builds and activates a new configuration, but does not install it 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:

```
$ 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 following:

```
$ 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 as follows:

```
$ ./result/bin/run-nixos-vm
```

which will 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 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 6.6, 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.

1. Boot the system from the NixOS installation CD.
2. Partition the target disk and create a root filesystem for NixOS.
3. Mount the target root filesystem on /mnt.
4. Declaratively specify the desired configuration of the system in /mnt/etc/nixos/-configuration.nix.
5. Run the nixos-install command, 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 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 nice, graphical installer. The same applies to the reconfiguration of an installed system: the manipulation of configuration.nix and running of nixos-rebuild can easily be made into the back-end technology of a user-friendly configuration tool.

6 Implementation

In this section, we describe the implementation of NixOS in greater detail.

6.1 Anatomy of a Linux system

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. So what exactly are the parts that constitute a Linux system?

Figure 11 shows the dependency graph of a small subset of the derivations that build NixOS. The top-level derivation is the attribute system in /etc/nixos/nixos/default.nix, 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 (see below); dotted nodes are scripts; dashed nodes are various helper derivations that compose logical parts of the system; and all other nodes are software packages. However, this distinction is entirely conceptual. As far as Nix is concerned, they are all derivations: pure, atomic build actions.

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

- The Linux kernel (kernel), along with external kernel modules (drivers and other kernel extensions in the Linux model) such as the NVIDIA graphics drivers (nvidia-Drivers) and the Intel Wireless drivers (iwlwifi) that depend on it. These are part of Nixpkgs. A very nice consequence of the purely functional approach is that an upgrade of the kernel (i.e., a change to the Nix expression that builds the kernel) will trigger a rebuild of all dependent external kernel modules. This is in contrast to many other Linux distributions, where a common scenario is that one upgrades the kernel and then discovers that the X display server will not start anymore because the NVIDIA drivers were not rebuilt. This is not possible here. (Of course the external modules might not be compatible with the new kernel, but such a problem usually manifests itself at build time; and one cannot activate a new configuration unless *all* of it builds successfully.)
  A small wrapper component (modulesTree) combines (through symlinks) the kernel modules from the kernel package with those from the external module packages in a single directory tree. This is necessary because the kernel module loading command (modprobe) expects all modules to live in a single directory tree. This is an example of how we circumvent the impure models of various tools and applications: modprobe espouses an impure model where modules from various packages are statefully installed in an existing directory (/lib/modules/*kernel-version* on conventional Linux distributions). modulesTree turns this into a purely functional discipline.
- The derivation initrd builds the *initial ramdisk* necessary for booting the system. 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. The latter contains a small root filesystem containing everything necessary to allow the real root filesystem to be mounted. Notably, it must contain any kernel modules for the hardware containing the filesystem (e.g. drivers for SCSI or USB), the filesystem itself (e.g. ext3) and perhaps other modules such as network drivers for remote booting. The initrd thus is not fixed, but depends on the hardware configuration of the user. It therefore contains a copy of the closure under the module dependency relation of the set of modules needed.
  Furthermore, the initial ramdisk contains a boot script called stage1Init, which loads the kernel modules, runs the filesystem checker fsck (in the e2fsprogs package)

