Purely Functional System Configuration Management

Eelco Dolstra
Department of Information and Computing Sciences
Utrecht University
http://www.cs.uu.nl/~eelco

Armijn Hemel
Loohuis Consulting
armijn@uulug.nl

Abstract

System configuration management is difficult because systems evolve in an undisciplined way: packages are upgraded, configuration files are edited, and so on. The management of existing operating systems is strongly *imperative* in nature, since software packages and configuration data (e.g., /bin and /etc in Unix) can be seen as imperative data structures: they are updated in-place by system administration actions. In this paper we present an alternative approach to system configuration management: a *purely functional* method, analogous to languages like Haskell. In this approach, the static parts of a configuration — software packages, configuration files, control scripts — are built from pure functions, i.e., the results depend solely on the specified inputs of the function and are immutable. As a result, realising a system configuration becomes deterministic and reproducible. Upgrading to a new configuration is mostly atomic and doesn't overwrite anything of the old configuration, thus enabling rollbacks. We have implemented the purely functional model in a small but realistic Linux-based operating system distribution called NixOS.

1 Introduction

A system configuration is the composition of artifacts necessary to make computer systems performs their intended functions. Managing the configuration of a system is difficult because there are typically thousands of these interrelated artifacts that together make up a system (e.g., software packages, configuration data, or control scripts) and we need to manage the evolution of such a configuration.

To illustrate, and as a running example, Figure 1 shows a very small subset of the artifacts that make up a simple Linux system running an OpenSSH server and Apache (providing access to Subversion repositories). Boxes denote software components (including scripts), solid ellipses denote configuration files, and dashed ellipses are conceptual groupings of configuration aspects. An arrow $a \rightarrow b$ indicates that system component a has some kind of reference to component b, be it a dynamic linker dependency, a line in a configuration file, and so on.

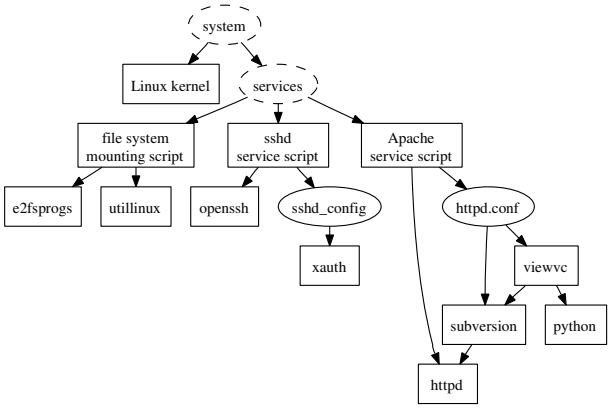

Figure 1: A small subset of a system configuration

In this paper we argue that existing configuration management tools — ranging from package managers such as RPM [7] to configuration tools such as Cfengine [2] — have an *imperative model* in a sense analogous to imperative programming languages such as C. That is, configuration actions such as upgrading a package or modifying a configuration file are *stateful*: they depend on and transform the state of the system.

Statefulness has a number of serious consequences:

- No *traceability*: if a configuration is the result of a sequence of imperative actions over time (some of which may have been performed manually), then there may not be a record of these actions that allows the configuration to be recreated on an empty machine. As a result, it is hard to *reproduce* configurations.

- No *predictability*: if a configuration action depends on an ill-defined initial state, then the end result may be unpredictable. This is why upgrading an operating

system is so much more error-prone than reinstalling it: the upgrade action transforms a poorly defined initial state, the result often being equally poorly defined, while the installation action starts from a well-defined initial state.

- Inability to run multiple configurations *side-by-side*. For instance, if we want to test some modification to the Apache server in Figure 1, then we could of course modify the httpd.conf of the production instance to run on a different port. But this would affect the production instance as well. Instead, we should copy httpd.conf, but this propagates through the dependency graph; for instance, we would have to copy the Apache service script as well to refer to the new httpd.conf.

- A special instance of the previous problem is the inability to *roll back* the system to a previous configuration. For instance, if we upgrade a set of RPM packages and the upgrade turns out to be less than desirable, then undoing the change is hard: we would have to revert to backups (but it may not be clear which files to restore) or install the previous RPM packages (if we know which ones!). Likewise, if we make a bad modification to a configuration file, we'd better have the file under version control to revert the change.

