The Purely Functional Software Deployment Model

Het puur functionele softwaredeploymentmodel (met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op woensdag 18 januari 2006 des middags te 12.45 uur

door

Eelco Dolstra

geboren op 18 augustus 1978, te Wageningen

Promotor:	Prof. dr. S. Doaitse Swierstra, Universiteit Utrecht
Copromotor:	Dr. Eelco Visser, Universiteit Utrecht

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). Dit proefschrift werd mogelijk gemaakt met financiële steun van CIBIT|SERC ICT Adviseurs.

Cover illustration: Arthur Rackham (1867–1939), *The Rhinegold & The Valkyrie* (1910), illustrations for Richard Wagner's *Der Ring des Nibelungen*: the gods enter Walhalla as the Rhinemaidens lament the loss of their gold.

ISBN 90-393-4130-3

Acknowledgements

This thesis could not have come about without the help of many individuals. Foremost I wish to thank my supervisor and copromotor Eelco Visser. He was also my master's thesis supervisor, and I was quite happy that we were able to continue to work together on the Variability/TraCE project. He shared my eventual interest in configuration management-related issues, and package management in particular. His insights have improved this research and this thesis at every point.

The Software Engineering Research Center (SERC), now known as CIBIT|SERC ICT Adviseurs, funded my PhD position. Gert Florijn initiated the variability project and the discussions with him were a valuable source of insights. The results of the present thesis are probably not what any of us had expected at the start; but then again, the nice thing about a term like "variability" is that it can take you in so many directions.

My promotor Doaitse Swierstra gave lots of good advice—but I ended up not implementing his advice to keep this thesis short. I am grateful to the members of the reading committee—Jörgen van den Berg, Sjaak Brinkkemper, Paul Klint, Alexander Wolf and Andreas Zeller—for taking the time and effort to go through this book. Karl Trygve Kalleberg, Armijn Hemel, Arthur van Dam and Martin Bravenboer gave helpful comments.

Several people have made important contributions to Nix. Notorious workaholic Martin Bravenboer in particular was an *early adopter* who contributed to almost every aspect of the system. Armijn Hemel ("I am the itch programmers like to scratch") contributed to Nixpkgs and initiated the NixOS, which will surely conquer the world. René de Groot coined the marketing slogan for NixOS—"Nix moet, alles kan!" Roy van den Broek has replaced my hacky build farm supervisor with something much nicer (and Web 2.0 compliant!). Eelco Visser, Rob Vermaas and Merijn de Jonge also contributed to Nixpkgs and the build farm. In addition, it is hard to overestimate the importance of the work done by the free and open source software community in providing a large body of non-trivial modular components suitable for validation. Thanks!

The Software Technology group is a very pleasant work environment, and I thank all my current and former colleagues for that. I especially thank my roommates Dave Clarke ("The cause of the software crisis is software"), Frank Atanassow (who demanded "pointable research"), Merijn de Jonge, Martin Bravenboer, Rob Vermaas and Stefan Holdermans. Andres Löh, Bastiaan Heeren, Arthur Baars, Karina Olmos and Alexey Rodriguez were more than just good colleagues. Daan Leijen insisted on doing things the right way. The #klaplopers provided a nice work environment in virtual space—though whether IRC is a medium that increases productivity remains an unanswered empirical question.

Arthur van Dam and Piet van Oostrum provided valuable assistance in wrestling with TEX. Atze Dijkstra, Bastiaan Heeren and Andres Löh had helpful advice (and code!) for the preparation of this thesis.

And finally I would like to thank my family—Mom, Dad, Jitske and Menno—for their support and love through all these years.

Contents

Part I.

Introduction

1. Introduction

[169] André van der Hoek, Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. Software release management. In M. Jazayeri and H. Schauer, editors, Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), pages 159–175. Springer-Verlag, 1997.

Nix ~~This thesis~~ is about getting computer programs from one machine to another—and having them still work when they get there. This is the problem of *software deployment*~~. Though it is~~ a part of the field of Software Configuration Management (SCM~~), it has not been a subject of academic study until quite recently [169]. The development of principles and tools to support the deployment process has largely been relegated to industry, system administrators, and Unix hackers. This has resulted in a large number of often *ad hoc* tools that typically automate manual practices but do not address fundamental issues in a systematic and disciplined way.~~

This is evidenced by the huge number of mailing list and forum postings about deployment failures, ranging from applications not working due to missing dependencies, to subtle malfunctions caused by incompatible components. Deployment problems also seem curiously resistant to automation: the same concrete problems appear time and again. Deployment is especially difficult in heavily component-based systems—such as Unix-based open source software—because the effort of dealing with the dependencies can increase super-linearly with each additional dependency.

~~This thesis describes a system for software deployment called *Nix* that addresses many of the problems that plague existing deployment systems. In this introductory chapter I describe the problem of software deployment, give an overview of existing systems and the limitations that motivated this research, summarise its main contributions, and outline the structure of this thesis.~~

1.1. Software deployment

Software deployment is the problem of managing the distribution of software to end-user machines. That is, a developer has created some piece of software, and this ultimately has to end up on the machines of end-users. After the initial installation of the software, it might need to be upgraded or uninstalled.

Presumably, the developer has tested the software and found it to work sufficiently well, so the challenge is to make sure that the software works just as well, i.e., the same, on the end-user machines. I will informally refer to this as *correct deployment*: given identical inputs, the software should behave the same on an end-user machine as on the developer machine[1].

This *should* be a simple problem. For instance, if the software consists of a set of files, then deployment should be a simple matter of *copying* those to the target machines. In

[1] This is all grossly oversimplified: ~~I'm making several gross simplifications, of course.~~ First, in general there is no single "developer". Second, there are usually several intermediaries between the developer and the end-user, such as a system administrator. However, for a discussion of the main issues this will suffice.

practice, deployment turns out to be much harder. This has a number of causes. These fall into two broad categories: *environment issues* and *manageability issues*.

Environment issues The first category is essentially about correctness. The software might make all sorts of demands about the *environment* in which it executes: that certain other software components are present in the system, that certain configuration files exist, that certain modifications were made to the Windows registry, and so on. If any of those environmental characteristics does not hold, then there is a possibility that the software does not work the same as it did on the developer machine. Some concrete issues are the following:

- A software component is almost never self-contained; rather, it depends on other components to do some work on its behalf. These are its *dependencies*. For correct deployment, it is necessary that all dependencies are identified. This identification is quite hard, however, as it is often difficult to *test* whether the dependency specification is complete. After all, if we forget to specify a dependency, we don't discover that fact if the machine on which we are testing already happens to have the dependency installed.

- Dependencies are not just a runtime issue. To *build* a component in the first place we need certain dependencies (such as compilers), and these need not be the same as the runtime dependencies, although there may be some overlap. In general, deployment of the build-time dependencies is not an end-user issue, but it might be in *source-based* deployment scenarios; that is, when a component is deployed in source form. This is common in the open source world.

- Dependencies also need to be *compatible* with what is expected by the referring component. In general, not all versions of a component will work. This is the case even in the presence of type-checked interfaces, since interfaces never give a full specification of the observable behaviour of a component. Also, components often exhibit build-time *variability*, meaning that they can be built with or without certain optional features, or with other parameters selected at build time. Even worse, the component might be dependent on a specific compiler, or on specific compilation options being used for its dependencies (e.g., for Application Binary Interface (ABI) compatibility).

- Even if all required dependencies are present, our component still has to *find* them, in order to actually establish a concrete composition of components. This is often a rather labour-intensive part of the deployment process. Examples include setting up the dynamic linker search path on Unix systems [160], or the CLASSPATH in the Java environment.

  [160] TIS Committee. Tool Interface Specification (TIS) Executable and Linking Format (ELF) Specification, Version 1.2. http://www.x86.org/ftp/manuals/tools/elf.pdf, May 1995.

- Components can depend on non-software artifacts, such as configuration files, user accounts, and so on. For instance, a component might keep state in a database that has to be initialised prior to its first use.

- Components can require certain hardware characteristics, such as a specific processor type or a video card. These are somewhat outside the scope of software deployment, since we can at most *check* for such properties, not *realise* them if they are missing.

- Finally, deployment can be a *distributed* problem. A component can depend on other components running on remote machines or as separate processes on the same machine. For instance, a typical multi-tier web service consists of an HTTP server, a server implementing the business logic, and a database server, possibly all running on different machines.

So we have two problems in deployment: we must *identify* what our component's requirements on the environment are, and we must somehow *realise* those requirements in the target environment. Realisation might consist of installing dependencies, creating or modifying configuration files, starting remote processes, and so on.

Manageability issues The second category is about our ability to properly manage the deployment process. There are all kinds of operations that we need to be able to perform, such as packaging, transferring, installing, upgrading, uninstalling, and answering various queries; i.e., we have to be able to support the *evolution* of a software system. All these operations require various bits of information, can be time-consuming, and if not done properly can lead to incorrect deployment. For example:

- When we uninstall a component, we have to know what steps to take to safely undo the installation, e.g., by deleting files and modifying configuration files. At the same time we must also take care never to remove any component still in use by some other part of the system.

- Likewise, when we perform a component upgrade, we should be careful not to overwrite any part of any component that might induce a failure in another part of the system. This is the well-known *DLL hell*, where upgrading or installing one application can cause a failure in another application due to shared dynamic libraries. It has been observed that software systems often suffer from the seemingly inexplicable phenomenon of "bit rot," i.e., that applications that worked initially stop working over time due to changes in the environment [26].

- Administrators often want to perform queries such as "to what component does this file belong?", "how much disk space will it take to install this component?", "from what sources was this component built?", and so on.

- Maintenance of a system means keeping the software up to date. There are many different policy choices that can be made. For instance, in a network, system administrators may want to push updates (such as security fixes) to all client machines periodically. On the other hand, if users are allowed to administer their own machines, it should be possible for them to select components individually.

- When we upgrade components, it is important to be able to *undo*, or *roll back* the effects of the upgrade, if the upgrade turns out to break important functionality. This requires both that we remember what the old configuration was, and that we have some way to reproduce the old configuration.

- In heterogeneous networks (i.e., consisting of many different types of machines), or in small environments (e.g., a home computer), it is not easy to stay up to date with software updates. In particular in the case of security fixes this is an important problem. So we need to know what software is in use, whether updates are available, and whether such updates should be performed.

- Components can often be deployed in both source and binary form. Binary packages have to be built for each supported platform, and sometimes in several variants as well. For instance, the Linux kernel has thousands of build-time configuration options. This greatly increases the deployment effort, particularly if packaging and transfer of packages is a manual or semi-automatic process.

- Since components often have a huge amount of variability, we sometimes want to expose that variability to certain users. For instance, Linux distributors or system administrators typically want to make specific feature selections. A deployment system should support this.

1.2. The state of the art

TODO: Re-evaluate these claims; it has been almost 20 years since this thesis

Having seen some of the main issues in the field of software deployment, we now look at some representative deployment tools. This section does not aim to provide a full survey of the field; rather, the goal is to show the main approaches to deployment. ~~Section 7.6 has a more complete discussion of related work.~~

Package management is a perennial problem in the Unix community [2]. In fact, entire operating system distributions rise and fall on the basis of their deployment qualities. It can be argued that Gentoo Linux's [77] quick adoption in the Linux community was entirely due to the perceived strengths of its package management system over those used by other distributions. This interest in deployment can be traced to Unix's early adoption in large, advanced and often academic installations (in contrast to the single PC, single user focus in the PC industry in a bygone era).

Also, for better or for worse, Unix systems have traditionally insisted on storing components in global namespaces in the file system such as the /usr/bin directory. This makes management tools indispensable. But more importantly, modern Unix components have fine-grained reuse, often having dozens of dependencies on other components. Since it is not desirable to use monolithic distribution (as is generally done in Windows and Mac OS X, as discussed below), a package management tool is absolutely required to support the resulting deployment complexity. Therefore Unix (and specifically, Linux) package management is what we will look at first.

TODO: Are these statements still valid?

RPM The most widely used Linux package management system is the Red Hat Package Manager (RPM) [62], and as it is a good, mature tool it is instructive to look at it in some detail. RPM is a *low-level* deployment tool: it installs and manages components, keeping meta-information about them to prevent unsafe upgrades or uninstalls.

TODO: The concepts in this section is important but not sure if one pkg manager should be singled out to represent them. But then how? Add a section for other popular package managers? I don't think I'll have time to do that...

Maybe a comparison to other tools should be skipped altogether?

```
Summary: Hello World program
Name: hello
Version: 1.0
Source0: %{name}-%{version}.tar.gz

%description
This program prints out "Hello
World".

%build
./configure --prefix=%{_prefix}
make

%install
make install

%files
/usr/bin/hello
/usr/share/man/man1/hello.1.gz
/etc/hello.conf
```

Figure 1.1.: Spec file for hello

```
Summary: Hello World GUI
Name: xhello
Version: 1.4
Source0: %name-%version.tar.gz
Requires: hello >= 1.0 [1]
Requires: XFree86

%description
This program provides a graphical
user interface around Hello.

%build
./configure --prefix=%_prefix
make

%install
make install

%files
/usr/bin/xhello
```

Figure 1.2.: Spec file for xhello

A software component is deployed by packaging it into an *RPM package* (or simply *RPM*), which is an archive file that consists of the files that constitute the component, and some meta-information about the package. This includes a specification of the dependencies of the package, scripts to be run at various points in the install and uninstall process, and administrative information such as the originator of the package. RPMs are built from *spec files*. Figure 1.1 shows a trivial spec file for an imaginary component called Hello that provides a command /usr/bin/hello.

An RPM package is typically produced from source as follows[2]:

```
$ rpmbuild -ba hello.tar.gz
```

where hello.tar.gz contains the sources of the component and its spec file. The resulting RPM (say, hello-1.0.i686.rpm) must then be transferred to the client machines in some way that the RPM tool itself does not specify. Once it is on a client machine, it can be installed:

```
$ rpm -i hello-1.0.i686.rpm
```

Similarly, we can upgrade the package when a new RPM with the same name comes along (rpm -u hello-2.0.i686.rpm), and uninstall it (rpm -e hello). To perform upgrades and uninstalls cleanly, it is necessary to track which files belong to what packages. This information can be queried by users:

```
$ rpm -ql hello
```

[2] In this thesis, the prefix $ in code samples indicates a Unix shell command.

```
/usr/bin/hello
/etc/hello.conf

$ rpm -qf /usr/bin/hello
hello-1.0
```

[145] Bruce Schneier. Applied Cryptography. John Wiley & Sons, second edition, 1996.

RPM also maintains a cryptographic hash [145] of the contents of each file as it existed at installation time. Thus, users can verify that files have not been tampered with.

RPM maintains the integrity of installed packages with respect to the dependency requirements. Figure 1.2 shows another RPM package, xhello, that provides a GUI front-end for hello and thus depends on the hello package (expressed at point $\boxed{1}$). When we have the xhello package installed, we cannot uninstall hello unless xhello is uninstalled also:

```
$ rpm -e hello
hello is needed by (installed) xhello-1.4
```

Similarly, RPM does not allow two packages that contain files with equal path names to exist simultaneously in the system. For instance, when we have our hello-1.0 package installed, it is not possible to install another package that also installs a file with path name /usr/bin/hello. In general, this means that *we cannot have multiple versions of the same component installed simultaneously*. It also means that it is hard for multiple users to independently select, install and manage software.

A fundamental problem of RPM and essentially all general package management systems is that they cannot reliably determine the correctness of dependency specifications. In our example, xhello depends on hello, and so its spec file will contain a dependency specification to that effect, e.g.,

```
Requires: hello
```

But what happens when we *forget* to specify this? When the developer builds and tests the RPM package, the component will probably work because in all likelihood the developer has hello installed. If we then deploy xhello to clients, the component will work if hello happens to have been installed previously. If it was not, xhello may fail mysteriously (Figure 1.3; black ovals denote broken components). Thus, it is intrinsically hard to validate dependency specifications. (It is also hard to prevent *unnecessary dependencies*, but that does not harm correctness, just efficiency.) An analysis of the actual number of dependency errors in a large RPM-based Linux distribution is described in [87]. The number of dependency errors turned out to be quite low, but this is likely to be at least in part due to the substantial effort invested in specifying complete dependencies. Missing dependencies lead to *incomplete deployment*; correct deployment on the other hand requires complete deployment.