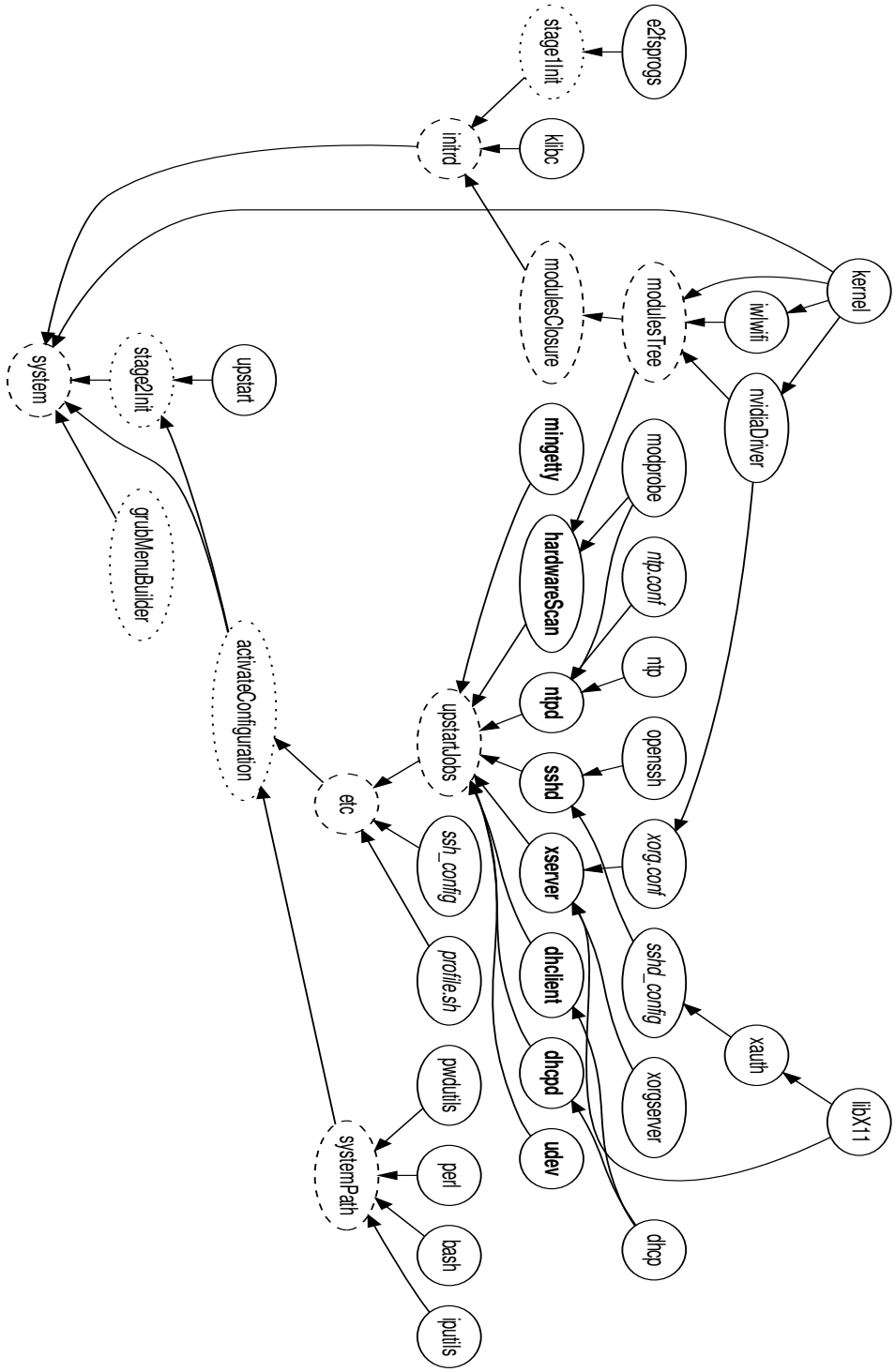

Fig. 11. A small part of the dependency graph of the derivations that build NixOS

if needed, mounts the root filesystem, and passes control to the final boot script stage2Init.

- The stage 2 initialisation script (stage2Init) performs boot-time initialisation and runs the 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 must be created (see below), and so on. This is done by the *activation script* (built by the derivation activateConfiguration).
- 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 (http://upstart.ubuntu.com/), 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 when networking comes up. Each Upstart job is generated by a derivation, e.g. sshd or xserver in the graph.
- Most generated configuration files are used directly from the Nix store. For instance, the sshd Upstart job starts the sshd process with command-line arguments containing the full path to the sshd_config file in the Nix store (see Figure 13 below). Thus, such files do not have to be visible under "global" paths such as /etc/ssh/sshd_config, as would be the case in conventional Unix systems.

  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 isn't dealt with in the functional model at all, such as the system password file /etc/passwd or the DNS configuration file /etc/resolv.conf, which must be modified at runtime. However, others are static (i.e., only change as a result of explicit reconfigurations) but still crosscutting, 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 and stored in the Nix store. The etc derivation in Figure 11 takes as inputs the files that are needed in /etc and combines them into a tree of symlinks. Later, the activation script (described below) copies these symlinks into /etc.