Analogous problems exist in imperative programming languages, such as the inability to reason about the result of function calls due to global variables or I/O. This was an important motivation for the development of purely functional programming languages [11], such as Haskell [12]. In those languages, the result of a function call only depends on the inputs of the function, and variables are immutable. This makes it easier to reason about the behaviour of a program. For instance, two function calls $f(x)$ and $f(y)$ can never interfere with each other (e.g., because of mutable global variables or I/O), and $x = y$ implies $f(x) = f(y)$. This property is known as *referential transparency*, and it is what's lacking in conventional system configuration tools. For instance, when a configuration file such as sshd_config has a reference to some path, say /usr/X11/bin/xauth, then the *referent* (the file being pointed to) is not constant. Thus a configuration action in one corner of the system can affect the behaviour in another. This is what causes problems such as the "DLL hell".

In this paper, we show that it is possible to do system configuration management in a purely functional way. This means that the static parts of a configuration — software packages, configuration files, control scripts — are built from pure functions and *never change after they have been built*. This gives a number of concrete advantages:

- A configuration can be realised (*built*) deterministically from a single formal description. This also means that a configuration can be reproduced easily on another machine.

- Upgrading a configuration is as safe as installing from scratch, since the realisation of a configuration is not stateful.

- Configuration changes are not destructive. As a result, we can always roll back to a previous configuration that hasn't been garbage collected yet.

To show that a purely functional model is feasible, we have implemented a small but realistic Linux-based operating system distribution called NixOS. The remainder of this paper shows the basic principles behind NixOS.

2 Purely functional package management

NixOS is based on Nix, a purely functional package management system [5, 6]. *Nix expressions* describe how to build immutable software packages from source. For instance, the following Nix expression is a *function* that builds Apache:

```
{stdenv, openssl}:

stdenv.mkDerivation {
  name = "apache-2.2.3";
  src = fetchurl {
    url = http://.../httpd-2.2.3.tar.bz2;
    md5 = "887bf4a85505e97b...";
  };
  buildCommand = "
    tar xjf $src
    ./configure --prefix=$out \
      --with-openssl=${openssl}
    make; make install";
}
```

Here, stdenv and openssl are function arguments representing dependencies of Apache (stdenv is a standard build environment: GCC, Make, etc.). When the function is called, it builds a *derivation*, which is an atomic build action. In this case, the build action unpacks, configures, compiles and installs Apache. All attributes specified in the derivation (such as src) are passed to the builder through environment variables. The out environment variable contains the target path of the package, discussed below. Dependencies such as src, stdenv and openssl are recursively built first.

To build a concrete Apache instance, we write an expression that calls the function:

```
apache = import ./apache.nix {
  inherit stdenv openssl;
}
stdenv = ...;
openssl = ...;
```

That is, the file containing the Apache function is imported and called with specific instances of stdenv and openssl, which are defined similarly. (inherit simply copies the values of stdenv and openssl from the surrounding lexical scope. The details of the language aren't very important here; the interested reader is referred to [5].) The user can now do

```
$ nix-env -f all-packages.nix -i apache
```

to install the Apache package.

The Nix expression language happens to be purely functional, but what really matters is that *the storage of packages is also purely functional*. Packages are built in the *Nix store*, a designated part of the file system, typically /nix/store. Each package is stored separately under a name that contains a 160-bit cryptographic hash of the inputs involved in building the package, e.g., /nix/store/5lbfaxb7...-openssl-0.9.8d or /nix/store/2m732xrk...-apache-2.2.3. This location is passed to the build script through the out environment variable. As a result, any change to any input causes a different path, so if we build the Nix expression, the package will be rebuilt in a different path in the store. Any previous installations of the package are left untouched, and so we prevent problems like the "DLL hell". On the other hand, if the path already exists, then we can safely skip rebuilding it.

Packages are made read-only after they have been built. This is why the Nix store can be called purely functional: the cryptographic hash in the path of a package is defined by the inputs to the package's build process, and the contents never change; thus there is a unique correspondence between the hash and the contents. This scheme has several important advantages [6], such as preventing undeclared build time dependencies, allowing detection of runtime dependencies, and allowing automatic garbage collection of unused packages.