Related to the inability to validate dependency specifications is the fact that dependencies tend to be *inexact*. Above, xhello required that a component named hello is present—but it makes no requirements on the actual properties or interfaces of that component. That is, the dependency specification is *nominal* (determined by name only), not *by contract* (requiring that the dependency has certain properties). So any component named hello satisfies the dependency. Actually, we can require specific versions of the dependency:

```
Requires: hello >= 1.0
```

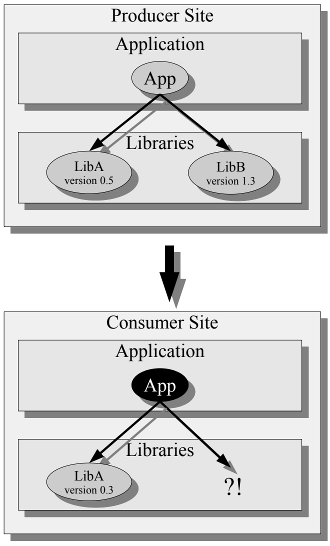


Figure 1.3.: Incomplete deployment

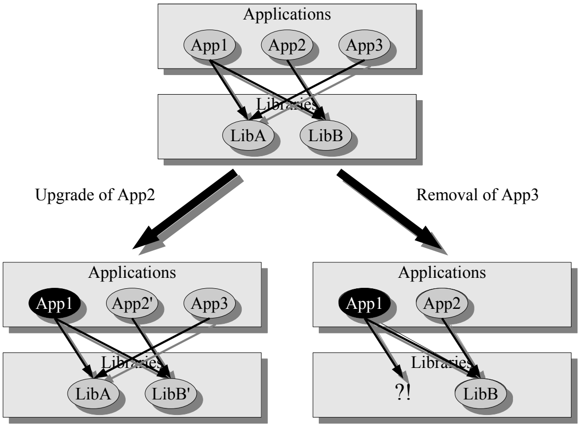


Figure 1.4.: Interference

which excludes version 1.0. However, such version specifications involve a high degree of wishful thinking, since we can never in general rely on the fact that any version in an open range works. For instance, there is no way to know whether future release 1.3.1 of hello will be backwards compatible. Even "exact" dependencies such as

```
Require: hello = 1.0
```

are unsafe, because this is still a nominal dependency: we can conceive of any number of component instances with name hello and version number 1.0 that behave completely differently. In fact, this is a real problem: Linux distributions from different vendors can easily have components with equal names (e.g., glibc-2.3.5) that actually have vendor-specific patches applied, have been built with specific options, compilers, or ABI options, and so on.

Incomplete dependencies and inexact notions of component compatibility give rise to *interference* between components, which is the phenomenon that an operation on one component can "break" an essentially unrelated component. Figure 1.4 shows two examples of interference. In the left scenario, the upgrading of application App2 breaks App1 because the new version App2' requires a newer version of LibB', which happens not to be sufficiently backwards compatible. In the right scenario, the uninstallation of App3 also breaks App1, because the package manager has removed LibA under the mistaken belief that App3 was the sole user of LibA.

A more subtle problem in RPM (and most other deployment tools) is that it has the property of *destructive upgrading*, which means that components are upgraded by overwriting the files of the old version with those of the new one. This is the case because the new ver-

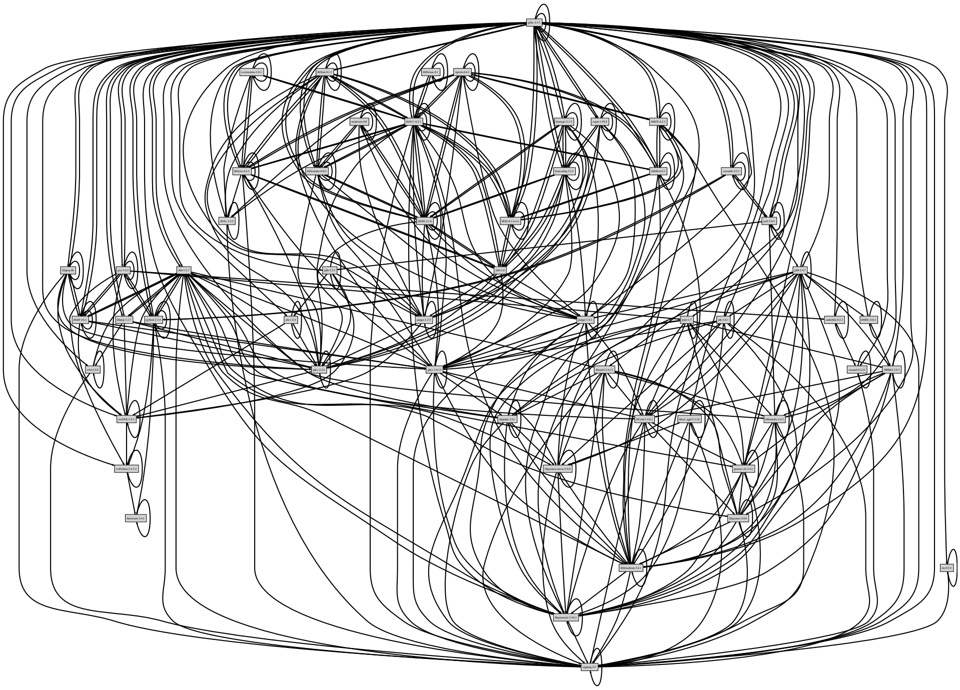


Figure 1.5.: The *dependency hell*: the runtime dependency graph of Mozilla Firefox

sion typically lives in the same paths in the file system, e.g., hello-2.0 will still install into /usr/bin/hello and /etc/hello.conf. Apart from the resulting inability to have multiple versions installed at the same time, this gives rise to a temporary inconsistency in the system: there is a time window in which we have some of the files of the old version, and some of the new version. Thus, upgrading is not *atomic*. If a user were to start the Hello program during this time window, the program might not work at all or misbehave in certain ways.

The main criticism leveled at RPM by some of its users is the difficulty in obtaining RPM packages. If we want to install a complex, multi-component system such as X.org, KDE, or GNOME, we have to manually download a possibly large number of RPM packages until all missing dependencies are resolved. That is, RPM verifies that an installation of a package proceeds safely according to its spec file, but has no way to resolve problems by itself if they do occur. This gives rise to the *dependency hell*, where users find themselves chasing down RPMs that may not be easy to obtain. Figure 1.5 shows the dependency graph of Mozilla Firefox, a popular web browser[3]. Each node is a component that must be present. This gives an intuition as to the magnitude of the problem.

However, it is not actually a problem that RPM does not obtain packages automatically: it is in fact a good separation of mechanism and policy. Higher-level deployment tools can be built on top of RPM that do provide automatic fetching of packages. Installation

[3] This picture was produced from the Firefox component in the Nix Packages collection using the nix-env --query --graph command and Graphviz [54]. TODO: Is this command still valid?

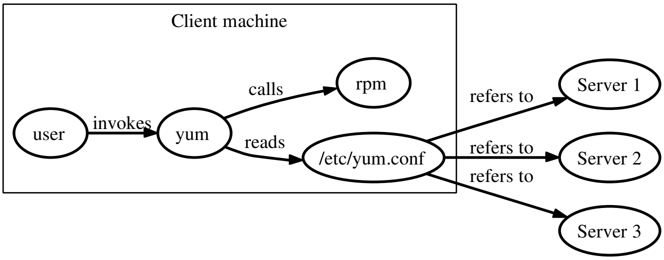


Figure 1.6.: Software deployment using yum

tools such as SuSE Linux's YaST have a global view of all available packages in a Linux distribution, and will automatically download them from a network site or prompt for the appropriate installation CD or DVD. The tool yum (short for *Yellow Dog Updater, Modified*) [174], used among others by the Fedora Core Linux distribution, is a wrapper around RPM to add support for network repositories from which the tool can automatically acquire packages. Figure 1.6 illustrates a typical yum deployment scenario. Client machines have a file /etc/yum.conf that contains a list of RPM repositories. We can then install a package *with all its dependencies* by saying, e.g.,

```
$ yum install firefox
```

Yum will consult the repositories defined in /etc/yum.conf to find the appropriate RPM packages, download them, and install them by calling the RPM tool.

Source deployment models In the open source community there are several operating system distributions that deploy packages not (primarily) in binary form but in source form. The main advantage of source deployment is that it allows greater flexibility by allowing users to make specific build time configuration choices. For instance, we can optimise components for our specific processor architecture, or remove all unneeded optional functionality in components. Whether this is actually a desirable feature for end-users is up for debate, but the ability to easily construct a customised set of component instances is quite valuable to some users, such as developers, system administrators, deployers of complex installations, and distributors.

The archetypical source deployment system is the FreeBSD Ports Collection [73]. Packages in source form (called *ports*) are essentially Makefiles [56] that describe how to recursively build the dependencies of the package, download the sources of the package, apply possible FreeBSD-specific patches, build the package, and finally install it in the file system. By passing arguments to Make (or editing the Makefile, associated files, and options in global configuration files), the component can be adapted to local requirements.

An obvious downside to source deployment is its rather considerable resource consumption. While binary deployment only requires disk space for the binaries, and possibly network resources for obtaining the binary package, source deployment requires resources to obtain and store the sources, CPU time and memory to build the package, and disk space for temporary files and the final results. To make matters worse, binary deployment only

[56] Stuart I. Feldman. Make—a program for maintaining computer programs. Software—Practice and Experience, 9(4):255–65, 1979. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.9198&rep=rep1&type=pdf

involves runtime dependencies, while source deployment may involve additional dependencies that are only used at build time, such as compilers. Unsurprisingly, FreeBSD therefore also offers a binary deployment option, called *packages*, which are pre-built ports. However, ports and packages are not quite integrated: for instance, when installing a port, dependencies that might be installed using packages are still built from source.

A further problem with source deployment is that it increases the risk of deployment failure, as now not just runtime dependencies but also build-time dependencies can affect the result of the deployment. For instance, if the user's installed C compiler is not quite what a component's developer tested against, there is a slight probability that the component fails to build or run correctly—and the product of sufficiently many slight failure probabilities is a large probability.

Windows and Mac OS Windows and Mac OS X tend to use *monolithic deployment* for applications: except for some large-grained dependencies on the operating system environment (e.g., on the kernel, the core GUI libraries, or Direct X), dependencies tend to be distributed as part of the application itself, with no sharing of dependencies between applications. This can be accomplished through static linking or by having dynamic libraries be part of the private namespace (directory tree) of the application (e.g., C:\Program Files\MyApp\Foo.DLL). While this reduces deployment complexity at the end-user machine, it has several downsides. First, it removes sharing: if two applications use the "same" component, they will nevertheless end up using private copies. The result is increased resource consumption in terms of disk space, RAM, cache efficiency, download time, and so on. Clearly, it is bad if all or many of Firefox's dependencies in Figure 1.5 were unshared. In the worst case, we get a quadratic blowup in the disk space requirements: if we have N applications that share the same M dependencies, then we need disk space $\Theta(NM)$ instead of $\Theta(N+M)$. Second, it still requires the developer to obtain and compose the components, typically through a semi-manual process.

TODO: How much has changed since then?

Especially elegant from an end-user perspective are Mac OS's *application bundles*, which are directory trees containing all files belonging to an application. Generally, such bundles are self-contained, except for operating system component dependencies. Contrary to typical Windows applications, they do not have other environment dependencies such as registry settings. This means that bundles can be copied or moved around in the file system arbitrarily. For instance, the whole of Microsoft Office X on Mac OS X can be copied between machines by dragging it from one disk to another. Again, the limitation of this approach is that it falls apart when components have dependencies on each other. That is, the bundle approach works only for "top level" components, i.e., end-user applications[4].

.NET Historically, Windows has suffered from the DLL hell, a result of an unmanaged global namespace being used to store shared dependencies, e.g., the directory C:\Windows\System. An installation or uninstallation of one application frequently caused other applications to break because a shared library (DLL) would be replaced with an incompatible version, or deleted altogether. This is a classic example of component interference.

[4] For instance, the management of Mac OS X's BSD-based Unix foundations is a complete mess, which has prompted the development of tools such as Fink [138], which is based on Debian's APT system (discussed in Section 7.6).

Microsoft's .NET platform [17] improves on this situation through its *assemblies*, which are files that store code and other resources. Assemblies can have *strong names*, which are globally unique identifiers signed by the assembly producer. Assemblies can depend on each other through these strong names. The *Global Assembly Cache* (GAC) holds a system-wide repository of assemblies, indexed by strong name. If assemblies in the GAC differ in their strong names, they will not overwrite each other. Thus, interference does not happen.

However, the GAC is only intended for shared components. Application code and miscellaneous resources are typically not stored in the GAC, but in application-specific directories, e.g., C:\Program Files\MyApp. That is, management of those components must be handled through other mechanisms. Also, the GAC only holds components that use a specific component technology—.NET assemblies.

Other systems The *Zero Install system* [112] installs components *on demand* from network repositories. On-demand installation is enabled by using a virtual file system that intercepts file system requests for component files. For instance, when a process accesses /uri/0install/abiword.org/abiword, and this file is not already present in the local component cache, it is fetched from a repository, at which point the requesting process resumes. Users can independently install software in this manner, and are unaffected by the actions of other users. Unfortunately, systems depending on non-portable extensions to the file system face difficulty in gaining wide adoption[5]. This is especially the case for deployment systems as they should support a wide variety of platforms.

1.3. Motivation TODO

From the previous discussion of existing deployment systems it should be clear that they lack important features to support safe and efficient deployment. In particular, they have some or all of the following problems:

- Dependency specifications are not validated, leading to incomplete deployment.
- Dependency specifications are inexact (e.g., nominal).
- It is not possible to deploy multiple versions or variants of a component side-by-side.
- Components can interfere with each other.
- It is not possible to roll back to previous configurations.
- Upgrade actions are not atomic.
- Applications must be monolithic, i.e., they must statically contain all their dependencies.

[5] For instance, Apple is moving away from *resource forks* to *bundles* in recent releases of its Mac OS. The former approach stores streams of metadata (such as icons) in files in addition to its regular data contents, which is not portable. The latter approach on the other hand uses directory trees of regular files to keep such data elements together, which *is* portable.

- Deployment actions can only be performed by administrators, not by unprivileged users.
- There is no link between binaries and the sources and build processes that built them.
- The system supports either source deployment or binary deployment, but not both; or it supports both but in a non-unified way.
- It is difficult to adapt components.
- Component composition is manual.
- The component framework is narrowly restricted to components written in a specific programming language or framework.
- The system depends on non-portable techniques.

The objective of the research described in this thesis is to develop a deployment system that does not have these problems.

1.4. The Nix deployment system

~~This thesis describes the *Nix deployment system*, which overcomes the limitations of contemporary deployment tools described above. I describe the concepts and implementation (*how* it works), the underlying principles (*why* it works), our experiences and empirical validation (*that* it works), and the application areas to which it can be applied (*where* it works).~~

The main idea of the Nix approach is to store software components in isolation from each other in a central component store, under path names that contain cryptographic hashes of all inputs involved in building the component, such as /nix/store/rwmfbhb2znwp...-firefox-1.0.4. As I show in this thesis, this prevents undeclared dependencies and enables support for side-by-side existence of component versions and variants.

~~**Availability** At the time of writing, the Nix system is available as free software at the homepage of the TraCE project [161].~~

1.5. Contributions

The core idea of Nix is the

~~The main contribution of this thesis is the development of a~~ *purely functional deployment model*, which we implemented in the Nix system. In this model a binary component is uniquely defined by the declared inputs used to build the component. This enables arbitrary component instances to exist side by side. ~~I show how such a model can be "retrofitted" onto existing software that was designed without such a model in mind.~~

~~Concretely, the contributions of this thesis are the following.~~

The main properties of Nix:

- The cryptographic hashing scheme used by the Nix component store prevents undeclared dependencies, giving us complete deployment. Furthermore it provides support for side-by-side existence of component versions and variants.

TODO Make sure to mention that "**Nix component store**" is the longer version of "**Nix store**" - and explain what a component is.
See *What is a Nix expression in regard to Nix package management?*
https://stackoverflow.com/questions/58243554/what-is-a-nix-expression-in-regard-to-nix-package-management
SO thread about what a "component" is and other definitions.

- Isolation between components prevents interference.

- Users can install software independently from each other, without requiring mutual trust relations. Components that are common between users are shared, i.e., stored only once.

- Upgrades are atomic; there is no time window in which the system is in an inconsistent state.

- Nix supports $O(1)$-time rollbacks to previous configurations.

- Nix supports automatic garbage collection of unused components.

- The Nix component language describes not just how to build individual components, but also *compositions*. The language is a lazy, purely functional language. This is a good basis for a component composition language, as it allows dependencies and variability to be expressed in an elegant and flexible way.

- Nix has a *transparent source/binary deployment model* that marries the best parts of source deployment models such as the FreeBSD Ports Collection, and binary deployment models such as RPM. In essence, binary deployment is an automatic optimisation of source deployment.

- Nix is *policy-free*; it provides *mechanisms* to implement various deployment policies, but does not enforce a specific one. Some policies described in this thesis are *channels* (in push and pull variants), *one-click installations*, and *pure source deployments*.

  TODO Not sure how long channels will be around. Maybe forever, but that's probably because of all the confusion. flox seems to be using them in a sane way, but look into it again.

- I show that a purely functional model supports efficient component upgrades despite the fact that a change to a fundamental component can propagate massively through the dependency graph.

  QUESTION How to generate this dependency graph?
  nix-env --query --graph ?

- Nix supports distributed multi-platform builds in a transparent manner, i.e., a remote build is indistinguishable from a local build from the perspective of the user.

- Nix provides a good basis for the implementation of a build farm, which supports continuous integration and portability testing. I describe a Nix build farm that has greatly improved manageability over other build farms and is integrated with *release management*, that is, it builds concrete installable components that can be deployed directly through Nix.

- The use of Nix for software deployment extends naturally to the field of *service deployment*. Services are running processes that provide some useful facility to users, such as web servers. They consist of software components, static configuration and data, and mutable state. The first two aspects can be managed using Nix, and its advantages—such as rollbacks and side-by-side deployment—apply here as well.

  TODO See Sander van der Burg's thesis and works on this topic.

- Though Nix is typically used to build *large-grained* components (i.e., traditional packages), it can also be used to build *small-grained* components such as individual source files. When used in this way it is a superior alternative to build managers such as Make [56], ensuring complete dependency specifications and enabling more sharing between builds.

The above may sound like a product feature list, which is not necessarily a good thing as it is always possible to add any feature to a system given enough *ad-hockery*. However, these contributions all follow from a small set of principles, foremost among them the purely functional model.