  Note that the use of "global" files such as those in /etc compromises the purely functional approach, as the targets of the symlinks in /etc are stateful. We discuss the extent of this problem in Section 7.2.
- systemPath is a set of symlinks to selected packages and is placed in the users' PATH environment variable. Since it is constructed by a Nix expression, it enables declarative package management. By contrast, manual package management actions such as nix-env -i (Section 3) are stateful and don't use a declarative specification to define the set of installed packages.
- Finally, the system derivation itself simply creates symlinks to its inputs, e.g. $out/kernel links to the kernel image, $out/activate links to the activation script, and so on.

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

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

build the entire configuration, including all software dependencies, configuration files, Upstart jobs, etc., leaving a symlink result in the current directory; and then run the activation script. (This is essentially what the user command nixos-rebuild test does; nixos-rebuild switch in addition regenerates the Grub boot menu.) 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.

6.2 Specifying a system, declaratively

NixOS thus consists of a set of Nix expressions describing derivations to build the various parts that constitute a Linux system: static configuration files, boot scripts, and so on. These build upon the software packages already provided by Nixpkgs. 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 or support for a certain class of hardware device), even though its *implementation* might cross-cut many "physical" parts (that is, derivations such as etc, systemPath, or upstartJobs).

As an example, Figure 12 shows the NixOS module that defines the OpenSSH daemon system service. A NixOS module defines a nested attribute set config [22] containing *option definitions*. These option definitions can be used by other modules. For instance, the sshd module *uses* (among others) the option services.sshd.enable [23], which should be defined by another module, and *defines* the options users.extraUsers, jobs, 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 script that sets up the firewall. Thus, definitions across many modules contribute to the derivation that generates the firewall[2]. Each option must be *declared* in some module before it can be *defined* by others. Modules declare options in their options attribute [21].

The most important option defined by the sshd module is jobs, which *declaratively* specifies the Upstart job that runs the sshd daemon, rather than using concrete Upstart syntax. (Abstracting away from the Upstart syntax in this manner allows us to switch to a different init system, or to build test scripts for system services to allow them to be run

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

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

let
  sshdConfig = pkgs.writeText "sshd_config" (elided; see Figure 8);
in {

  options [21] = {
    services.sshd.enable = pkgs.lib.mkOption {
      default = false;
      description = "Whether to enable the Secure Shell daemon.";
    };
    services.sshd.forwardX11 = pkgs.lib.mkOption {
      default = true;
      description = "Whether to allow X11 connections to be forwarded.";
    };
  };

  config [22] = pkgs.lib.mkIf config.services.sshd.enable [23] {

    users.extraUsers = pkgs.lib.singleton
      { name = "sshd";
        uid = config.ids.uids.sshd;
        description = "SSH privilege separation user";
        home = "/var/empty";
      };

    jobs = pkgs.lib.singleton [24]
      { name = "sshd";
        description = "OpenSSH server";

        startOn = "network-interfaces/started";
        stopOn = "network-interfaces/stop";

        preStart =
          ''
            mkdir -m 0755 -p /etc/ssh
            if ! test -f /etc/ssh/ssh_host_dsa_key; then
                ${pkgs.openssh}/bin/ssh-keygen -f /etc/ssh/ssh_host_dsa_key ...
            fi
          '';
        exec = "${pkgs.openssh}/sbin/sshd -D ... -f ${sshdConfig} [25]";
      };

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

Fig. 12. sshd.nix: NixOS module for the OpenSSH server daemon

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

start on network-interfaces/started
stop on network-interfaces/stop

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 \
        -f /etc/ssh/ssh_host_dsa_key ...
  fi
end script