3 System configuration management

We can quite naturally extend purely functional package management to purely functional system configuration management, simply by treating the other parts of a configuration — such as configuration files and control scripts — as packages.

For instance, consider the configuration file for the sshd daemon, sshd_config. We can make a trivial Nix expression that builds it in the Nix store:

```
{stdenv, xauth}:
stdenv.mkDerivation {
  name = "sshd_config";
  buildCommand = "
    echo 'X11Forwarding yes' > $out
    echo 'XAuthLocation \
      ${xauth}/bin/xauth ' >> $out";
}
```

(The construct ${xauth} places the store path of the xauth argument in the enclosing string. Also, the actual implementation uses more sophisticated configuration file templating mechanisms.)

Of course, since files in the Nix store are immutable, this only works for the static parts of a configuration. Mutable state (such as /etc/passwd or databases) falls outside the scope of this approach. Fortunately, most configuration data is fairly static; it doesn't change dynamically but only due to administrator action.

In the same way that we built sshd_config purely, we can build everything else that goes into a system, such as the configuration in Figure 1. Thus there are Nix expressions to build the kernel, the initial ramdisk (initrd) necessary for providing boot-time modules required by the kernel to mount the root file system, the boot scripts, the Upstart jobs[1], the X server plus its configuration, etc.

There is also a top-level Nix expression, system.nix, that builds the entire system configuration by calling the individual expressions that build specific parts of the system. The output of this expression is a package containing an *activation script* that makes the configuration the current configuration of the system. For instance, it modifies the Grub startup menu so that the system will boot with the new configuration the next time the system is booted. The Grub menu also contains all previous configurations that haven't been garbage collected yet, allowing the user to go back to old configurations very easily if there is a problem with the new configuration.

Of course, most configuration changes do not require the system to be restarted. A nice property of NixOS's purely functional model is that it allows the activation script to determine precisely which system services have to be restarted, simply by comparing the store paths of their Upstart job files. After all, due to the immutability of files in the Nix store, if an Upstart job with a certain

[1] Upstart is a event-based replacement for the classic /sbin/init (http://upstart.ubuntu.com/). It's responsible for running system startup scripts and monitoring system daemons.

name (e.g., sshd) has the same path in the new configuration as in the previous configuration, then it must be the same.

When the user changes anything to a Nix expression, the new configuration so described can be realised by

```
$ nix-env -f system.nix -i system
$ activate-configuration.sh
```

Since the building of a new configuration is non-destructive — it doesn't overwrite any existing files in the store — the user can roll back to any previous configuration (that hasn't been garbage collected yet) by running its activation script.

An important advantage of our approach is that since the entire system configuration is expressed in a single formalism, a user doesn't have to know how a specific configuration choice is implemented. Regardless of the configuration change that we perform — upgrading to a new version of OpenSSH, changing an sshd configuration option, upgrading to a new kernel — the change is realised in the same manner.

Another advantage of using a functional language is that it is easy to abstract over configuration choices. Rather than encoding those choices directly in a Nix expression, they can be passed as function arguments to the expression. For instance, whether to turn on X11 forwarding for sshd can be passed as a function argument:

```
{stdenv, xauth, forwardX11}:
stdenv.mkDerivation {
  name = "sshd_config";
  buildCommand = "
    ${if forwardX11 then
        "echo 'X11Forwarding yes' > $out
         echo 'XAuthLocation \
           ${xauth}/bin/xauth ' >> $out"
      else ""}";
}
```

This causes the line echo 'XAuthLocation ...' to be included in the build command only if forwardX11 is true. This has the additional advantage that due to lazy evaluation (derivations are only built when they are actually referenced), xauth is built only when forwardX11 is set. This kind of optimisation follows from the integration of package management and system configuration management into a single formalism.

Similarly, system.nix is a function that accepts a *configuration specification*, which is a hierarchical set of attributes specifying various system options. It passes these options on to the appropriate Nix expressions, e.g., services.sshd.forwardX11 is passed on to the function that builds the sshd service. Figure 2 shows a real example of a configuration specification corresponding to the configuration in Figure 1.