1.6. Outline of this thesis

This thesis is divided in four parts. Part I (which includes this chapter) introduces the problem domain and motivates the Nix system. Chapter 2, *"An Overview of Nix"*, introduces the Nix system from a high-level perspective (i.e., that of users or authors of Nix components), and gives some intuitions about the underlying principles.

Part II is about the principles and formal semantics of Nix. The basic philosophy behind Nix—to solve deployment problems by treating them analogously to memory management issues in programming languages—is introduced in Chapter 3, *"Deployment as Memory Management"*.

The syntax and semantics of the *Nix expression language*—a purely functional language used to describe components and compositions—is given in Chapter 4, *"The Nix Expression Language"*. This chapter also discusses some interesting aspects of the implementation of the Nix expression evaluator.

The next two chapters form the technical core of this thesis, as they describe the Nix component store and the fundamental operations on it. In fact, there are *two* models for the Nix store that are subtly different with important consequences. Chapter 5, *"The Extensional Model"*, formalises the *extensional model*, which was the first model and the one with which we have gained the most experience. This chapter describes the objects that live in the Nix store, the building of Nix expressions, binary deployment as an optimisation of source deployment, distributed and multi-platform builds, and garbage collection.

The extensional model however has several downsides, in particular the inability to securely share a Nix store between multiple users. These problems are fixed in the more recent *intensional model*, described in Chapter 6, *"The Intensional Model"*.

Part III discusses in detail the various applications of Nix. Chapter 7, *"Software Deployment"*, is about Nix's *raison d'être*, software deployment. It describes principles for Nix expression writers that help ensure correct deployment, and various *policies* that deploy components to clients. The main vehicle for this discussion is the *Nix Packages collection* (Nixpkgs), a large set of existing Unix components that we deploy through Nix. Most of our experience with the Nix system is in the context of the Nix Packages collection, so this chapter provides a certain level of empirical validation for the Nix approach.

Chapter 8, *"Continuous Integration and Release Management"*, shows that Nix provides a solid basis for the implementation of a *build farm*, which is a facility that automatically builds software components from a version management repository. This is useful to support *continuous integration testing*, which provides developers with almost immediate feedback as to whether their changes integrate cleanly with the work of other developers.

Also, a Nix build farm can produce concrete *releases* that can be automatically deployed to user machines, for instance through the channel mechanism described in Chapter 7.

Chapter 9, *"Service Deployment"*, takes Nix beyond deployment of software components and into the realm of *service deployment*. It shows that Nix extends gracefully into this domain using several examples of production services deployed using Nix.

Chapter 10, *"Build Management"*, extends Nix in another direction, towards support for low-level build management. As stated above, Nix provides exactly the right technology to implement a build manager that fixes the problems of most contemporary build tools.

Roadmap Different readers will find different parts of this thesis of interest. Chapter 2, *"An Overview of Nix"* should be read by anyone. The semantics of the Nix system in Part II can be skipped—initially, at least—by readers who are interested in Nix as a tool, but is required reading for those who wish to modify Nix itself. Nix users will be most interested in the various applications discussed in Part III. Chapter 7, *"Software Deployment"* on deployment should not be skipped as it provides an in-depth discussion on Nix expressions, builders, and deployment mechanisms and policies that are also relevant to the subsequent application chapters.

~~Nix users and developers will also find the online manual useful [161]~~.

TODO Point to the latest online docs.

Origins Parts of this thesis are adapted from earlier publications. Chapter 2, *"An Overview of Nix"* is very loosely based on the LISA '04 paper "Nix: A Safe and Policy-Free System for Software Deployment", co-authored with Merijn de Jonge and Eelco Visser [50]. Chapter 3, *"Deployment as Memory Management"* is based on Sections 3–6 of the ICSE 2004 paper "Imposing a Memory Management Discipline on Software Deployment", written with Eelco Visser and Merijn de Jonge [52]. Chapter 6, *"The Intensional Model"* is based on the ASE 2005 paper "Secure Sharing Between Untrusted Users in a Transparent Source/Binary Deployment Model" [48]. Section 7.5 on patch deployment appeared as the CBSE 2005 paper "Efficient Upgrading in a Purely Functional Component Deployment Model" [47]. Chapter 9, *"Service Deployment"* is based on the SCM-12 paper "Service Configuration Management", written with Martin Bravenboer and Eelco Visser [49]. Snippets of Section 10.2 were taken from the SCM-11 paper "Integrating Software Construction and Software Deployment" [46].

1.7. Notational conventions

Part II of this thesis contains a number of algorithms in a pseudo-code notation in an imperative style with some functional elements. Keywords in algorithms are in **boldface**, variables are in *italic*, and function names are in sans serif. Callouts like this $\boxed{1}$ are frequently used to refer to points of interest in algorithms and listings from the text. Variable assignment is written $x \leftarrow value$.

Sets are written in curly braces, e.g., $\{1,2,3\}$. Since many algorithms frequently extend variables of set type with new elements, there is a special notation $x \overset{\cup}{\leftarrow} set$ which is syntactic sugar for $x \leftarrow x \cup set$. For instance, $x \overset{\cup}{\leftarrow} \{2,3\}$ extends the set stored in x with the elements 2 and 3.

Set comprehensions are a concise notation for set construction: $\{expr \mid conditions\}$ produces a set by applying the expression *expr* to all values from a (usually implicit) universe that meet the predicate *conditions*. For example, $\{x^2 \mid x \in \{2,3,4\} \wedge x \neq 3\}$ produces the set $\{4,16\}$.

Ordered lists are written between square brackets, e.g., $[1,2,1,3]$. Lists can contain elements multiple times. Lists are sometimes manipulated in a functional (Haskell-like [135]) style using the function map and λ-abstraction: $x \leftarrow \mathsf{map}(\lambda v\,.\,f(v), y)$ constructs a list x by applying a function $f(v)$ to each element in the list x. For instance, $\mathsf{map}(\lambda x\,.\,x^2,\ [4,2,6])$ produces the list $[16,4,36]$. *Tuples* or pairs are enclosed in parentheses, e.g., (a,b,c).

Throughout this thesis, storage sizes are given in IEC units [33]: 1 KiB = 1024 bytes, 1 MiB = 1024^2 bytes, and 1 GiB = 1024^3 bytes.

Data types are defined in a Haskell-like notation. For instance, the definition

```
data Foo = Foo {
    x : String,
    ys : [String],
    zs : {String}
}
```

defines a data type Foo with three fields: a string x, a list of strings ys, and a set of strings zs. An example of the construction of a value of this type is as follows:

```
Foo {x = "test", ys = ["hello", "world"], zs = ∅}
```

In addition, types can have unnamed fields and multiple constructors. For example, the definition

```
data Tree = Leaf String | Branch Tree Tree
```

defines a binary tree type with strings stored in the leaves. An example value is

```
Branch (Leaf "a") (Branch (Leaf "b") (Leaf "c")).
```

2. An Overview of Nix

~~The previous chapter discussed the features that we require from a software deployment system, as well as the shortcomings of current technology. The subject of this thesis is the Nix deployment system, which addresses many of these problems. This chapter gives a gentle introduction to Nix from a user perspective (where "user" refers to component deployers as well as end-users). The succeeding parts of this thesis will present a more formal exposition of Nix's semantics (Part II) and applications (Part III).~~

TODO/NOTES - COMPONENT
https://stackoverflow.com/questions/58243554/what-is-a-nix-expression-in-regard-to-nix-pac
kage-managemen
https://discourse.nixos.org/t/what-are-interesting-unique-and-or-non-standard-uses-of-nix/12
977

2.1. The Nix store

Nix is a system for software deployment. The term *component* will be used to denote the basic units of deployment. These are simply sets of files that implement some arbitrary functionality through an interface. In fact, Nix does not really care what a component actually is. As far as Nix is concerned a component is just a set of files in a file system. That is, we have a very weak notion of *component*, weaker even than the commonly used definition in [155]. (What we call a component typically corresponds to the ambiguous notion of a *package* in package management systems. Nix's notion of components is discussed further in Section 3.1.)

Nix stores components in a *component store*, also called the *Nix store*. The store is simply a designated directory in the file system, usually /nix/store. The entries in that directory are components (and some other auxiliary files discussed later). Each component is stored in *isolation*; no two components have the same file name in the store.

A subset of a typical Nix store is shown in Figure 2.1. It contains a number of applications—GNU Hello 2.1.1 (a toy program that prints "Hello World", Subversion 1.1.4 (a version management system), and Subversion 1.2.0—along with some of their dependencies. These components are not single files, but directory trees. For instance, Subversion consists of a directory called bin containing a program called svn, and a directory lib containing many more files belonging to the component. (For simplicity, many files and dependencies have been omitted from the example.) The arrows denote runtime dependencies between components, which will be described shortly.

The notable feature in Figure 2.1 is the *names* of the components in the Nix store, such as /nix/store/bwacc7a5c5n3...-hello-2.1.1. The initial part of the file names, e.g., bwacc7a5c5n3..., is what gives Nix the ability to prevent undeclared dependencies and component interference. It is a representation of the *cryptographic hash* of *all inputs involved in building the component*.

Cryptographic hash functions are hash functions that map an arbitrary-length input onto a fixed-length bit string. They have the property that they are collision-resistant: it is computationally infeasible to find two inputs that hash to the same value. Cryptographic hashes are discussed in more detail in Section 5.1. Nix uses 160-bit hashes represented in

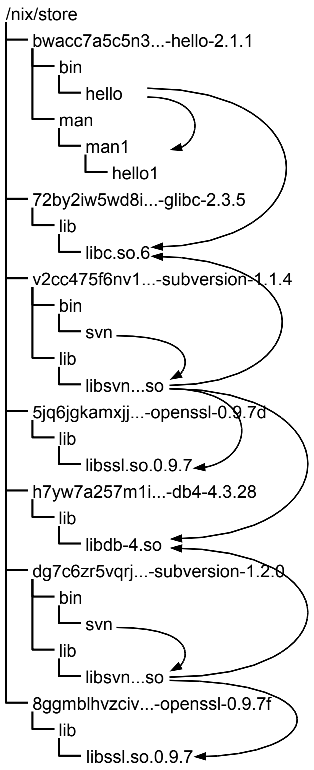

/nix/store
bwacc7a5c5n3...-hello-2.1.1
bin
hello
man
man1
hello1
72by2iw5wd8i...-glibc-2.3.5
lib
libc.so.6
v2cc475f6nv1...-subversion-1.1.4
bin
svn
lib
libsvn...so
5jq6jgkamxjj...-openssl-0.9.7d
lib
libssl.so.0.9.7
h7yw7a257m1i...-db4-4.3.28
lib
libdb-4.so
dg7c6zr5vqrj...-subversion-1.2.0
bin
svn
lib
libsvn...so
8ggmblhvzciv...-openssl-0.9.7f
lib
libssl.so.0.9.7

Figure 2.1.: The Nix store

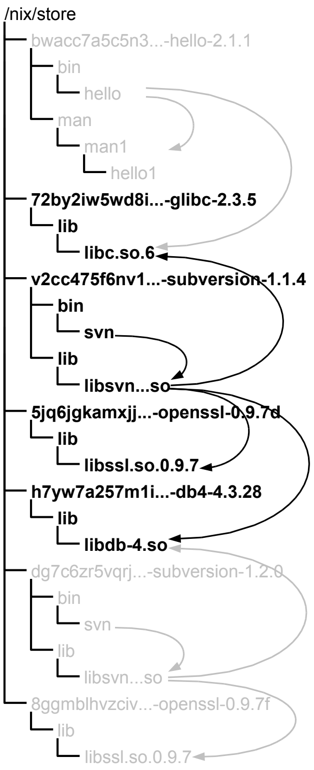

/nix/store
bwacc7a5c5n3...-hello-2.1.1
bin
hello
man
man1
hello1
72by2iw5wd8i...-glibc-2.3.5
lib
libc.so.6
v2cc475f6nv1...-subversion-1.1.4
bin
svn
lib
libsvn...so
5jq6jgkamxjj...-openssl-0.9.7d
lib
libssl.so.0.9.7
h7yw7a257m1i...-db4-4.3.28
lib
libdb-4.so
dg7c6zr5vqrj...-subversion-1.2.0
bin
svn
lib
libsvn...so
8ggmblhvzciv...-openssl-0.9.7f
lib
libssl.so.0.9.7

Figure 2.2.: Closure in the store

a base-32 notation, so each hash is 32 characters long. In this thesis, hashes are abridged to ellipses (...) most of the time for reasons of legibility. The full path of a directory entry in the Nix store is referred to as its *store path*. An example of a full store path is:

```
/nix/store/bwacc7a5c5n3qx37nz5drwcgd2lv89w6-hello-2.1.1
```

So the file bin/hello in that component has the full path

```
/nix/store/bwacc7a5c5n3qx37nz5drwcgd2lv89w6-hello-2.1.1/bin/hello
```

The hash is computed over all inputs, including the following:

- The sources of the components.
- The script that performed the build.
- Any arguments or environment variables [152] passed to the build script.
- All build time dependencies, typically including the compiler, linker, any libraries used at build time, standard Unix tools such as cp and tar, the shell, and so on.

TODO
How is the hash computed? Have there been major changes since the thesis?

Cryptographic hashes in store paths serve two main goals. They prevent interference between components, and they allow complete identification of dependencies. The lack

[152] W. Richard Stevens and Stephen A. Rago. Advanced Programming in the UNIX Environment. AddisonWesley, second edition, June 2005.

of these two properties is the cause for the vast majority of correctness and flexibility problems faced by conventional deployment systems, as we saw in Section 1.2.

Preventing interference Since the hash is computed over all inputs to the build process of the component, any change, no matter how slight, will be reflected in the hash. This includes changes to the dependencies; the hash is computed recursively. Thus, the hash essentially provides a unique identifier for a configuration. If two component compositions differ in any way, they will occupy different paths in the store (except for dependencies that they have in common). Installation or uninstallation of a configuration therefore will not interfere with any other configuration.

For instance, in the Nix store in Figure 2.1 there are two versions of Subversion. They were built from different sources, and so their hashes differ. Additionally, Subversion 1.2.0 uses a newer version of the OpenSSL cryptographic library. This newer version of OpenSSL likewise exists in a path different from the old OpenSSL. Thus, even though installing a new Subversion entails installing a new OpenSSL, the old Subversion instance is not affected, since it continues to use the old OpenSSL.

Recursive hash computation causes changes to a component to "propagate" through the dependency graph. This is illustrated in Figure 2.3, which shows the hash components of the store paths computed for the Mozilla Firefox component (a web browser) and some of its build time dependencies, both before and after a change is made to one of those dependencies. (An edge from a node *A* to a node *B* denotes that the build result of *A* is an input to the build process of *B*.) Specifically, the GTK GUI library dependency is updated from version 2.2.4 to 2.4.13. That is, its source bundle (gtk+-2.2.4.tar.bz2 and gtk+-2.4.13.tar.bz2, respectively) changes. This change propagates through the dependency graph, causing different store paths to be computed for the GTK component *and* the Firefox component. However, components that do not depend on GTK, such as Glibc (the GNU C Library), are unaffected.

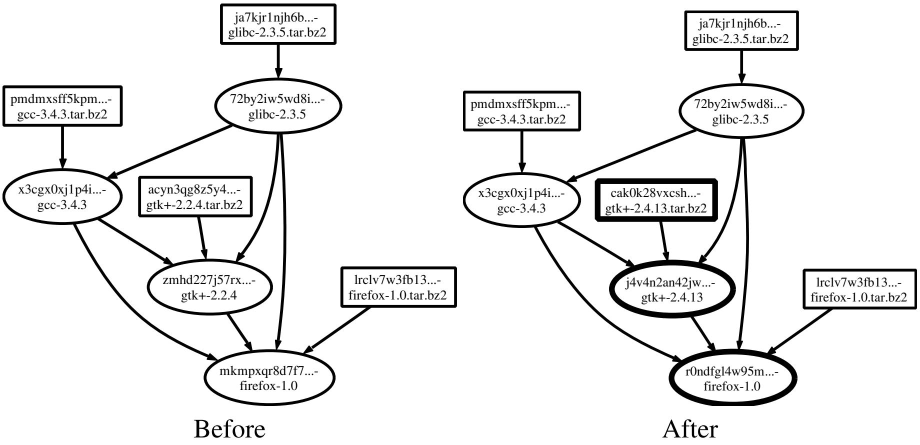
An important point here is that upgrading *only* happens by rebuilding the component in question and all components that depend on it. We never perform a destructive upgrade. Components never change after they have been built—they are marked as read-only in the file system. Assuming that the build process for a component is deterministic, this means that the hash identifies the contents of the components at all times, not only just after it has been built. Conversely, the build-time inputs determine the contents of the component.

That's a big "if", and I seem to remember some issues about this.

Therefore we call this a *purely functional model*. In purely functional programming languages such as Haskell [135], as in mathematics, the result of a function call depends exclusively on the definition of the function and on the arguments. In Nix, the contents of a component depend exclusively on the build inputs. The advantage of a purely functional model is that we obtain strong guarantees about components, such as non-interference.

Identifying dependencies The use of cryptographic hashes in component file names also prevents the use of undeclared dependencies, which (as we saw in Section 1.2) is the major cause of incomplete deployment. The reason that incomplete dependency information occurs is that there is generally nothing that prevents a component from accessing another component, either at build time or at runtime. For instance, a line in a build script or Makefile such as

QUESTION
Because the hash changes if a dep is add/removed? Or what else am I'm missing here?