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

respawn
```

Fig. 13. /nix/store/qjyn954j5jab...-upstart-sshd/etc/init/sshd.conf: Generated Upstart job for the OpenSSH server daemon

outside of Upstart.) For instance, the sshd module in Figure 12 defines a job 24 named sshd, with some shell code to be run before the job is started (preStart), and the actual command to start the service (exec). A different NixOS module, upstart.nix, maps the elements of the jobs configuration value to derivations that generate concrete Upstart jobs. Figure 13 shows the output of the generated Upstart job for sshd.

There are two points of interest in the sshd service in Figure 12 that are representative of NixOS in general. First, it is self-initialising and idempotent: at startup, 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.

Second, the configuration file for the daemon is not stored in /etc/ssh/sshd_config, as on other Linux distributions. Instead, as we have seen, it is generated and kept in the Nix store. The path of the generated sshd_config in the store is directly substituted in the generated Upstart job file through the string interpolation at 25. While this is not a big deal for the sshd service, it is very important for other services: for instance, it makes it trivial to run multiple web servers side-by-side, such as production and test instances, simply by instantiating the appropriate function multiple times with different arguments. The use of "global variables" such as /etc/ssh/sshd_config would preclude this.

Figure 14 shows a small part of the file hierarchy structure in the Nix store that resulted from building a NixOS system configuration with services.sshd.enable and services.sshd.forwardX11 both enabled. As in Figure 4, italicised names and dotted arrows denote symbolic links. Note that the output of the system derivation has a symlink to the output of the etc derivation; this enables the activation script to update /etc. 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 13, this file contains build-time generated references to the openssh package and

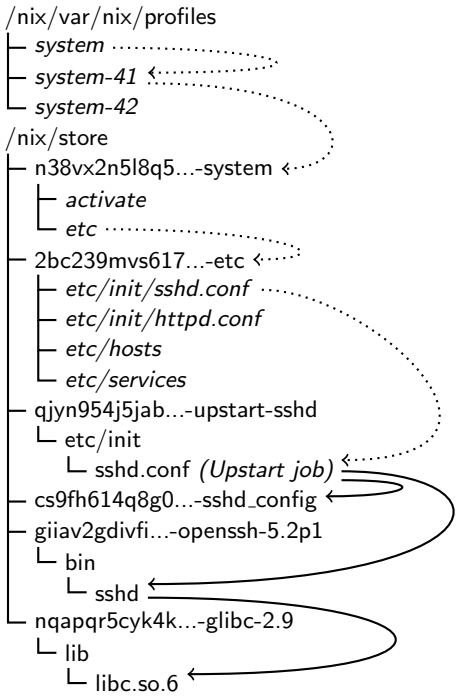

Fig. 14. Outputs of derivations related to the OpenSSH server daemon, as well as the system profile

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 14 will be garbage-collected as long as they are reachable from any of the symlinks in /nix/var/nix/profiles.

6.3 Module structure

The general structure of a NixOS module is a Nix expression of this form:

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

let ... local definitions ... in

{
  imports = [ paths of other NixOS modules to be included in the system configuration ];

  options = { nested attribute set of option declarations using mkOption };

  config = { nested attribute set of option definitions };
}
```

The config attribute sets returned by each module are *merged together* to form the final *system configuration*. The final configuration contains the option definitions from every

module and is passed as an *input* to each module through the config function argument[3]. 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 simply 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 networking.firewall.allowedTCPPorts = [22 80] (or [80 22], depending on the module order).

Likewise, the option attribute sets returned by each module are combined to form the set of permissible option definitions. Each option definition must have a corresponding option declaration, and each option declaration that is actually used must have at least one corresponding option definition, unless the option declaration specifies a default value.

Modules have the ability to add other modules to the system configuration through imports. However, the system configuration is unscoped: a module can use any value in config, regardless of whether those values are declared in a module in its imports list.

As a convenience, modules are called with the argument pkgs, which contains the Nix Packages collection, i.e., it is an import of all-packages.nix in Nixpkgs (Figure 3). This allows modules to refer to packages easily, without having to import Nixpkgs explicitly. The ... literal in the list of function arguments means that any other arguments to the function will be ignored. This allows the NixOS module calling convention to be extended in the future.