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

Figure 2: A simple configuration specification

4 Evaluation

Does the purely functional approach work? That is, to what extent can we eliminate the "global" namespace of files in /etc, /bin and so on and replace them with purely built files?

Quite well, in fact. With regard to software, NixOS has no /usr, /sbin, or /lib. There is only one file in /bin: namely, a symlink /bin/sh to a Bash instance in the Nix store. This is because many programs (such as Glibc's system() function) hard-code the location of the shell. Of course, we could patch all those programs, but instead we took the pragmatic route — a slight concession to the purely functional model. Apart from the symlink /bin/sh, all software resides in the Nix store. For example, once we had NixOS bootstrapped with just the single /bin/sh compromise, we were able to build Mozilla Firefox and all its dependencies all the way to Glibc and GCC purely. (Details of the NixOS bootstrap can be found in [9].)

How about configuration data in /etc? A lot of configuration data can be "purified" easily. For instance, a configuration file such as /etc/ssh/sshd_config can easily be generated in a Nix expression and used directly from the Nix store by also generating an Upstart job that calls

sshd with the appropriate -f argument to specify the Nix store path of the generated sshd_config.

However, there are some configuration files for which this is either not possible (e.g., because the path is hard-coded into binaries) or infeasible (because the configuration cross-cuts the system). Examples are /etc/services and /etc/resolv.conf. We *do* generate those files in Nix expressions, but the activation script of the configuration creates symlinks to them in /etc. The configuration in Figure 2 requires 12 symlinked files and directories in /etc, though we can almost certainly get rid of a number of these with little effort. In total, that configuration, when built, consists of 125 packages in the Nix store; its build-time dependency graph consists of 382 derivations.

5 Related Work

Our work on NixOS is an extension of our previous work on purely functional software deployment [6, 5]. We extended software deployment to service deployment in [4].

The Vesta configuration management system [10] has a purely functional language for build management. It may be possible to apply it to system configuration management.

The need to make system configuration more declarative has been felt by many. An approach inspired by Vesta is suggested in [3]. The present work could be seen as a realisation of that idea. Configuration tools such as Cfengine [2] and LCFG [1] are declarative, but stateful. For instance, Cfengine actions transform the current state of the system.

While to our knowledge this is the first attempt at purely functional system configuration management, there have been operating systems written *in* a functional language, such as House [8]. This is a rather orthogonal issue; the system configuration management of those OSes is entirely conventional.

6 Conclusion

We have shown that it is possible to implement an operating system with a purely functional configuration management model, where all software packages and configuration files are built from a formal description of the system using pure functions and are never changed after they have been built. Hudak [11] points out that people unfamiliar with functional languages often find it hard to believe that it is possible to live without assignments; likewise, it may not be entirely obvious that one can manage an operating system without destructively updating files. We have shown that this is in fact quite possible. However, more experience is necessary to see to which extent this approach is *practical*; our previous work on purely functional package management [5] suggests that it is.

NixOS is available from `http://nix.cs.uu.nl/nixos`.

Acknowledgements This research was supported by NWO/JACQUARD project 638.001.201, *TraCE: Transparent Configuration Environments*. We wish to thank Martin Bravenboer, Eelco Visser, Rob Vermaas and others for contributing to the development of Nix.

References

[1] P. Anderson and A. Scobie. LCFG: The next generation. In *UKUUG Winter Conference*, Feb. 2002.

[2] M. Burgess. Cfengine: a site configuration engine. *Computing Systems*, 8(3), 1995.

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

[4] E. Dolstra, M. Bravenboer, and E. Visser. Service configuration management. In *12th Intl. Workshop on Software Configuration Management (SCM-12)*, Sept. 2005.

[5] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In L. Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, Nov. 2004. USENIX.

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

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

[8] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Tenth ACM SIGPLAN Intl. Conf. on Functional Programming*, pages 116–128. ACM Press, 2005.

[9] A. Hemel. NixOS: the Nix based operating system. Master's thesis, Dept. of Information and Computing Sciences, Utrecht University, Aug. 2006. INF/SCR-2005-091.

[10] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN '00 Conf. on Programming Language Design and Implementation*, pages 311–320. ACM Press, 2000.

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

[12] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Apr. 2004.