Before

After

Figure 2.3.: Propagation of dependency changes through the store paths of the build-time component dependency graph

```
gcc -o program main.c ui.c -lssl
```

which links a program consisting of two C modules against a library named ssl, causes the linker to search in a set of standard locations for a library called ssl[1]. These standard locations typically include "global" system directories such as /usr/lib on Unix systems. If the library happens to exist in one of those directories, we have incurred a dependency. However, there is nothing that forces us to include this dependency in the dependency specifications of the deployment system (e.g., in the RPM spec file of Figure 1.2).

At runtime we have the same problem. Since components can perform arbitrary I/O actions, they can load and use other components. For instance, if the library ssl used above is a dynamic library, then program will contain an entry in its dynamic linkage meta-information that causes the dynamic linker to search for the library when program is started. The dynamic linker similarly searches in global locations such as /lib and /usr/lib.

Of course, there are countless mechanisms other than static or dynamic linking that establish a component composition. Some examples are including C header files, importing Java classes at compile time, calling external programs found through the PATH environment variable, and loading help files at runtime.

The hashing scheme neatly prevents these problems by not storing components in global locations, but in isolation from each other. For instance, assuming that all components in the system are stored in the Nix store, the linker line

```
gcc -o program main.c ui.c -lssl
```

will simply *fail* to find libssl. Unless the path to an OpenSSL instance (e.g., /nix/store/-5jq6jgkamxjj...-openssl-0.9.7d) was explicitly fed into the build process and added to the linker's search path, no such instance will be found by the linker.

[1] To be precise, libssl.a or libssl.so on Unix.

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  40 bb 04 08 34 00 00 00  |........@...4...|
...
00000130  04 00 00 00 2f 6e 69 78  2f 73 74 6f 72 65 2f 37  |..../nix/store/7|
00000140  32 62 79 32 69 77 35 77  64 38 69 68 72 35 79 37  |2by2iw5wd8ihr5y7|
00000150  6e 31 31 31 6a 66 77 6c  37 33 71 32 68 36 37 2d  |n111jfwl73q2h67-|
00000160  67 6c 69 62 63 2d 32 2e  33 2e 35 2f 6c 69 62 2f  |glibc-2.3.5/lib/|
00000170  6c 64 2d 6c 69 6e 75 78  2e 73 6f 2e 32 00 00 00  |ld-linux.so.2...|
...
00002670  73 74 6f 72 65 2f 35 6a  71 36 6a 67 6b 61 6d 78  |store/5jq6jgkamx|
00002680  6a 6a 64 64 6c 61 76 67  76 63 39 6e 76 30 6c 72  |jjddlavgvc9nv0lr|
00002690  6d 38 66 79 73 37 2d 6f  70 65 6e 73 73 6c 2d 30  |m8fys7-openssl-0|
000026a0  2e 39 2e 37 64 2f 6c 69  62 3a 2f 6e 69 78 2f 73  |.9.7d/lib:/nix/s|
...
```

Figure 2.4.: Retained dependencies in the svn executable

Also, we go to some lengths to ensure that component builders are *pure*, that is, not influenced by outside factors. For example, the builder is called with an empty set of environment variables (such as the PATH environment variable, which is used by Unix shells to locate programs) to prevent user settings such as search paths from reaching tools invoked by the builder. Similarly, at runtime on Linux systems, we use a patched dynamic linker that does not search in any default locations—so if a dynamic library is not explicitly declared with its full path in an executable, the dynamic linker will not find it.

Thus, when the developer or deployer fails to specify a dependency explicitly (in the Nix expression formalism, discussed below), the component will fail *deterministically*. That is, it will not succeed if the dependency already happens to be available in the Nix store, without having been specified as an input. By contrast, the deployment systems discussed in Section 1.2 allow components to build or run successfully even if some dependencies are not declared.

Retained dependencies A rather tricky aspect to dependency identification is the occurrence of *retained dependencies*. This is what happens when the build process of a component stores a path to a dependency in the component. For instance, the linker invocation above will store the path of the OpenSSL library in program, i.e., program will have a "hard-coded" reference to /nix/store/5jq6jgkamxjj...-openssl-0.9.7d/lib/libssl.so.

Figure 2.4 shows a dump of parts of the Subversion executable stored in the file /nix/store/v2cc475f6nv1...-subversion-1.1.4/bin/svn. Offsets are on the left, a hexadecimal representation in the middle, and an ASCII representation on the right. The build process for Subversion has caused a reference to the aforementioned OpenSSL instance to be stored in the program's executable image. The path of OpenSSL has been stored in the RPATH field of the header of the ELF executable, which specifies a list of directories to be searched by the dynamic linker at runtime [160]. Even though our patched dynamic linker does not search in /nix/store/5jq6jgkamxjj...-openssl-0.9.7d/lib by default, it will find the library anyway through the executable's RPATH.

TODO
Understand this

This might appear to be bad news for our attempts to prevent undeclared dependencies. Of course, we happen to know the internal structure of Unix executables, so for this specific file format we can discover retained dependencies. But we do not know the format of *every* file type, and we do not wish to commit ourselves to a single component composition mechanism. E.g., Java and .NET can find retained dependencies by looking at class files and assemblies, respectively, but only for their specific dynamic linkage mechanisms (and not for dependencies loaded at runtime).

The hashing scheme comes to the rescue once more. The hash part of component paths is highly distinctive, e.g., 5jq6jgkamxjj.... Therefore we can discover retained dependencies *generically*, independent of specific file formats, by *scanning* for occurrences of hash parts. For instance, the executable image in Figure 2.4 contains the highlighted string 5jq6jgkamxjj..., which is evidence that an execution of the svn program might need that particular OpenSSL instance. Likewise, we can see that it has a retained dependency on some Glibc instance (/nix/store/72by2iw5wd8i.... Thus, we automatically add these as runtime dependencies of the Subversion component. Using this scanning approach, we find the runtime dependencies indicated in Figure 2.1.

This approach might seem a bit dodgy. After all, what happens when a file name is represented in a non-standard way, e.g., in UTF-16 [34, Section 2.5] or when the executable is compressed? In Chapter 3 the dependency problem is cast in terms of memory management in programming languages, which justifies the scanning approach as an analogue of *conservative garbage collection*. Whether this is a legitimate approach is an empirical question, which is addressed in Section 7.1.5.

QUESTIONS
This scanning stuff is still fuzzy. Why is it important? What needs to be done when such a path is found? Doesn't this mean that the path of the dep already references a Nix store path - so why is that bad if we want to keep everything in the Nix store?

Note that strictly speaking it is not the use of cryptographic hashes per se, but globally unique identifiers in file names that make this work. A sufficiently long pseudo-random number also does the trick. However, the hashes are needed to prevent interference, while at the same time preventing unnecessary duplication of identical components (which *would* happen with random paths).

Closures Section 1.2 first stated the goal of *complete deployment*: safe deployment requires that there are no missing dependencies. This means that we need to deploy *closures* of components under the “depends-on” relation. That is, when we deploy (i.e., copy) a component X to a client machine, and X depends on Y, then we also need to deploy Y to the client machine.

The hash scanning approach gives us all runtime dependencies of a component, while hashes themselves prevent undeclared build-time dependencies. Furthermore, these dependencies are *exact*, not *nominal* (see page 8). Thus, *Nix knows the entire dependency graph*, both at build time and runtime. With full knowledge of the dependency graph, Nix can compute closures of components. Figure 2.2 shows the closure of the Subversion 1.1.4 instance in the Nix store, found by transitively following all dependency arrows.

TODO
Come up with a definition of **closure**. (And first, figure out what it really is; e.g., all deps and transitive deps both build- and run-time?)

See also Figure 2.2 above!

The formal definition of "closure" is in 3.3 Closures section.

In summary, the purely functional model, and its concrete implementation in the form of the hashing approach used by the Nix store, prevents interference and enables complete deployment. It makes deployment much more likely to be correct, and is therefore one of the major results of this thesis. However, the purely functional model does provoke a number of questions, such as:

- Hashes do not appear to be very user-friendly. Can we hide them from users in everyday interaction?
- Can we deploy upgrades efficiently? That is, suppose that we want to upgrade Glibc

(a dependency of all other components in Figure 2.1). Can we prevent a costly redownload of all dependent components?

As we will see in the remainder of this thesis, the answer to these questions is “yes”.

2.2. Nix expressions

SO: What are Nix expressions?

TODO
Follow along this with no (or minimal) deps on Nixpkgs. The best would be to create a "tutorial Nixpkgs" for educational purposes. One reason is that `all-packages.nix` calls `hello/default.nix` using `callPackage` and that is not very educational...

Nix components are built from *Nix expressions*. The language of Nix expressions is a simple purely functional language used to describe components and the compositions thereof.

This section introduces the Nix expression language using a simple example. In computer science’s finest tradition we will start with “Hello World”; to be precise, a Nix expression that builds the GNU Hello package. GNU Hello is a program that prints out the text Hello World and so “allows nonprogrammers to use a classic computer science tool which would otherwise be unavailable to them” [67]. It is representative of what one must do to deploy a component using Nix. Generally, to deploy a component one performs the following three steps:

1. function

- Write a Nix expression for the component (Figure 2.6), which describes all the inputs involved in building the component, such as dependencies (other components required by the component), sources, and so on.

2.

- Write a *builder* (Figure 2.7)—typically a shell script—that actually builds the component from the inputs.

QUESTION
How does the builder come in to the picture (i.e., between the function and its invocation)?

3. function invocation

- Create a composition (Figure 2.8). The Nix expression written in the first step is a *function*: in order to build a concrete component, it requires that the dependencies are filled in. To compose the component with its dependencies, we must write another Nix expression that calls the function with appropriate arguments.

The Nix Packages collection The GNU Hello example is taken from the Nix Packages collection (Nixpkgs), a large set of Nix expressions for common and not-so-common software components. Since Nixpkgs contains many components and also describes their compositions, many files are involved in building a component. It is useful to have a mental picture of how the different parts fit together. Figure 2.5 shows the directory structure of parts of the Nix Packages collection, including some of the Nix expressions, builders, and miscellaneous inputs contained therein. The function that builds Hello components is defined in pkgs/applications/misc/hello/default.nix. Likewise, many other components are defined, hierarchically organised by component purpose or topic. Builders are placed in the same directory as the Nix expression that uses them, e.g., pkgs/applications/misc/hello/builder.sh for the Hello build script.

NOTE/TODO
Yeah, there is a default builder that is automatically invoked and not sure what the convention is for custom builders. Look up the docs (but which one?)

Compositions are defined in ~~pkgs/system/all-packages-generic.nix~~ pkgs/top-level/all-packages.nix, which imports the various component-specific Nix expressions and puts them all together. The file has generic in its name because it is itself a function that returns compositions for various machine architecture and operating system platforms. The function pkgs/system/all-packages.nix does actually evaluate to a set of concrete components that can be built and installed on the current platform.

TODO
The squiggly underlined parts are kind of true; there is a modular composition of Nix expressions but no clue how the logic behind it has changed.

```
nixpkgs
└── pkgs
    ├── applications
    │   ├── misc
    │   │   └── hello
    │   │       ├── default.nix
    │   │       └── builder.sh
    │   └── networking
    │       └── browsers
    │           └── firefox
    │               ├── default.nix
    │               ├── builder.sh
    │               └── writable-copies.patch
    ├── development
    │   ├── libraries
    │   │   └── glibc
    │   │       ├── default.nix
    │   │       └── builder.sh
    │   ├── compilers
    │   │   └── gcc
    │   │       └── ...
    │   └── interpreters
    │       └── perl
    │           └── ...
    └── system
        ├── all-packages-generic.nix
        ├── all-packages.nix
        └── stdenvs.nix
```

Figure 2.5.: Organisation of the Nix Packages collection

QUESTION
How much has changed?

There is at least one re-write that I remember being talked about but wasn't paying attention at that time...

E.g., pkgs/applications/misc/hello doesn't have `builder.sh` anymore as it uses the default one (from where? can't remember) This is true for the rest of the packages as well.

There is nothing in Nix that requires this organisation; it is merely a useful convention. It is perfectly possible to place component expressions and builders in the same directory by naming them appropriately, e.g., hello.nix, hello-builder.sh, firefox.nix, and so on. It is also possible to put all Nix expressions in a single file, e.g., put everything in all-packages-generic.nix, which of course would make this file quickly spiral out of control. In fact, it is not even necessary to define components as functions and compose them separately, as done in the 3-step procedure above; we could remove all variability and write Nix expressions that describe exactly one build action, rather than a family of build actions. Clearly, that is not a very general approach.

We now look at the parts that constitute the Hello component in detail.

Nix expression for the Hello component Figure 2.6 shows a Nix expression for GNU Hello. It has a number of salient features:

2 This states that the expression is a *function* that expects to be called with three arguments: stdenv, fetchurl, and perl. They are needed to build Hello, but we don't know how to build them here; that's why they are function arguments. stdenv is a component that is used by almost all Nix Packages components; it provides a "standard" environment consisting of the things one expects in a basic Unix environment: a C/C++ compiler (GCC, to be precise), the Bash shell, fundamental Unix tools

NOTE
The situation is a bit different as of 2/7/2022:
https://github.com/NixOS/nixpkgs/blob/6958276347071cf1092c0db4b2c503eb5acb889f/pkgs/applications/misc/hello/default.nix

Build-time deps?

```
{stdenv, fetchurl, perl}: 2

stdenv.mkDerivation { 3
  name = "hello-2.1.1"; 4
  builder = ./builder.sh; 5
  src = fetchurl { 6
    url = http://ftp.gnu.org/pub/gnu/hello/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  inherit perl; 7
}
```

NOTE: This has already been deprecated and replaced with "pname" and "version". See callout 4 below with a link to discourse.

Figure 2.6.: pkgs/applications/misc/hello/default.nix: Nix expression for GNU Hello

such as cp, grep, tar, etc. fetchurl is a function that downloads files. perl is the Perl interpreter.

Nix functions generally have the form {*x*, *y*, ..., *z*}: *e* where *x*, *y*, etc. are the names of the expected arguments, and where *e* is the body (result) of the function. So here, the entire remainder of the file is the body of the function; when given the required arguments, the body describes how to build an instance of the Hello component.

3 The result of the function in this case is a *derivation*. This is Nix-speak for a component build action, which *derives* the component from its inputs. We perform a derivation by calling stdenv.mkDerivation. mkDerivation is a function provided by stdenv that builds a component from a set of *attributes*[2]. An attribute set is just a list of key/value pairs where each value is an arbitrary Nix expression. They take the general form {name$_1$ = expr$_1$; ... name$_n$ = expr$_n$;}.

See "What is a store derivation?

The attributes given to stdenv.mkDerivation are the concrete inputs to the build action.

4 A few of the attributes to derivations are special. The attribute name specifies the symbolic name and version of the component. It is appended to the cryptographic hash in store paths (see, e.g., Figure 2.1). Nix doesn't care very much about it most of the time.

NOTE: This has already been deprecated and replaced with "pname" and "version".
(Can't remember the nixpkgs issue where this pname+version scheme was introduced...)
Until then, here's this thread: https://discourse.nixos.org/t/brainstorming-for-rfc-pname-and-version/12873

5 The attribute builder specifies the script that actually builds the component. This attribute can sometimes be omitted, in which case stdenv.mkDerivation will fill in a default builder (which essentially performs a "standard" Unix component build sequence consisting of the commands configure; make; make install). Hello is sufficiently simple that the default builder suffices; but in this case, for educational purposes an actual builder is shown in Figure 2.7, discussed below.

TODO
Find the default builder in Nixpkgs.

6 The builder has to know what the sources of the component are. Here, the attribute src is bound to the result of a call to the fetchurl function. Given a URL and an MD5 hash of the expected contents of the file at that URL, this function builds a derivation

[2] mkDerivation is actually a wrapper around the primitive operation derivation, shown on page 80.