Another convenience is that trivial modules that do not declare any options or import any other modules can simply return a configuration attribute set directly, i.e.,

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

{
  nested attribute set of option definitions
}
```

This means that the end-user configuration file, /etc/nixos/configuration.nix (Figure 9), is a NixOS module. It can do anything that other NixOS modules can: for instance, define a jobs attribute to add additional Upstart jobs. Thus, the mechanisms for *implementing* and *configuring* NixOS are one and the same.

6.4 Evaluating the system configuration

Figure 15 illustrates how the system configuration is computed. Given a set of paths to NixOS modules – usually configuration.nix and most of the modules in /etc/nixos/nixos/modules – a function in /etc/nixos/nixos/lib/eval-config.nix performs the following steps:

[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 23 in Figure 12 refer to the final system configuration.

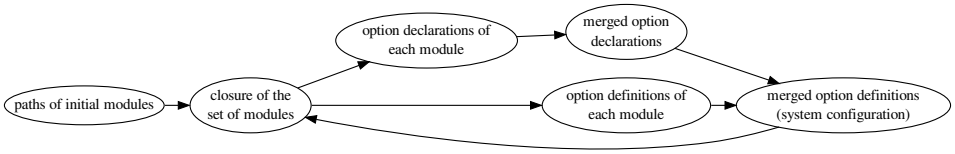

Fig. 15. Evaluation of the system configuration

1. We compute the closure of the set of modules under the imports relation, i.e., all modules listed in the imports attribute of a module are added to the set. Note that to get this information from a module, we have to call the module with the required config and pkgs arguments. The former is the system configuration returned by the last step below.
2. From each module in the closure, we obtain the options and config attributes.
3. The options attribute sets in each module are recursively merged. Multiple declarations of the same option cause an error. The result is a nested attribute set of option declarations (i.e., calls to mkOption).
4. We recursively "zip" the set of options with the config attribute sets in each module to compute the final configuration. For each option declaration *o* and the corresponding option definitions *cs*, the resulting merged option definition is *o*.merge (filter (x: x != ignore) *cs*. (The special value ignore is discussed below.)

Circularity Thus, the output – the system configuration – 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 12) 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 = ...;
        jobs = ...;
        networking.firewall.allowedTCPPorts = ...;
      }
    else
      { };
}
```

However, 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 12 at 23), 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;
  jobs =
    if config.services.sshd.enable then ... else ignore;
  networking.firewall.allowedTCPPorts =
    if config.services.sshd.enable then ... else ignore;
}
```

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.

6.5 Option types

The Nix expression language is dynamically typed. Thus, the language itself provides no mechanism to attach a static type to NixOS configuration options. Without a mechanism to specify the expected types of options, incorrectly typed values in NixOS modules (such as the user's configuration.nix) will lead to dynamic type error messages that are hard to understand, or worse, to incorrect output. Therefore, we have added a mechanism to declare option types, which are then dynamically enforced by the option merging code.

For instance, the following declares an option of Boolean type:

```
services.sshd.enable = pkgs.lib.mkOption {
  default = false;
  type = pkgs.lib.types.bool;
  description = "Whether to enable the Secure Shell daemon.";
};
```

Here, pkgs.lib.types.bool is a *type constructor* defined in Nixpkgs' standard library. Since the Nix language doesn't provide type declarations, it is implemented in Nix code essentially as follows:

```
bool = {
  name = "boolean";
  check = builtins.isBool;
  merge = fold lib.or false;
};
```

That is, types are encoded as *attribute sets* containing a *function* (t.check) that is applied to each option definition at runtime and returns true only if the value matches whatever notion of type we wish to express. In this case, it simply applies the builtin function isBool to check whether the argument is true or false. Note that this potentially reduces laziness: to check the type correctness of a NixOS option definition, its value must be computed. The attribute t.name is used in runtime error messages, while t.merge associates a default merge strategy for option definitions. For instance, fold lib.or false expresses that the merged value is true if any of the input values if true, and false otherwise. This is a good

strategy for bool since Booleans are typically used to enable features that are disabled by default.

Of course, this kind of dynamic type system allows arbitrarily complicated types to be expressed. For instance, higher-kinded types are possible, such as lists:

```
list = t: mkOptionType {
  name = "list of ${t.name}s";
  check = xs: builtins.isList xs && all t.check xs;
  merge = concatLists;
};
```

Thus, (pkgs.lib.types.list pkgs.lib.types.bool) declares the type of lists of Booleans.

Similarly, types can be assigned to attribute sets. This allows, for instance, to express the type of the fileSystems option in Figure 9: a list of sets containing string-valued attributes mountPoint and device. It is also possible to declare default values for missing attributes.

6.6 Module organisation

NixOS consists of dozens of modules. However, some play a pivotal role in the construction of the system. Here, we list some core modules that build key parts of the system, along with the most important options that they *define* (in their config set) or *declare* (in option).

modules/system/activation/top-level.nix:

Declares and defines:

- system.build.system: the top-level system derivation, which simply builds a directory in the Nix store containing symlinks to the principal components of a NixOS system (defined in other modules): notably the kernel, the initial ramdisk, the activation script, and the stage-2 init script. These are sufficient to allow the nixos-rebuild switch command 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).
  NixOS' top-level Nix expression, /etc/nixos/nixos/default.nix, simply evaluates the final system configuration as described in Section 6.4, and returns the value of the system.build.system option. The evaluation of that value then causes the entire system to be built.

modules/system/activation/activation-script.nix:

Declares:

- system.activationScripts: a list of script fragments to be included in the activation script. These can perform stateful initialisation that (by its nature) cannot be done as part of purely functional build actions, such as creating user accounts or creating state directories such as /var/log.

Declares and defines:

- system.build.activationScript: a derivation that concatenates the values of system.activationScripts together. This value is used by system.build.system.

modules/system/boot/stage-1.nix:

Declares and defines:

- system.build.bootStage1: a derivation that builds a shell script to performs all actions necessary to mount the filesystems containing / and /nix/store and start the second stage (system.build.bootStage2).
- system.build.initialRamdisk: a derivation that builds the initial ramdisk (see Section 6.1), a compressed archive containing the Nix closure of system.build.bootStage1 and the kernel modules needed to access the root filesystem.

modules/system/boot/stage-2.nix:

Declares and defines:

- system.build.bootStage2: a derivation that builds a shell script that performs the second stage of system initialisation. After performing activation actions that should only happen at boot time (such as clearing the contents of the runtime state directory /var/run), it runs the activation script, then executes Upstart to handle the further lifetime of the system.

modules/system/etc/etc.nix:

Declares:

- environment.etc: a list of attribute sets describing files in the Nix store that have to symlinked from /etc at activation time. For instance, the value

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

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

Defines:

- system.activationScripts: adds an activation scriptlet that creates the symlinks declared in environment.etc.

modules/system/upstart/upstart.nix:

Declares:

- jobs: a list of attribute sets declaring Upstart jobs, as in Figure 12.

Defines:

- environment.etc: the result of the expression map makeJob jobs, where the function makeJob returns {source = pkgs.writeText "upstart-*job.name*" *...(the text of the Upstart job)...*; target = "init/*job.name*";}.

modules/config/system-path.nix:

Declares:

- environment.systemPackages: a list of Nix packages to be made available to users of the system. For instance, the programs in the bin/ and sbin/ subdirectories of each package should appear in the users' PATH environment variable.

modules/config/users-groups.nix:

Declares:

- users.extraUsers: a list of attribute sets defining additional user accounts to be created by the activation script. For instance, the sshd module in Figure 12 defines a user account needed by the sshd daemon.

Defines:

- system.activationScripts: adds an activation scriptlet that creates the user accounts and groups defined by the options above if they do not already exist; if they do exist but differ from the declarations, they are updated.

modules/programs/bash/bash.nix:

The GNU Bourne-Again Shell (bash) is the shell of the system. When users log in, the shell reads the file /etc/bashrc, providing an opportunity to set various system-wide settings such as global environment variables.

Declares:

- environment.shellInit: a string containing shell code to be executed when users log in. It has a merge function that concatenates multiple definitions of this option.

Defines:

- environment.etc: causes /etc/bashrc to be symlinked to a generated shell script that contains (among other things) the value of environment.shellInit.

modules/config/timezone.nix:

This trivial module illustrates the use of environment.shellInit.

Declares:

- time.timeZone: the timezone of the system, such as CET.

Defines:

- environment.shellInit: a shell fragment that sets the TZ environment variable (used by programs such as ls to display timestamps in the user's local time zone) to the value of time.timeZone.

modules/security/setuid-wrappers.nix:

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 program 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, the purely functional model disallows overwriting of files, and only deletes files if they can be safely garbage-collected.

Since programs in the Nix store cannot have these permissions, we push the problem 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.

Declares:

- security.setuidPrograms: a list of programs for which wrappers must be created at activation time, along with the desired permissions.

Defines:

- system.activationScripts: adds an activation scriptlet that creates the setuid wrapper programs in /var/setuid-wrappers and deletes old ones.

Building disk images and virtual machines Normally, building a NixOS configuration entails evaluating the option system.build.system, 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 that build a system in rather different ways. Notably, these are the generation of CD-ROM and DVD images containing NixOS, and the generation of NixOS virtual machines.

modules/installer/cd-dvd/iso-image.nix:

Declares and defines:

- system.build.isoImage: a derivation 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.system (i.e., the usual top-level derivation), along with some 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/default.nix -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 isn't 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.

modules/installer/cd-dvd/installation-cd-graphical.nix:

This module and the next illustrate the usefulness of the modular specification of system configurations. The first 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:

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

modules/installer/cd-dvd/installation-cd-minimal.nix:

By contrast, this module specifies a small CD image without a GUI and related features, e.g.,

```
{ services.xserver.enable = false;
  services.sshd.forwardX11 = false;
  services.dbus.enable = false; # because it depends on libX11
}
```

modules/virtualisation/qemu-vm.nix:

Defines:

- system.build.vm: a derivation to produce a shell script that starts a virtual machine running the given NixOS system configuration. 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} ..."
```

- fileSystems: this option is set to

```
[ { 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 don't have to generate a disk image containing the closure of system.build.system, which would typically be hundreds of megabytes large. As a result, nixos-rebuild build-vm (Section 5.2) can build VMs for test purposes very quickly.

7 Evaluation

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

7.1 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 August 2009, the Nix Packages collection (on which NixOS builds) contains Nix expressions for some 1959 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[4]): the build graph rooted at the top-level system attribute in /etc/nixos/nixos/default.nix consists of 990 derivations (543 excluding fetchurl derivations) and 160 miscellaneous source files. The closure of the output of the system derivation (i.e., its runtime dependencies) consists of 449 store paths with a total size of 1639 MiB. 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 498 Nix expression files in the NixOS and Nixpkgs trees, and takes 4.5 seconds of CPU time on a Core 2 Duo 7700.

7.2 Purity

The goal of NixOS was to create a Linux distribution built and configured in a purely functional way. Thus build actions should be deterministic and therefore reproducible, and there should be no "global variables" like /bin that prevent multiple versions of packages and services to exist side-by-side. There are several aspects to evaluating the extent to which we reached those goals.

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

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

[4] Revision 16796 of the configuration at https://svn.nixos.org/repos/nix/configurations/trunk/misc/eelco-dutibo.nix.

Configuration data NixOS has many fewer configuration files in /etc than other Linux distributions. This is because most configuration files concern only a single daemon, which almost always has an option to specify the full path to the configuration file in the Nix store directly (such as sshd -f ${sshdConfig} in Figure 12). What remains is cross-cutting configuration files, which, as discussed in Section 5, are built purely but then symlinked in /etc by the configuration's activation script. The configuration above has just 47 such symlinks.

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

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

However, there is one case in NixOS and Nixpkgs of a library dependency that *must* be overridable at runtime and cannot be fixed statically: the implementation of 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 five years we haven't had a single instance of a package having an undeclared build-time dependency on another package *in the Nix store*, or having a runtime dependency on another package in the Nix store not detected by the reference scanner. However, with Nix under other Linux distributions or operating systems, there have been numerous instances of packages affected by paths outside the Nix store. We prevent most of those impurities through a wrapper script around GCC and ld that ignores or fails on paths outside of the store. However, this cannot prevent undeclared dependencies such as direct calls to other programs, e.g., a Makefile running /usr/bin/yacc.

Since NixOS has no /bin, /usr and /lib, the effect of such impurities is greatly reduced. However, even in NixOS such impurities can occur. For instance, we recently encountered a problem with the build of the dbus package, which failed when /var/run/dbus didn't exist. 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[5] 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.

[5] 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.

8 Related work

This article 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.

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 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; Clemm, 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).

9 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 We are investigating how to extend the Nix expression language with a type system. In the long term, we do not just want to check as much as possible of the current structural type system statically, but also introduce user-defined datatypes, such that for instance all variants of a single package such as GHC have the same type, and one cannot inadvertently pass a different package as a parameter where really GHC is expected. Currently, such a package fails at evaluation time. 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.