```
source $stdenv/setup 8

PATH=$perl/bin:$PATH 9

tar xvfz $src 10
cd hello-*
./configure --prefix=$out 11
make 12
make install
```

Figure 2.7.: pkgs/applications/misc/hello/builder.sh: Builder for GNU Hello

that downloads the file and checks its hash. So the sources are dependencies that like all other dependencies are built before Hello itself is built.

Instead of src any other name could have been used, and in fact there can be any number of sources (bound to different attributes). However, src is customary, and it is also expected by the default builder (which we don't use in this example).

7 Since the derivation requires Perl, we have to pass the value of the perl function argument to the builder. All attributes in the set are actually passed as environment variables to the builder, so declaring an attribute

QUESTION
Based on Figure 2.7, perl (and other pkg-related attributes) provides the path to the perl package/component in the Nix store (that's why the $perl/bin has been added to the PATh to get the executable)

```
    perl = perl; === inherit perl;
```

will do the trick: it binds an attribute perl to the function argument which also happens to be called perl. However, this looks a bit silly, so there is a shorter syntax. The inherit keyword causes the specified attributes to be bound to whatever variables with the same name happen to be in scope.

Builder for the Hello component Figure 2.7 shows the builder for GNU Hello. To build a component, Nix executes the builder. The attributes in the attribute set of the derivation are passed as environment variables. Attribute values that evaluate to derivations (such as perl, which below in Figure 2.8 is bound to a derivation) are built recursively. The paths of the resulting components are passed through the corresponding environment variables. The special environment variable out holds the intended output path. This is where the builder should store its result.

Ok, so to continue from the question above, `perl` evaluates to a derivation (see 2.8 below) but in the builder itself it is used as if it was built already (because, in effect, at that point it already is); so by the time it gets to hello's builder, it points to the perl package's path in the Nix store. (It says this right there in the next sentence...)

The build script for Hello is a typical Unix build script: it unpacks the sources, runs a configure script that detects platform characteristics and generates Makefiles, runs make to compile the program, and finally runs make install to copy it to its intended location in the file system. However, there are some Nix-specific points:

8 When Nix runs a builder, it initially completely clears all environment variables (except for the attributes declared in the derivation). For example, the PATH variable is empty[3]. This is done to prevent undeclared inputs from being used in the build process. If for example the PATH contained /usr/bin, then the builder might accidentally use /usr/bin/gcc.

[3] Actually, it is initialised to /path-not-set to prevent Bash from setting it to a default value.

The first step is therefore to set up the environment so that the tools included in the standard environment are available. This is done by calling the setup script of the standard environment. The environment variable stdenv points to the location of the standard environment being used. (It wasn't specified explicitly as an attribute in the Hello expression in Figure 2.6, but mkDerivation adds it automatically.)

9 Since Hello needs Perl, we have to make sure that Perl is in the PATH. The perl environment variable points to the location of the Perl component (since it was passed in as an attribute to the derivation), so the shell expression $perl/bin evaluates to the directory containing the Perl interpreter.

10 Now we have to unpack the sources. The src attribute was bound to the result of fetching the Hello source distribution from the network, so the src environment variable points to the location in the Nix store to which the distribution was downloaded. After unpacking, we change to the resulting source directory.

QUESTION Is this downloaded and extracted dir in the Nix store?

11 GNU Hello is a typical Autoconf-based package. Autoconf [64, 172] is a tool that allows a source-based package to automatically adapt itself to its environment by detecting properties such as availability and paths of libraries and other dependencies, quirks in the C compiler, and so on. Autoconf-based packages always have a script called configure that performs these tests and generates files such as Makefiles, C header files (e.g., config.h), etc. Thus, we first have to run Hello's configure script (which was generated by Autoconf). The store path where the component is to be stored, is passed in through the environment variable out. Here we tell configure to produce Makefiles that will build and install a component in the intended output path in the Nix store.

12 Finally we build Hello (make) and install it into the location specified by the out variable (make install).

Composition of the Hello component Since the Nix expression for Hello in Figure 2.6 is a function, we cannot use it directly to build and install a Hello component. We need to *compose* it first; that is, we have to call the function with the expected arguments. In the Nix Packages collection this is done in the file pkgs/system/all-packages-generic.nix, where all Nix expressions for components are imported and called with the appropriate arguments. This is a large file, but the parts relevant to Hello are shown in Figure 2.8.

NOTE It's called by a different name today and it's composition is different too. Also, it's humongous.

13 This file defines a set of attributes, all of which in this case evaluate to concrete derivations (i.e., not functions). In fact, we define a *mutually recursive* set of attributes using the keyword rec. That is, the attributes can refer to each other[4]. This is precisely what we want, namely, "plug" the various components into each other.

14 Here we *import* the Nix expression for GNU Hello. The import operation just loads and returns the specified Nix expression. In fact, we could just have put the contents

[4] While attributes can refer to values defined in the same scope, derivations cannot be mutual dependencies: such derivations would be impossible to build (see also Lemma 3 on page 130).

```
...

rec { 13

  hello = (import ../applications/misc/hello 14 { 15
    inherit fetchurl stdenv perl;
  };

  perl = (import ../development/interpreters/perl) { 16
    inherit fetchurl stdenv;
  };

  fetchurl = (import ../build-support/fetchurl) {
    inherit stdenv; ...
  };

  stdenv = ...;
}
```

Figure 2.8.: pkgs/system/all-packages-generic.nix: Composition of GNU Hello and other components

of Figure 2.6 in all-packages-generic.nix at this point; this is completely equivalent, but it would make the file rather bulky.

Relative paths such as ../applications/misc/hello are relative to the directory of the Nix expression that contains them, rather than the current directory or some global search path. Since the expression in Figure 2.8 is stored in pkgs/system/all-packages-generic.nix, this relative path resolves to pkgs/applications/misc/hello, which is the directory of the Nix expression in Figure 2.6.

Note that we refer to ../applications/misc/hello, not ../applications/misc/hello/default.nix. The latter is actually equivalent to the former in the case of directories; when importing a path referring to a directory, Nix automatically appends /default.nix to the path.

15 This is where the actual composition takes place. Here we *call* the function imported from ../applications/misc/hello with an attribute set containing the arguments that the function expects, namely fetchurl, stdenv, and perl. We use the inherit construct again to copy the attributes defined in the surrounding scope (we could also have written fetchurl = fetchurl;, etc.). Note that Nix function arguments are not positional; Nix functions just expect an attribute set containing attributes corresponding to the declared (formal) arguments.

The result of this function call is an actual derivation that can be built by Nix (since when we fill in the arguments of the function, what we get is its body, which is the call to stdenv.mkDerivation in Figure 2.6).

TODO Make sure to define in the new guide what a **derivation** is and use it consistently according to its meaning.

For example, a Nix expression that evaluation to a derivation is NOT a derivation - is a function that evaluates to a derivation.

NOTE That is, until the function parameter **is** an attribute set.

Just realized though that this is a mute point as all Nix functions only accept a single parameter:)

16 Likewise, concrete instances of Perl, fetchurl, and the standard environment are con-

TODO ... and make sure that all the definitions are linked to everywhere they are used.

QUESTION How to do this in a way that this doesn't become intrusive? Maybe links to definitions should be marked up with a different style than regural links.

NOTE Currently, "fetchurl" is defined in
/pkgs/build-support/fetchurl/default.nix

structed by calling their functions with the appropriate arguments.

The attribute fetchurl actually does *not* evaluate to a derivation, but to a function that takes arguments url and md5, as seen in Figure 2.6. This is an example of *partial parameterisation*.

Thus, the value bound to the hello attribute in Figure 2.8 represents a derivation of an instance of Hello, as well as of all its dependencies.

Surprisingly, describing not just components but also compositions is not a common feature of the component description formalisms of deployment systems. For instance, RPM spec files (see page 6) describe how to build components, but not how to build the dependencies—even though the dependencies can have a profound effect on the build result. The actual composition is left implicit. Nominal dependency specifications are used to require the presence of certain dependencies, but as we have seen, nominal dependency specifications are insufficient. Formalisms such as RPM spec files therefore woefully *underspecify* components. Only a full set of RPM spec files closed under the dependency relation fully describes a component build (disregarding, of course, the real possibility of undeclared dependencies and other environmental influences).

A slightly more complex example Figure 2.9 shows a Nix expression that demonstrates some additional language features. It contains a function that produces *Subversion* components. Subversion [31, 137] is a version management system [159, 180]. Subversion clients can access a central version management repository through various means: local file system access, WebDAV (an extension to HTTP to support authoring and versioning [30]), and the specialised svnserve protocol. These various access methods lead to a number of optional features, which in turn lead to *optional dependencies*[5]:

- Subversion has two storage back-ends for repositories: one that uses a built-in file system based storage, and one that uses the external *Berkeley DB* embedded database library. Use of the latter back-end induces a dependency on the Berkeley DB component.

- Subversion can build an *Apache module* that allows it to act as a WebDAV server for remote clients. This feature requires the Apache HTTPD server component.

- Subversion can access remote WebDAV repositories over HTTP by default. If access over HTTPS (i.e., HTTP encrypted over the Secure Sockets Layer [76]) is desired, then it requires the OpenSSL cryptographic library component.

- Data sent over HTTP connections can be compressed using the Zlib algorithm, which requires the Zlib library component.

In other words, Subversion has quite a bit of build-time variability. Figure 2.9 shows how such variability can be expressed in the Nix expression language. For instance, the optional support for SSL encryption is expressed by giving the function a Boolean parameter sslSupport $\boxed{17}$ that specifies whether to build a Subversion instance with SSL support.

[5] There are actually even more dependencies than listed here, such as bindings for various languages. These have been omitted to keep the example short.

QUESTION See callout 20 below, but what I'm confused about is: so **(1)** are the dep arguments to a Nix **derivation function** (=/= derivation) derivations or still Nix expression? (By the way, **(2)** is a Nix derivation a valid Nix expression?)

```
{ bdbSupport ? false
, httpServer ? false
, sslSupport ? false [17]
, compressionSupport ? false
, stdenv, fetchurl
, openssl ? null [18], httpd ? null, db4 ? null, expat, zlib ? null
}:

assert bdbSupport -> db4 != null; [19]
assert httpServer -> httpd != null && httpd.expat == expat; [20]
assert sslSupport -> openssl != null &&
  (httpServer -> httpd.openssl == openssl);
assert compressionSupport -> zlib != null;

stdenv.mkDerivation {
  name = "subversion-1.2.1";

  builder = ./builder.sh;
  src = fetchurl {
    url = http://subversion.tigris.org/downloads/subversion-1.2.1.tar.bz2;
    md5 = "0b546195ca794c327c6830f2e88661f7";
  };

  openssl = if sslSupport then openssl else null; [21]
  zlib = if compressionSupport then zlib else null;
  httpd = if httpServer then httpd else null;
  db4 = if bdbSupport then db4 else null;

  inherit expat bdbSupport httpServer sslSupport;
}
```

"httpd" here is obviously such a dep argument to "mkDerivation" here and it appears to be an attribute set as the usage of "httpd.expat" hints. Based on the para explaining 20 states that "httpd.expat" (and, as a corollary, "expat") evaluates to the Nix store paths.

NOTE -> is logical implication.
e1 -> e2 is syntactic sugar for
!e1 || e2

(-> has the lowest
precedence of any operator)

NOTE So this becomes "inputSrc" in the derivation, and "builder" becomes bash/bin/sh ...

NOTE
assert :: Bool -> Expr
assert e1; e2

if "e1" evaluates to true, "e2" is executed

if "e1" evaluates to false, the entire Nix expression will crash with a backtrace

NOTE: This is a long chain of "assert" statements in the form of "**assert e1; assert e2; ...; assert en-1; en**" - if any of e1..en-1 evaluates to false, the Nix expression will crash

Figure 2.9.: pkgs/applications/version-management/subversion/default.nix: Nix expression for Subversion

This is distinct from the passing of the actual OpenSSL dependency [18] in order to separate logical features from the dependencies required to implement them, if any. Many features require no dependencies, or require multiple dependencies. The list of function arguments also shows the language feature of *default arguments*, which are arguments that may be omitted when the function is called. If they are omitted, the default is used instead.

A call to the Subversion function (e.g., in all-packages-generic.nix) to build a minimal Subversion component will look like this:

```
subversion = (import .../subversion) {
  inherit fetchurl stdenv expat;
};
```

On the other hand, the following call builds a full-featured Subversion component:

```
subversion = (import .../subversion) {
    inherit fetchurl stdenv openssl db4 expat zlib;
    httpd = apacheHttpd;
    localServer = true;
    httpServer = true;
    sslSupport = true;
    compressionSupport = true;
  };
```

There are some other language aspects worth noting in Figure 2.9. *Assertions* enable consistency checking between components and feature selections. The assertion in $\boxed{\textbf{19}}$ verifies that if Berkeley DB support is requested, the db4 dependency should not be null. The value null is a special constant that allows components to be omitted, which would be wrong here. Note that the default for the db4 argument *is* in fact null, which is appropriate for bdbSupport's default of false. But if the function is called with bdbSupport set to true, then the assertion protects the user against forgetting to specify the db4 parameter.

see questions there

The next two assertions $\boxed{\textbf{20}}$ express consistency requirements *between* components. The first one states that if the Apache WebDAV module is to be built, then Apache's expat dependency (Expat is an XML parsing library) should be *exactly equal* to Subversion's expat dependency, where "exactly equal" means that they should be in the same store path. This is because the dynamic libraries of Apache and Subversion will end up being linked against each other. If both bring in *different* instances of Expat, a runtime link error will ensue. Hence the requirement that both Expats are to be the same. Likewise, if SSL *and* WebDAV support are enabled, Apache and Subversion should use the same OpenSSL libraries.

As in the Hello example, the dependencies must be passed to the build script. But we only pass the optional dependencies if their corresponding features are enabled $\boxed{\textbf{21}}$. If they are disabled, the value null is passed to the builder (null corresponds to an empty string in the environment variables). This is to relieve the programmer from having to figure out at the call site what specific dependencies should be passed for a given feature selection. After all, that would break abstraction—such decisions should be made by the component's Nix expression, not by the caller. Thus, the caller can always pass *all* dependencies, and the component's Nix expression will figure out what dependencies to use. As we will see in Section 4.1 (page 62), unused dependencies will not be evaluated, let alone be built, due to the fact that Nix expressions are a *lazy* language (values are only computed when needed).

Figure 2.10 shows the builder for the Subversion component, just to give an idea of what is involved in getting a non-trivial piece of software to work in Nix. This builder is at a higher level than the one we have seen previously for Hello (Figure 2.7), as it uses the *generic builder*, which is a shell function defined by the standard environment that takes care of building most Unix components with little or no customisation. Most modern Unix software has a standard build interface consisting of the following sequence of commands:

```
tar xvf ...sources.tar...
./configure
make
make install
```

```
buildInputs="$openssl $zlib $db4 $httpd $expat" 22
source $stdenv/setup

configureFlags="--without-gdbm --disable-static"

if test "$bdbSupport"; then 23
    configureFlags="--with-berkeley-db=$db4 $configureFlags"
fi

if test "$sslSupport"; then
    configureFlags="--with-ssl --with-libs=$openssl $configureFlags"
fi

if test "$httpServer"; then
    configureFlags="--with-apxs=$httpd/bin/apxs --with-apr=$httpd ..."
    makeFlags="APACHE_LIBEXECDIR=$out/modules $makeFlags"
else
    configureFlags="--without-apxs $configureFlags"
fi

installFlags="$makeFlags"

genericBuild 24
```

TODO/QUESTION So, again, an arg to a derivation argument resolves to a store path (in essence, to a string).

Figure 2.10.: pkgs/applications/version-management/subversion/builder.sh: Builder for Subversion

This is in essence what the generic builder does. For many components, a simple call $\boxed{24}$ to the generic builder suffices. The generic builder is discussed in more detail in Section 7.1.2.

So the effort to build Subversion is quite modest: it is a matter of calling Subversion's configure script with the appropriate flags to tell it where it can find the dependencies. For instance, the configure flags to enable and locate Berkeley DB are set in $\boxed{23}$. Actually, following the Autoconf philosophy of auto-detecting system capabilities, some of the optional features are enabled automatically if the corresponding dependency is found. For example, Zlib compression support is enabled automatically if configure finds Zlib in the compiler and linker search path. The variable buildInputs contains a list of components that should be added to such search paths; thus all components added in $\boxed{22}$ are in the scope of the compiler and linker.

TODO Dig into this topic deeper. Is buildInputs a special variable name for genericBuild to be added to the Autoconf path(s)?

Learn more about autotools too.

This concludes the brief overview of the Nix expression language. Chapter 4 provides a more in-depth discussion of the syntax and semantics of the language.

2.3. Package management

Installing Now that we have written Nix expressions for Hello, we want to build and install the component. That is, we want to reach a state where the Hello component is in some way "available" to the user. In a Unix system, this means that the application should

TODO Tread lightly here; this only shows imperative pkg mgmt, but it can muddy the waters, especially if one already has a declarative setup (e.g., because their system supports it - QUESTION **When is declarative pkg mgmt not supported? On non-NixOS systems? That is why home manager is there?**) Many (or most of the) times, one really wants to use nix-shell -p instead of imperative methods.

TODO The CLI commands have also changed considerably and their evolution left messy wakes (pre vs post 2.0, etc.); disentangle it, there will be many post on discourse alone.

be in the user's search path:

```
$ hello
Hello, world!
```

while in different settings it might mean that the application becomes available as an icon on the desktop, or as a menu item in the Start Menu.

On Unix, operations such as installation, upgrading, and uninstallation are performed through the command nix-env, which manipulates so-called *user environments*—the views on "available" components (discussed below on page 36). For instance,

This is probably a typo

```
$ nix-env -f .../all-packages.nix -i hello
installing 'hello-2.1.1'
```

There is also -I nixpkgs=... and others; as stated in the TODO above, figure this out.

evaluates the Nix expression in all-packages.nix, looks for a derivation with symbolic name hello, builds the derivation, and makes the result available to the user. The building of the derivation is a recursive process: Nix will first build the dependencies of Hello, which entails building *their* dependencies, and so on. However, if a derivation has been built previously, it will not be rebuilt.

Queries It is of course possible to query the set of installed components:

```
$ nix-env -q
hello-2.1.1
```

This query prints the set of components in the current user environment. Likewise, it is possible to query which components are *available* for installation in a Nix expression:

```
$ nix-env -f .../all-packages.nix -qa
a52dec-0.7.4
acrobat-reader-7.0
alsa-lib-1.0.9
ant-blackdown-1.4.2
ant-j2sdk-1.4.2
ant-j2sdk-1.5.0
...
```

It is usually not convenient to specify the Nix expression all the time. Therefore it is possible to specify a specific Nix expression as the default for package management operations:

```
$ nix-env -I .../all-packages.nix
```

TODO I don't think this syntax of -I is valid anymore.

All subsequent nix-env invocations will then use this file if no expression is explicitly specified.

Upgrades and rollbacks When a new version of the set of Nix expressions that contains Hello comes along, we can upgrade the installed Hello:

```
$ hello -v
hello - GNU hello 2.1.1
```

```
$ nix-env -f .../all-packages.nix -i hello
upgrading ‘hello-2.1.1’ to ‘hello-2.1.2’

$ hello -v
hello - GNU hello 2.1.2
```

This is standard procedure for a package management system, completely analogous to, e.g., rpm -i hello and rpm -U hello. However, if it turns out that the new version of Hello does not meet our requirements, we can *roll back* to the previous version:

```
$ nix-env --rollback

$ hello -v
hello - GNU hello 2.1.1
```

The reason that this is possible is that the new version of Hello resides in a different path in the Nix store, since at least some inputs of the build process of the new Hello must be different (e.g., its sources and its name attribute). The old one is still there. There is no destructive upgrading: in the purely functional model, components never change after they have been built.

User environments User environments are Nix’s mechanism for allowing different users to have different configurations, and to do atomic upgrades and rollbacks. Clearly, we want to abstract over store paths; it would not do for users to have to type

```
$ /nix/store/dpmvp969yhdq...-subversion-1.1.3/bin/svn
```

to start a program.

Of course we could set up the PATH environment variable to include the bin directory of every component we want to use, but this is not very convenient since changing PATH doesn’t take effect for already existing processes. The solution Nix uses is to create directory trees of symlinks to *activated* components (similar to systems such as GNU Stow [78])[6]. These are called *user environments* and they are components themselves (though automatically generated by nix-env), so they too reside in the Nix store. In Figure 2.11 the user environment /nix/store/0c1p5z4kda11...-user-env contains a symlink to just Subversion 1.1.2 (dashed arrows in the figure indicate symlinks). This is what we would have obtained if we had performed

```
$ nix-env -i subversion
```

on a set of Nix expressions that contained Subversion 1.1.2.

This does not in itself solve the problem, of course; a user would not want to type /nix/store/0c1p5z4kda11...-user-env/bin/svn either. This is why there are symlinks outside of the store that point to the user environments in the store, e.g., the symlinks default-42-link and default-43-link in the example. These are called *generations* since whenever one performs a nix-env operation, a new user environment is generated based on the current one. For instance, generation 43 was created from generation 42 when the user executed

[6] While this approach is rather Unix-specific, user environments can be implemented in many other ways, as is discussed in Section 7.2.

TODO Did "sudo find /nix/store -name '*user-env*'", found a couple results, but be aware that (1) couldn't find anything named "generation" (obviously), and (2) there is a specific directory structure that may not be documented anywhere.

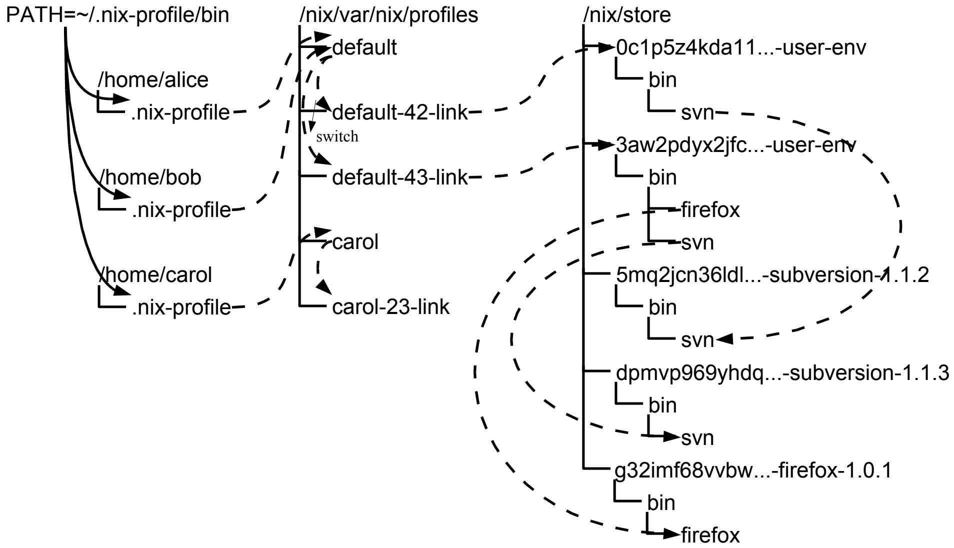


Figure 2.11.: User environments

```
$ nix-env -i subversion firefox
```

on a set of Nix expressions that contained Firefox and a new version of Subversion.

Generations are grouped together into *profiles* so that different users don't interfere with each other (unless they share a profile). For example, the links default-42-link and default-43-link are part of the profile called default. The file default itself is a symlink that points to the current generation. When we perform a nix-env operation, a new user environment and generation link are created based on the current one, and finally the default symlink is made to point at the new generation.

Upgrades are *atomic*. This means that a user can never observe that an upgrade is "half-way through"; the user either sees all the components of the old user environment, or all the components of the new user environment. This is a major improvement over most deployment systems which work by destructive upgrading. Due to the purely functional model, as we are building new components (including user environments, which are components also), the components in the user's current user environment are left entirely untouched. They are not in any way changed. The last step in the upgrade—switching the current generation symlink—can be performed in an atomic action on Unix.

Different users can have different profiles. The user's current profile is pointed to by the symlink ~/.nix-profile (where ~ denotes the user's home directory). Putting the directory ~/.nix-profile/bin in the user's PATH environment variables completes the user environments picture (Figure 2.11), for if we now resolve the chain of symlinks from ~/.nix-profile/bin, we eventually end up in the bin directory of the current user environment, which in turn contains symlinks to the activated applications. A user can create or switch to a new profile:

```
$ nix-env --switch-profile /nix/var/nix/profiles/my-new-profile
```

which will simply flip the ~/.nix-profile symlink to the specified profile. Subsequent nix-env operations will then create new generations in this profile.

Uninstalling and garbage collection A component can be removed from a user environment:

```
$ nix-env -e hello
uninstalling ‘hello-2.1.1’

$ hello -v
bash: hello: No such file or directory
```

However, nix-env -e does not actually remove the component from the Nix store. This is because the component might still be in use by the user environment of another user, or be a dependency of a component in some environment. More importantly, if the component were removed, it would no longer be possible to perform a rollback.

To reclaim disk space, it is necessary to periodically run the *garbage collector*, which deletes from the store any components not *reachable* from any generation of any user environment. (Reachability is defined with respect to the runtime dependencies found by the hash scanning approach discussed earlier, and in more detail in Section 3.4.) That is, the generations are *roots* of the garbage collector.

This means that the Hello component cannot be garbage collected until the old generations that included it are gone. The following command removes any generations of the user's profile except the current:

```
$ nix-env --remove-generations old
```

Note that this removes the ability to roll back to any configuration prior to the invocation of this command. Thus, the user should only perform this operation when sure that rollback is not necessary.

After we have removed the old generations, the Hello component has become garbage, assuming no generations of other profiles reach it. Thus an execution of the garbage collector will remove the component from the Nix store:

```
$ nix-store --gc
deleting ‘/nix/store/bwacc7a5c5n3...-hello-2.1.1’
...
```

(The command nix-store performs "low-level" Nix store operations.)

The advantage of garbage collection of components is that users never need to consider whether some dependency might still be needed. A common phenomenon in many deployment systems is that users are asked to confirm whether it is safe to remove some component (e.g., "Should Windows delete foo.dll?"). Since Nix ensures reliable dependency information, garbage collection can be safe and automatic. The act of uninstallation is a *logical* action concerning only top-level components (i.e., applications), not dependencies that should remain hidden to users.

TODO I wonder though if anything has changed regarding this process.

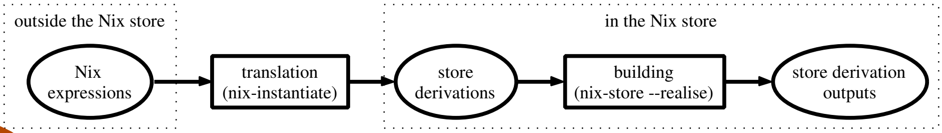


Figure 2.12.: Two-stage building of Nix expressions

2.4. Store derivations

So how do we build components from Nix expressions? This could be expressed directly in terms of Nix expressions, but there are several reasons why this is a bad idea. First, the language of Nix expressions is fairly high-level, and as the primary interface for developers, subject to evolution; i.e., the language might change to accommodate new features. However, this means that we would have to be able to deal with variability in the Nix expression language itself: several versions of the language would need to be able to co-exist in the store. Second, the richness of the language is nice for users but complicates the sorts of operations that we want to perform (e.g., building and deployment). Third, Nix expressions cannot easily be identified uniquely. Since Nix expressions can import other expressions scattered all over the file system, it is not straightforward to generate an identifier (such as a cryptographic hash) that uniquely identifies the expression. Finally, a monolithic architecture makes it hard to use different component specification formalisms on top of the Nix system (e.g., we could retarget Makefiles to use Nix as a back-end).

For these reasons Nix expressions are not built directly; rather, they are translated to the more primitive language of *store derivations*, which encode single component build actions. This is analogous to the way that compilers generally do the bulk of their work on simpler intermediate representations of the code being compiled, rather than on a full-blown language with all its complexities. Store derivations are placed in the Nix store, and as such have a store path too. The advantage of this two-level build process is that the paths of store derivations give us a unique identification of objects of source deployment, just as paths of binary components uniquely identify objects of binary deployment.

Figure 2.12 outlines this two-stage build process: Nix expressions are first *translated* to store derivations that live in the Nix store and that each describe a single build action with all variability removed. These store derivations can then be *built*, which results in derivation outputs that also live in the Nix store.

11.

12.

The exact details of the translation of Nix derivation expressions into store derivations is not important here; the process is fully described in Section 5.4. What matters is that each derivation is translated recursively into a store derivation. This translation is performed automatically by nix-env, but it can also be done explicitly through the command nix-instantiate. For instance, supposing that foo.nix is a Nix expression that selects the value hello from all-packages-generic.nix in Figure 2.8, then applying nix-instantiate will yield the following:

NOTE This is probably the part whre things started getting muddied; to be pedancitc, the Nix expression calling mkDerivation (that in turn calls the "derivation" primitive) gets translated into a **store derivation**.

NOTE Wonder why it wasn't called "nix-translate"...

```
$ nix-instantiate foo.nix
/nix/store/1ja1w63wbk5qrscwg4wmzk9cbri4iykx-hello-2.1.1.drv
```

That is, the expression foo.nix is evaluated, and the resulting top-level derivation is trans-

NOTE This is **not** how .drv files look like; see page 104 where the truth is revealed. The below is a Nix expression representation (more specifically, an attribute set) of the ATerm terms stored in .drv files. The .drv files can be translated to this using "nix show-derivation" command; see https://github.com/toraritte/nix.land/tree/main/derivation-mysteries .

```
{   output = "/nix/store/bwacc7a5c5n3...-hello-2.1.1" [25]
,   inputDrvs = { [26]
        "/nix/store/7mwh9alhscz7...-bash-3.0.drv",
        "/nix/store/fi8m2vldnrxq...-hello-2.1.1.tar.gz.drv",
        "/nix/store/khllx1q519r3...-stdenv-linux.drv",
        "/nix/store/mjdfbi6dcyz7...-perl-5.8.6.drv" [27] }
    }
,   inputSrcs = {"/nix/store/d74lr8jfsvdh...-builder.sh"} [28]
,   system = "i686-linux" [29]
,   builder = "/nix/store/3nca8lmpr8gg...-bash-3.0/bin/sh" [30]
,   args = ["-e", "/nix/store/d74lr8jfsvdh...-builder.sh"] [31]
,   envVars = { [32]
        ("builder", "/nix/store/3nca8lmpr8gg...-bash-3.0/bin/sh"),
        ("name", "hello-2.1.1"),
        ("out", "/nix/store/bwacc7a5c5n3...-hello-2.1.1"),
        ("perl", "/nix/store/h87pfv8klr4p...-perl-5.8.6"), [33]
        ("src", "/nix/store/h6gq0lmj9lkg...-hello-2.1.1.tar.gz"),
        ("stdenv", "/nix/store/hhxbaln5n11c...-stdenv-linux"),
        ("system", "i686-linux"),
        ("gtk", "/store/8yzprq56x5fa...-gtk+-2.6.6"),
    }
}
```

Personal note: I was never a fan of layering multiple models of increasing complexity when introducing a topic. In footnote 7 below it is pointed out that this is not the real derivation representation but for one skimming through the text it will be misleading.

QUESTION **inputDrvs vs envVars**
Why are the input derivations needed if the paths of the build components are already available? Is it because of the recursive nature of evaluating derivations (and because their outputs are stored in different store paths)?

Figure 2.13.: Store derivation for the hello attribute in Figure 2.8

lated to a store derivation which is placed in /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv. All input derivations of hello are recursively translated and placed in the store as well.

The store derivation in /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv is shown in Figure 2.13[7]. It lists all information necessary to build the component, with references to the store derivations of dependencies:

[25] The output field specifies the path that will be built by this derivation, if and when it is built[8]. It is computed essentially by hashing the store derivation with the output field cleared.

[26] The inputDrvs field contains the paths of all the input derivations. These have to be built before we can build this derivation.

[28] The inputSrcs field contains the paths of all input sources in the Nix store. The Hello Nix expression in Figure 2.8 referred to a local file builder.sh. This file has been copied by nix-instantiate to the Nix store, since everything used by the build process must be inside the Nix store to prevent undeclared dependencies.

[7] This is a pretty-printed representation of the actual store derivation, which is encoded as an ATerm [166]. Figure 5.8 on page 105 shows the actual ATerm.

[8] This field exists in the extensional model described in Chapter 5, where the output path is computed in advance, but not in the intensional model described in Chapter 6, where it is only known after the derivation has been built. (The "output" field in the derivation.)

QUESTION So both "inputDrvs" and "inputSrcs" refer to dependencies but it is not as simple as **build-time** or **runtime** deps. Their descriptions at 26 and 28 are not satisfying to me. How to formally define these? "inputDrvs" seems to be the software dependencies, whereas "inputSrcs" seems to be supporting files that help (and/or drive) the build process.
TODO Take a look at
https://comono.id/posts/2020-03-20-how-nix-instantiation-works/

NOTE I think using "builder" was also unfortunate... It usually defaults to bash, and "inputSrc" to "default-builder.sh". So something like build-interpreter and build-script would have been better (even if more verbose).

[29] The system field specifies the hardware and operating system system on which the derivation is to be built. This field is part of the hash computation, so derivations for different systems (e.g., i686-linux versus powerpc-darwin) result in different store paths.

[30] The builder field is the program to be executed. The reader might wonder about the apparent mismatch between the builder declared in Figure 2.6 and the one specified here. This is because the function stdenv.mkDerivation massages the arguments it receives a bit, substituting a shell binary as the actual builder, with the original shell script passed to the shell binary as a command line argument.

NOTE Yep, in the derivation (in the ATerm file) "builder.sh" is stored as "["-e","/nix/store/9krlzvny65gdc8s7kpb6lkx8cd02c25b-default-builder.sh"]" where "-e" is a clear indication that is a bash flag.

[31] The args field lists the command line arguments to be passed to the builder.

[32] The envVars field lists the environment variables to be passed to the builder.

See note at figure 2.13

Note that inputDrvs and envVars specify different paths for corresponding dependencies. For example, Hello's Perl dependency is built by the derivation stored in /nix/store/mjdfbi6dcyz7...-perl-5.8.6.drv [27]. The output path of this derivation is /nix/store/-h87pfv8klr4p...-perl-5.8.6 [33].

Building store derivations Store derivations can be built by calling the command nix-store --realise, which "realises" the effect of a derivation in the file system. It builds the derivation (if the output path of the derivation does not exist already), and prints the output path.

```
$ nix-store --realise /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv
...
/nix/store/bwacc7a5c5n3...-hello-2.1.1
```

Nix users do not generally have to deal with store derivations. For instance, the nix-env command hides them entirely—the user interacts only with high-level Nix expressions, which is really just a fancy wrapper around the two commands above. However, store derivations are important when implementing deployment policies. Their relevance is that they give us a way to uniquely identify a component both in source and binary form, through the paths of the store derivation and the output, respectively. This can be used to implement a variety of deployment policies, as we will see in Section 2.5.

QUESTION The binary form because of the "output" field, and the source from because of the drv itself? Does the model matter (i.e., extensional/intensional)?

ANSWER The answer to the 1st question is literally on the next line XDDD

Closures revisited A crucial operation for deployment is to query the closure of a store path under the dependency relation, as discussed in Section 2.1. Nix maintains this information for all paths in the Nix store. For instance, the closure of the Hello component can be found as follows:

TODO Is this still the only way to find the closure of a Nix expression?