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

Finally, as noted in Section 7.2, our model assumes that builds are pure, but current operating systems cannot enforce this. An interesting idea would be to add support for truly pure builds, e.g., kernel modifications to support the notion of a "pure process": one that is guaranteed to give the same output for some set of inputs. This would mean, for instance, that the time() system call must return a fake value, network access is blocked, files outside of a specified set are invisible, and so on.

Acknowledgments This research was supported in part by NWO-JACQUARD project 638.001.201, *TraCE: Transparent Configuration Environments*, 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: Michael Raskin, Marc Weber, Ludovic Courtès, Sander van der Burg, Wouter den Breejen, Lluís Batlle, Yury G. Kudryashov, Tobias Hammerschmidt, Rob Vermaas and Martin Bravenboer. Eelco Visser and Merijn de Jonge contributed to the development of Nix. We also wish to thank the anonymous ICFP 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. *Pages 213–218 of: ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM.

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.

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

Clemm, Geoffrey. 1986 (Feb.). *The Odin system — an object manager for extensible software environments*. Ph.D. thesis, University of Colorado at Boulder.

Clemm, Geoffrey. (1995). The Odin system. *Pages 241–262 of: Selected papers from the ICSE SCM-4 and SCM-5 workshops on software configuration management*. Lecture Notes in Computer Science, no. 1005. Springer-Verlag.

Cosmo, Roberto Di, Zacchiroli, Stefano, & Trezentos, Paulo. (2008). Package upgrades in FOSS distributions: details and challenges. *Pages 1–5 of: HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*. New York, NY, USA: ACM.

DeTreville, John. (2005). Making system configuration more declarative. *HotOS X: 10th Workshop on Hot Topics in Operating Systems*. USENIX.

Dolstra, Eelco. 2005 (Nov.). Secure sharing between untrusted users in a transparent source/binary deployment model. *Pages 154–163 of: 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*.

Dolstra, Eelco. (2006). *The purely functional software deployment model*. Ph.D. thesis, Faculty of Science, Utrecht University, The Netherlands.

Dolstra, Eelco. (2008). Maximal laziness — an efficient interpretation technique for purely functional DSLs. *Eighth workshop on language descriptions, tools and applications (LDTA 2008)*. Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers. To appear.

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

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

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

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

Hallgren, Thomas, Jones, Mark P., Leslie, Rebekah, & Tolmach, Andrew. (2005). A principled approach to operating system construction in Haskell. *Pages 116–128 of: ICFP '05: Tenth ACM SIGPLAN International Conference on Functional Programming*. ACM.

Hart, John, & D'Amelia, Jeffrey. (2002). An analysis of RPM validation drift. *Pages 155–166 of: Proceedings of the 16th systems administration conference (LISA '02)*. USENIX.

Heydon, Allan, Levin, Roy, & Yu, Yuan. (2000). Caching function calls using precise dependencies. *Pages 311–320 of: ACM SIGPLAN '00 conference on Programming Language Design and Implementation*. ACM.

Heydon, Allan, Levin, Roy, Mann, Timothy, & Yu, Yuan. 2001 (Mar.). *The Vesta approach to software configuration management*. Tech. rept. Research Report 168. Compaq Systems Research Center.

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

Kanies, Luke. (2003). ISconf: Theory, practice, and beyond. *Pages 115–124 of: Proceedings of the 17th USENIX conference on system administration (LISA '03)*. USENIX.

Pendry, Jan-Simon, & McKusick, Marshall Kirk. (1995). Union mounts in 4.4BSD-Lite. *Proceedings of the USENIX 1995 Technical Conference*. USENIX.

Peyton Jones, Simon. (1992). Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of functional programming*, **2**(2), 127–202.

Schneier, Bruce. (1996). *Applied cryptography*. Second edn. John Wiley & Sons.

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

Stevens, W. Richard, & Rago, Stephen A. (2005). *Advanced programming in the UNIX environment*. Second edn. Addison-Wesley.

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

Traugott, Steve, & Brown, Lance. (2002). Why order matters: Turing equivalence in automated systems administration. *Pages 99–120 of: Proceedings of the 16th systems administration conference (LISA '02)*. USENIX.

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