```
$ nix-store -qR /nix/store/bwacc7a5c5n3...-hello-2.1.1/bin/hello
/nix/store/72by2iw5wd8i...-glibc-2.3.5
/nix/store/bg6gi7s8mxir...-linux-headers-2.6.11.12-i386
/nix/store/bwacc7a5c5n3...-hello-2.1.1
/nix/store/q2pw1vn87327...-gcc-3.4.4
```

This means that when we deploy this instance of Hello, we must copy all four of the paths listed here[9]. When we copy these paths, we have *binary deployment*. It should be pointed out that /nix/store/h87pfv8klr4p...-perl-5.8.6 is not in the closure, since it is not referenced from the Hello binary or any of the paths referenced by it. That is, Perl is exclusively a build-time dependency.

Similarly, we can query the closure of the store derivation:

```
$ nix-store -qR /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv
/nix/store/0m055mdm0v5z...-perl-5.8.6.tar.bz2.drv
/nix/store/1ja1w63wbk5q...-hello-2.1.1.drv
/nix/store/d74lr8jfsvdh...-builder.sh
... (98 more paths omitted) ...
```

Copying these paths gives us *source deployment*, since the store derivations describe build actions from source.

It is also possible to combine the two:

```
$ nix-store -qR --include-outputs \
    /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv
/nix/store/0m055mdm0v5z...-perl-5.8.6.tar.bz2.drv
/nix/store/h87pfv8klr4p...-perl-5.8.6
/nix/store/1ja1w63wbk5q...-hello-2.1.1.drv
/nix/store/bwacc7a5c5n3...-hello-2.1.1
/nix/store/d74lr8jfsvdh...-builder.sh
... (155 more paths omitted) ...
```

The --include-outputs flag causes the closures of the outputs of derivations to be included. It is worth noting that this set is more than the union of the two sets above, since there are many output paths of derivations that are not in the closure of the Hello binary. For instance, though the Hello binary does not reference Perl, the Perl binary is included here because its derivation is included.

2.5. Deployment models

What we have described above is not deployment—it is build management combined with local package management. The calls to nix-env build components locally, and construct user environments out of the build results. To perform actual software deployment, it is necessary to get the Nix expressions to the client machines. Nix in itself does not provide any specific way to do so. Rather, it provides the *mechanisms* to allow various *deployment policies*. The commands nix-instantiate and nix-store provide the fundamental mechanisms. (The command nix-env implemented on top of these already implements a bit of policy, namely, user environments.)

Here are some examples of deployment models implemented and used in practice to distribute the Nix expressions of the Nix Packages collection:

- **Manual download.** A user can manually download releases of the Nix Packages collection as tar archives, unpack them, and apply nix-env to install components.

[9] In case the reader is wondering: the quite unnecessary dependency on linux-headers is through glibc; see Section 6.7. The dependency on gcc is entirely unnecessary and does not occur in the "production" version of Hello, which uses the utility patchelf (page 179) to prevent this retained dependency.

TODO Understand this part...

The downside to this approach is that it is rather laborious; in particular, it is not easy to stay up-to-date with new releases.

- **Updating through a version management system.** As a refinement to the previous model, the set of Nix expressions can be distributed through a version management system. For example, Nixpkgs can be obtained from its Subversion repository. After the initial checkout into a local working copy, staying up-to-date is a simple matter of running an update operation on the working copy. An added advantage is that it becomes easy to maintain local modifications to the Nix expressions.

TODO Understand this. Are channels being deprecated with flakes? Also flox channels seem to be fundamentally different...

- **Channels.** A more sophisticated model is the notion of *channels*. A channel is a URL that points to a tar archive of Nix expressions. A user can subscribe to a channel:

  ```
  $ nix-channel --add http://server/channels/nixpkgs-stable
  ```

  Then, the command

  ```
  $ nix-channel --update
  ```

  downloads the Nix expressions from the subscribed channels and makes them the default for nix-env operations. That is, an operation such as

  ```
  $ nix-env -i firefox
  ```

  QUESTION Why is "nix-env -i" always slower than "nix-env -f <nix-kgs> -iA"?

  will install a derivation named firefox from one of the subscribed channels. In particular, it is easy to keep the entire system up-to-date:

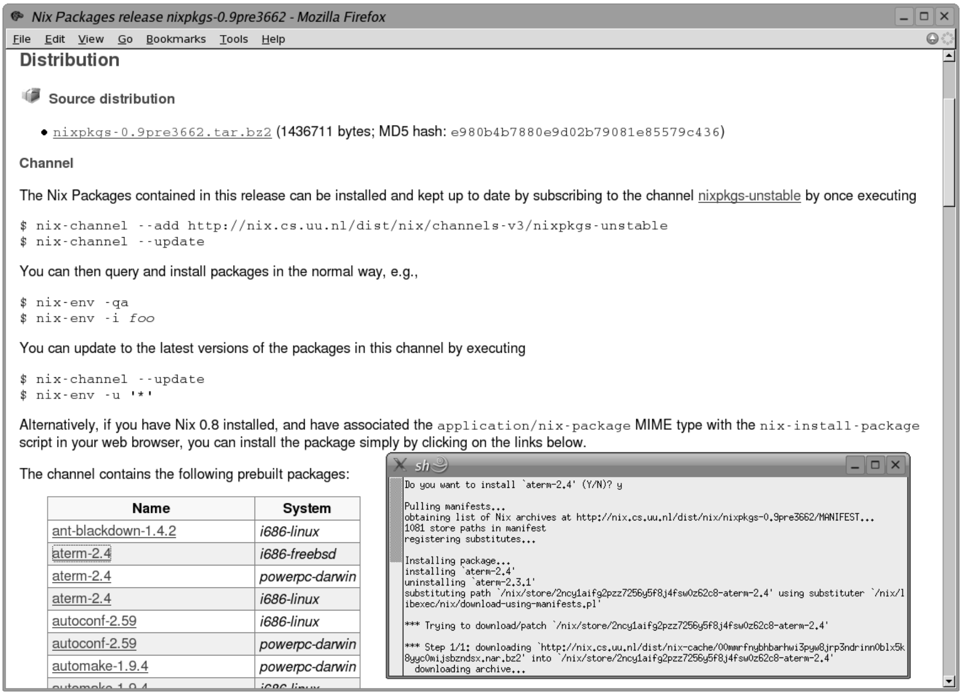
  ```
  $ nix-channel --update
  $ nix-env -u '*'
  ```

  where nix-env -u '*' updates all installed components in the user environment to the latest versions available in the subscribed channels.

TODO I believe these have been deprecated. Or not? At least, this thesis is the only play where I ever heard about this.

- ~~**One-click installations.** When a user wants to install a specific program, by far the easiest method is to simply install it by clicking on it on some distribution web page. Of course, this should also install all dependencies. An example of such *one-click installations* is shown in Figure 2.14 (along with direct download of the Nix expressions, and channel subscription information). Users can click on links on release web pages to install specific components into their user environments. The links lead to small files containing the information necessary to install the component (e.g., Nix expressions), while the installation of the component is performed by a simple tool associated with the MIME type [74] of those files.~~

Chapter 7 looks at the wide variety of deployment policies in much greater detail.



Distribution

Source distribution

- nixpkgs-0.9pre3662.tar.bz2 (1436711 bytes; MD5 hash: e980b4b7880e9d02b79081e85579c436)

Channel

The Nix Packages contained in this release can be installed and kept up to date by subscribing to the channel nixpkgs-unstable by once executing

```
$ nix-channel --add http://nix.cs.uu.nl/dist/nix/channels-v3/nixpkgs-unstable
$ nix-channel --update
```

You can then query and install packages in the normal way, e.g.,

```
$ nix-env -qa
$ nix-env -i foo
```

You can update to the latest versions of the packages in this channel by executing

```
$ nix-channel --update
$ nix-env -u '*'
```

Alternatively, if you have Nix 0.8 installed, and have associated the `application/nix-package` MIME type with the `nix-install-package` script in your web browser, you can install the package simply by clicking on the links below.

The channel contains the following prebuilt packages:

Name	System
ant-blackdown-1.4.2	*i686-linux*
aterm-2.4	*i686-freebsd*
aterm-2.4	*powerpc-darwin*
aterm-2.4	*i686-linux*
autoconf-2.59	*i686-linux*
autoconf-2.59	*powerpc-darwin*
automake-1.9.4	*powerpc-darwin*
automake-1.9.4	*i686-linux*

```
Do you want to install `aterm-2.4' (Y/N)? y

Pulling manifests...
obtaining list of Nix archives at http://nix.cs.uu.nl/dist/nix/nixpkgs-0.9pre3662/MANIFEST...
1081 store paths in manifest
registering substitutes...

Installing package...
installing `aterm-2.4'
uninstalling `aterm-2.3.1'
substituting path `/nix/store/2ncy1aifg2pzz7256y5f8j4fsw0z62c8-aterm-2.4' using substituter `/nix/l
ibexec/nix/download-using-manifests.pl'

*** Trying to download/patch `/nix/store/2ncy1aifg2pzz7256y5f8j4fsw0z62c8-aterm-2.4'

*** Step 1/1: downloading `http://nix.cs.uu.nl/dist/nix-cache/00mmrfnybhbarhwi3pyw8jrp3ndrinn0blx5k
8yyc0mijsbzndsx.nar.bz2' into `/nix/store/2ncy1aifg2pzz7256y5f8j4fsw0z62c8-aterm-2.4'
  downloading archive...
```

Figure 2.14.: Deployment policies: direct download, channels, and one-click installation

2.6. Transparent source/binary deployment

The Nix model as described above is a source deployment model. Nix expressions describe how to build components from source[10]. A source deployment model is convenient for deployers because they do not need to create binary packages explicitly. It also gives users the freedom to make local modifications and feature selections.

However, building all dependencies is rather slow. For the Hello component in Nix Packages[11], it entails building 63 derivations, including the builds of

> hello-2.1.1, stdenv-linux, gcc-3.4.4, bash-3.0, gnutar-1.15.1, curl-7.14.0, binutils-2.16.1, gnupatch-2.5.4, gnused-4.1.4, patchelf-0.1pre2286, linux-headers-2.6.11.12-i386, zlib-1.2.2, bzip2-1.0.3, findutils-4.2.20, gnugrep-2.5.1a, coreutils-5.2.1, perl-5.8.6, gcc-3.4.4, gnumake-3.80, gzip-1.3.3, gawk-3.1.4, pcre-6.0, glibc-2.3.5, diffutils-2.8.1, gcc-3.4.4,

the fetchurl downloads of the sources of those packages, and some derivations for the bootstrapping of stdenv (e.g., the statically linked compiler used to compile gcc-3.4.4); with

[10] As we will see in Section 7.1.4, it is possible to support binary-only packages by using a "trivial" builder that just fetches a binary distribution of a component, unpacks it and copies it to $out. This is how the Nix Packages collection supports components such as Adobe Reader.

[11] Data based on nixpkgs-0.9pre3252 on i686-linux.

downloads of source archives totalling 130 megabytes; and with the build results occupying 357 megabytes of disk space (although most of it can be garbage collected).

Thus, source deployment is clearly awful for most end-users, who do not have the resources or patience for a full build from source of the entire dependency graph.

However, Nix allows the best of both worlds—source deployment and binary deployment—through its *transparent source/binary deployment model*. The idea is that pre-built derivation outputs are made available in some central repository accessible to the client machines, such as a web server or installation CD-ROM. For instance, a component distributor or system administrator can pre-build components, then *push* (upload) them to a server using PUT requests:

QUESTION How does this work exactly nowadays? I know there is hydra but even sure about the basics. (E.g., what happens when I issue "nix-shell -p ..."?)

QUESTION This is another command I know nothing about...

```
$ nix-push http://server/dist/ $(nix-instantiate foo.nix)
```

This will build the derivations in foo.nix, if necessary, and upload it and all its dependencies to the indicated site. For instance, the path /nix/store/mkmpxqr8d7f7...-firefox-1.0 will be archived and compressed into an archive yq318j8lal09...-firefox.nar.bz2 and uploaded to the server. Such a file is called a *substitute*, since a client machine can substitute it for a build. The server provides a *manifest* of all available substitutes. An example entry in the manifest is[12]: TODO Has this mechanism changed?

TODO Manifests are not covered in the official docs at all.. https://github.com/toraritte/knowledge-gaps/blob/master/nix/nix-notes.md#buildenv-and-manifests

There are also hints that manifests are deprecated. See above link.

```
{ StorePath: /nix/store/mkmpxqr8d7f7...-firefox-1.0
  NarURL: http://server/dist/yq318j8lal09...-firefox.nar.bz2
  Size: 11480169 }
```

which states that to create store path /nix/store/mkmpxqr8d7f7...-firefox-1.0, Nix can download and unpack URL http://server/dist/yq318j8lal09...-firefox.nar.bz2.

To use these substitutes, client machines must register the fact that they are available using the command nix-pull:

```
$ nix-pull http://server/dist
```

This does not actually download any of the substitutes, it merely registers their availability. When the user subsequently attempts to install a Nix expression, and Nix when building some store path p notices that it knows a substitute for p, it will download the substitute and unpack it instead of building from source.

Thus, from a user perspective, source deployment automatically optimises into binary deployment. This is good: binary deployment can be considered a partial evaluation of source deployment with respect to a specific platform, and such optimisations should be "invisible".

However, if the user has made modifications to parts of the Nix expression, it is possible that some or all of the store paths will be different. In that case, the substitutes fail to be applicable and Nix will build from source. Thus, binary deployment automatically "degrades" back to source deployment.

The transparent source/binary model therefore combines the flexibility of source deployment systems such as Gentoo with the efficiency of binary deployment systems such as RPM.

[12] This is somewhat simplified. The full details of nix-push manifests are in Section 7.3.

Of course, the deployment models described in Section 2.5 support transparent source/-binary deployment. For instance, a Nix channel contains not just a set of Nix expressions, but also the URL of a manifest that clients can pull when they update from the channel. Thus, subscribers to the channel automatically get binary deployment for those derivations in the channel that have been pre-built.

Part II.

Foundations

3. Deployment as Memory Management

The previous chapter introduced the Nix system. It obtains isolation and non-interference between components using a Nix store that stores components under file names that contain a unique cryptographic hash of the component. However, some of the underlying concepts might seem a bit suspicious! For instance, storing components in an essentially flat, rigid "address space" of components is very different from the way software is typically stored in most operating systems[1]. And the hash scanning technique to identify dependencies clearly cannot work in all circumstances.

The purpose of this chapter is to "derive" the basic design of the Nix store by relating the main issues in software deployment to those in memory management in conventional programming languages. As we shall see, conventional deployment tools treat the file system as a chaotic, unstructured component store, similar to how an assembler programmer would treat memory. In contrast, modern programming languages impose a certain *discipline* on memory, such as rigidly defined object layouts and prohibitions against arbitrary pointer formation, to enable features such as garbage collection and pointer safety. The idea is that by establishing a mapping between notions in the two fields, solutions from one field carry over to the other. In particular, the techniques used in conservative garbage collection serve as a sort of *apologia* for the hash scanning approach used to find runtime dependencies.

3.1. What is a component?

It is useful to first say some things about what it is that we are deploying. Of course, we deploy software (although in Chapter 9 on service deployment, we will also see some things being deployed that cannot be called software proper), but what are the *units* of deployment? That is, what do we call the smallest meaningful artifacts that are subject to deployment? It is quite customary in some communities to call these *packages*, but this term has the problem that it implies that the artifact is in *packaged* form. For instance, a .RPM file is a package, but what do we call the corresponding installed object? In RPM terminology, this is also a package, and so the term is ambiguous.

NOTE I love that these questions are asked, and now we are on a track towards a new ontology but unfortunately Nix introduced its own set(s) of ambiguities... (See above and below.)

In the Component-Based Software Engineering (CBSE) community, the term *component* is used for units of deployment. Like some other terms in computer science (such as *object*), this term is overloaded almost to the point of uselessness. In fact, Czyperski [154, Chapter 11] surveys fourteen different definitions. Some definitions are not meaningfully distinguishable from the notion of a module, or are used to identify any code reuse mechanism.

Despite the multitude of definitions, I shall use the term anyway, because I believe that *the essential aspect that should be captured by a definition of components is deployment*.

[1] Indeed, no Linux distribution based on Nix will ever be compliant with the Linux Standards Base [82, 81]!

Otherwise, the notion of component does not essentially add anything not already provided by concepts such as *module*, *class*, *method*, *function*, and other organisational structures provided by programming languages. Components in CBSE have many other concerns than deployment aspects, such as interfaces, contracts, composition techniques, verification of performance and other extra-functional requirements, and so on. However, a discussion at CBSE-2005 revealed that packaging and deployment were the only aspects on which there was a consensus that they should be provided by a component-based system.

In this thesis, I will use the following definition of "software component":

1 - A software component is a software artifact that is subject to automatic composition. It can require, and be required by, other components.

2 - A software component is a unit of deployment.

The first property is of course entailed by the etymology of the word component, except for the stipulation that a composition can be established automatically. Thus, source code fragments printed in a book, while composable through human intervention, are not components. The second property implies that the *purpose* of the notion of a component is to be the principal object of deployment.

This definition has some important implications. For instance, it does not say anything about whether components are in "source" or "binary" form, since such a distinction is rather technology-specific (e.g., what does it mean for interpreted languages such as Python?). But since it has to be automatically composable, it must come with all the necessary infrastructure to support automatic composition. For instance, mere source code is not a component, but source code with the necessary meta-information—such as Makefiles, configuration and build scripts, deployment descriptors, and so on—is a component. So a Python source file in itself is not a component. But it *is* when placed in an RPM package or combined with a Nix expression, since that meta-information makes the component available for automatic composition.

A subtle point is what we mean by "software." Of course, software is a specification that can be executed automatically by a computer (possibly with the aid of other software, such as an interpreter), but not every constituent part of a software system is executable; some of it (such as data) is an adjunct to execution. This means that not all software components need to contain executable code themselves, but are intended for ultimate composition with an executable component that uses them. For instance, in a software system with an online help feature, it might be useful to split off the online help into a separate component so that it can be installed optionally.

The definition above can be considered the greatest common denominator of a number of influential definitions. Czyperski [154, Section 4.1.5] gives the following definition:

CZY !

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

The important aspects of this definition are the following:

- A component is a *unit of independent* deployment. Since it is a unit, it does not have to support partial deployment—the component must be deployed in its entirety.

PERSKI

Since it is independently deployable, it should not cause interference with its target environment, such as other components.

- Thus it must have *explicit context dependencies only*, since otherwise it would place requirements (such as component dependencies) on its target environment that are not formally specified by its deployment metadata.

- It has *contractually specified interfaces*, which enable it to be subject to *third-party* composition, i.e., it is possible for other people's components to use the component.

- It has *no externally observable state*. Different uses of the component over time should not interfere with each other. Note that this does not mean that the *code* of the component must have no side effects, i.e., should be written in a purely functional language. It is perfectly fine for an execution involving the component, e.g., the classes and objects loaded from a component, to have side effects. But the component should not modify its on-disk representation or other static resources in a way that affects future uses of the component. This is exactly the case with Nix components: they can be entirely impure *dynamically*, but their *static* representation—the files on disk—never changes after they have been built.

! !

These properties do not always hold for the things that one might want to deploy. For instance, Mozilla Firefox 1.0 (unless specially prodded not to do so) wants to make modifications to some of its own files when started for the first time, violating the requirement of having no externally observable state. This is bad because it makes security difficult (the files in question should be writable by all potential users), and hinders proper identification in the sense of configuration management (since the "same" component has different contents over time). Likewise, many components are not intended for third-party composition, but are rather specifically tied to certain other components and do not have a well-defined interface. The reason why such components are separate, rather than combined into a single component, is typically to allow optional functionality to be omitted, or for licensing reasons (e.g., some clients not being entitled to access the full feature set of a system).

This thesis's definition differs from Czyperski's in a number of ways:

DOLSTRA

- It does not require *explicit* context dependencies, it merely states that components can have dependencies. Many components have implicit dependencies—which is what makes correct deployment so difficult. Of course, to ensure correct deployment, it is necessary to make implicit dependencies explicit.

- It does not require the possibility of third-party composition. As explained above, many components are not in fact usefully reusable by third parties.

- It does not require contractually specified interfaces. It is enough for components to be composable automatically, which requires some kind of formal interface (since it can be used automatically), but this can be as weak as a symbol table in an ELF library [160], or the existence of a named function in a Perl module.

Czarnecki and Eisenecker [40] have an even more minimalistic definition: components are simply the "building blocks" of software systems that should be composable as easily and flexibly as possible in order to maximise reuse and minimise duplication. However, this definition has the problem that it makes a component nothing more than a general term for reuse mechanisms, as stated above. This is intentional: the authors feel that component is a “natural concept” for which it is hard to give a rigid definition. But a definition that is overly broad is not very useful.

Meyer [119] defines a component as a “modular unit” that can be used by other components, possesses an “official usage description” enabling its use by other components, and is not “tied to any fixed set of clients”. The definition used here does not have the latter two requirements (many components have poorly defined or missing usage descriptions, or are specific to a particular client, as argued above).

So the definition of component used in this thesis is quite minimal. I do not mean to imply that a stronger notion of component is not useful or appropriate in many cases. For instance, component technologies that require strong contracts from components are very valuable. But such kinds of “high level” composition aspects are not the subject of this thesis. This thesis is about the “low level” aspect of *storage* of components: how can we store components so that they do not interfere with each other, that their dependencies are known, and so on? That is, it is about *components in the file system*.

Components in the file system In this thesis, I shall further restrict the notion of a software component s are things that have an independent representation in the file system, i.e., a component consists of a set of files. Of course components do not *have* to exist as discrete objects in the file system. They might be stored in a database (as suggested, e.g., by the Java Language Specification [79, Section 7.2.2]). But this is quite rare, so we will not consider it.

Through the file system, components can *require* functionality from other components, and they can *provide* it. Thus, components need to be plugged into each other, i.e., *be composed*. There are many mechanisms by which a composition can be realised, such as static linking, dynamic linking, inclusion of source files, and so on; and these are just general classes of composition technologies. Concrete realisation mechanisms are legion: Unix static libraries, ELF dynamic libraries, Java JAR files, C preprocessor directives, .NET assemblies, Perl modules, configuration files, etc. These mechanisms also have various *binding times*: some are typically used at build time, others at runtime, with the more nebulous notions of *packaging time* and *installation time* thrown in for good measure.

But what they all have in common is that the composition must be established in the file system. That is, since the components exist in the file system, to compose them, the composition mechanism must be able to find them in the file system. This aspect of composition—establishing the “physical” composition,” so to speak—is often overlooked in the component-based software engineering literature as well as in programming systems. This is unfortunate, since the difficulty in correctly and efficiently establishing and managing the composition in the file system is *the* major source of deployment problems.

3.2. The file system as memory

In this section I recast the deployment problems identified in the introduction in terms of concepts from the domain of memory management in programming languages. That we can do this is not surprising: after all, file systems on the one hand and RAM on the other hand are just two different levels in the storage hierarchy. Where programs manipulate memory cells, deployment operations manipulate the file system. This analogy reveals that the safeguards against abuse of memory applied by programming languages are absent in deployment, and suggests that we can make deployment safe by trying to *impose* similar safeguards.

NOTE I understand what this is trying to say but the first subsentence makes it sound like the safeguards against memory abuse don't apply to deploymnet...

Components, as defined above, exist in the file system and can be accessed through *paths*, sequences of file names that specify a traversal through the directory hierarchy, such as /usr/bin/gcc. We can view a path as an *address*. Consequently, a string representing a path is a *pointer*, and accessing a file through a path is a pointer *dereference*. Thus, component interference due to file overwriting can be viewed as an address collision problem: two components occupy overlapping parts of the address space.

Furthermore, we can view components as representations of *values* or *objects*. Just as objects in a programming language can have references to other objects, so components can have references to other components. If dereferencing a pointer is not possible because a file does not exist, we have the deployment equivalent of a *dangling pointer*.

A component A is dependent on a component B if the set of files that constitutes A *enables* an execution involving A to dereference pointers to the files constituting B. It *enables* an execution to dereference B (rather than that it necessarily *must* dereference B), since A itself might not be executed or even be executable; e.g., when A is a configuration file containing a path to B, and this file is read by some other component C.

TODO I think I got it but make sure.

For example, suppose that we have an application that is dynamically linked against a file /lib/libc.so, the GNU C library on Linux systems. This means that the executable image of the application contains an instruction to the dynamic linker to load the file /lib/libc.so. Thus, the application enables a pointer to that file, since execution of the application will cause a dereference of that file when it is started. But if the file is missing, the deployed application contains a dangling pointer. Such dangling pointers are the result of incomplete deployment, and they must be prevented.

To prevent dangling pointers, we should consider the various ways through which a component A can cause a dereference of a pointer to another component B. Since components are analogous to objects, we can provide analogues to the ways in which a method of a class can obtain and dereference a pointer to another object.

RUNTIME DEPENDENCIES

First, a pointer to B can be obtained and dereferenced by A at *runtime*. This is a form of *late binding*, since the pointer is not passed in when it is built, but rather when it is executed. This may happen through environment variables (such as the PATH program search path on Windows and Unix), program arguments, function arguments (in the case of a library), registry settings, user interaction, and so on. Conceptually, this is similar to the following pseudo-Java:

```
class Foo {
    Foo() {}
    int run(Bar y) { return y.doIt(); }
}
```

We can consider the constructor method Foo() to be the build time of the objects of class Foo, while the method run corresponds to the runtime. Thus any arguments passed to the constructor are analogous to build-time dependencies, while arguments passed to run are analogous to runtime dependencies. In this case, an instance of Foo obtains a pointer to an instance of Bar at runtime.

BUILD-TIME DEPENDENCIES

Second, a pointer to B can be obtained and dereferenced by A at *build time*. In this case, a pointer is passed in at build-time and is completely "consumed", meaning that B is not needed subsequently. It can therefore no longer cause a dangling pointer. Examples include pointers to static libraries or the compiler, which are not usually retained in the result. This is comparable to a constructor that uses a pointer to another object to compute some derived value but that does not store the pointer itself:

```
class Foo {
    int x;
    Foo(Bar y) { x = y.doIt(); }
    int run() { return x; }
}
```

RETAINED DEPENDENCIES

Third, a pointer to B can be obtained by A at *build time* and dereferenced at *runtime*. In this case, a pointer to B is passed to and saved in the build process that constructed A, and is dereferenced during program execution. This is the notion of *retained dependencies* described on page 23. It happens often with Unix-style dynamically linked libraries: the build of an application stores the full path to the library in the RPATH of the application binary (see page 23), which is used to locate the library at program startup. This is equivalent to the constructor of an object storing a pointer that was passed in, which is later dereferenced:

```
class Foo {
    Bar x;
    Foo(Bar y) { x = y; }
    int run() { return x.doIt(); }
}
```

Here, the execution of the constructor is similar to the construction of a component, and the execution of method run() is similar to the use of a component.

DYNAMIC (?) DEPENDENCIES

Finally, and orthogonal to the previous methods, a pointer to B can be obtained using *pointer arithmetic*. Since paths are represented as strings, any form of string manipulation such as concatenation may be used to obtain new paths. If we have a pointer to /usr/bin, then we may append the string gcc to obtain /usr/bin/gcc. Note that the names to be appended may be obtained by dereferencing directories, that is, reading their contents.

It follows from the above that it is hard to find the set of pointers that can be dereferenced by a component. First, pointers may already be present in the source. Second, pointers passed in at build-time may or may not be stored in the component. Obviously, we do not want to distribute a compiler along with an application just because it was used to build it—but other build-time components, such as dynamic libraries, should be. Finally, pointer arithmetic can be used to obtain new pointers in uncontrolled ways. For correct software deployment it is essential that no dangling pointers can occur. Thus, we need a method to prevent these.

Figure 3.1 summarises the mapping of concepts between the domain of memory management in programming language implementation on the one hand, and deployment on the other hand. (The third set of correspondences is first discussed in Section 3.4.)

mapping of concepts between
1. the domain of memory management in programming language implementations, and
2. deployment

Programming Language Domain		*Deployment Domain*
	Concepts	
memory	⇔	disk
objects (values)	⇔	components
addresses	⇔	path names
pointer dereference	⇔	I/O
pointer arithmetic	⇔	string operations
dangling pointer	⇔	reference to absent component
object graph	⇔	dependency graph
	Kinds of dependencies	
calling already constructed object with reference to other object	⇔	runtime dependency through late binding
calling constructor with reference to other object, not stored	⇔	build-time dependency
calling constructor with reference to other object, stored	⇔	retained dependency
	Pointer discipline vs. deployment styles	
languages with total absence of pointer discipline (e.g., assembler)	⇔	typical Unix-style deployment
languages with enough pointer discipline to support conservative garbage collection (e.g., C, C++)	⇔	Nix
languages with full pointer discipline (e.g., Java, Haskell)	⇔	an as-yet unknown deployment style not enabled by contemporary operating systems

Figure 3.1.: Deployment / memory management analogies

3.3. Closures

As noted above, dangling pointers in components are a root cause of deployment failure. Thus, to ensure successful software deployment, we must copy to the target system not just the files that make up the component, but also all files to which it has pointers. Formally, the set of files to be included in the distribution of a component is the *closure* of the set of files in the component under the points-to relationship. Informally, a file that is part of a component is characterised as a tuple (p,c) where p is the path where the file is stored, and c is the contents required at that path. The contents of a path include not just the file contents if p is a regular file, but also metadata such as access permissions. In addition, the file may be a directory, in which case the contents include a mapping from directory entry names to the contents of these directory entries. This will be made more precise in Section 5.2.1.

Now the closure of a set of files C is the smallest set $C' \supseteq C$ satisfying

$$\forall (p,c) \in C' : \forall p_{ref} \in \mathsf{references}(c) : \exists (p',c') \in C' : p_{ref} = p'$$

where $\mathsf{references}(c)$ denotes the set of pointers contained in the contents of c. That is, if a path p is in the closure, then the paths to which it has pointers must be as well.

(p, c) - a file that is part of a component (p - path, c - content)
+ if **p** is a file, **c** includes (1) the **content of the file** and (2) **metadata** (e.g., access permissions)
+ if **p** is a directory, **c** includes a mapping from directory entry names to the contents of these directory entries.

By definition, a closure does not contain dangling pointers. A closure can therefore be distributed correctly to and deployed on another system. Unfortunately, as mentioned in Section 3.2, it is not possible in general to determine the set references(c) because of retained dependencies and pointer arithmetic. In the next section, I propose a heuristic approach to reliably determine this set.

3.4. A pointer discipline

TODO Mention No Starch Press' "Effective C"; haven't finished it yet but seems to go to the very heart of this stuff.

In the previous section we saw that correct deployment without dangling pointers can be achieved by deploying file system closures. Unfortunately, determining the complete set of pointers is hard.

In the domain of programming languages, we see the same problem. Languages such as C [106, 100] and C++ [153, 101] allow arbitrary pointer arithmetic (adding or subtracting integers to pointers, or casting between integers and pointers). In addition, compilers for these languages do not generally emit runtime information regarding record and stack layouts that enable one to determine the full pointer graph. Among other things, this makes it impossible to implement garbage collectors [104, 179] that can accurately distinguish between garbage and live objects. Garbage collectors consist of two logical phases [179]. In the *detection* phase, the collector discovers which objects on the heap are "live," i.e., reachable by following pointers from the stack, registers, and global variables. In the *reclamation* phase, all objects that are not live (i.e., *dead*) are freed. To find live objects in the detection phase, it must be clear what the pointers are. For instance, the collector must know what words on the stack are pointers, and not, say, integers.

QUESTION **heap vs stack**: This sentence seems to allude to an important connection between heap and stack; look into it.

Other languages address this problem by imposing a *pointer discipline*: the layouts of memory objects (including the positions of pointers) are known, and programs cannot manipulate pointers arbitrarily. For example, Java does not permit casting between integers and pointers, or direct pointer arithmetic. Along with runtime information of memory layouts, this enables precise determination of the pointer graph.

For file systems, the problem is that arbitrary pointer arithmetic is allowed, and that we do not know where pointers are stored in files. Certainly, if we restrict ourselves to components consisting of certain kinds of files, such as Java class files, we can determine at least part of the pointer graph statically, for instance by looking at the classes imported by a Java class file. However, this information is still not sufficient due to the possibility of dynamic class loading. Since the dependency information cannot be guaranteed to be complete and because this technique is not generally applicable, this approach does not suffice.

NOTE E.g., retained dependencies, or an arbitrary file named in a script written in an interpreted language (e.g., Erlang).

The solution to proper dependency identification comes from *conservative garbage collection* [15, 14]. This is a technique to provide garbage collection for languages that have pointer arithmetic and no runtime memory layout information, such as C and C++. Conservative garbage collection works by constructing a pointer graph during the detection phase by assuming that anything that looks like a valid pointer, *is* a valid pointer. For instance, if we see a bit pattern in memory corresponding to a pointer to some address p, and p is in fact part of an allocated part of memory, the object containing p is considered live. Of course, this assumption might be wrong: the bit pattern might simply encode an integer, a floating point number, and so on. But conservative garbage collection errs on the side of caution: a misclassification causes a possibly dead object to be considered live, preventing it from being reclaimed. The worst that can happen as a result is that we might run out of memory. Since misclassifications are rare, such cases are unlikely to occur[2].

SCANNING & ITS DI

Since there is a correspondence between objects in memory and files in a file system, we would like to borrow from conservative garbage collection techniques. That is, we want to borrow the concept of conservatively *scanning* for pointers from the detection phase. Specifically, we want to scan the contents of files (of which we do not know the layout), and if we see a string of characters that looks like a file name, we assume that the file being scanned has a pointer to the file indicated by the name. But there are two problems with this idea. First, there might be many false positives: the fact that the strings usr, bin and gcc occur in a component, does not necessarily imply that the component is dependent on /usr/bin/gcc). But more importantly, arbitrary pointer arithmetic makes it possible that even files that are *not* referenced, might still be dereferenced at runtime through pointers computed at runtime.

NOTE It will be about **scanning** until the end of this section.

There is a reason that conservative garbage collection works in languages such as C—namely, that they actually *do* have a minimal pointer discipline. For instance, pointer arithmetic is not allowed to increment or decrement a pointer beyond the boundaries of the object that it originally pointed to. Of course, the compiler has no way to verify the validity of such arithmetic, but programmers are well aware of the fact that a C fragment like

```
char * p = malloc(100);
...
char * q = p + 200; /* ouch! */
printf("%c", *q);
```

is quite illegal. Thus, *correct* C programs will work with a conservative garbage collector.

But even this minimal pointer discipline is absent in general component storage in the file system! Tools can perform arbitrary pointer arithmetic on paths to produce pointers to *any* file in the file system. Most tools don't, but even in a limited sense it is a problem. For instance, once the C compiler has a pointer to /usr/include, it will happily dereference that directory and produce pointers to all the files in it. In a way, how we store components in the file system in conventional deployment systems is reminiscent of programming in assembler—there is perfect freedom, and because of that freedom, all bets are off and we can make no statements about the structure of memory objects (see the bottom third of Figure 3.1).

So we have to *impose* a discipline on the storage of components. There are two aspects to this:

1
- Pointers have to be *shaped* in such a way that they are reliably recognisable. Short strings such as bin offer too much opportunity for misdetection.

2
- Components have to be separated from each other. The constituent files of multiple components should not be mixed. Otherwise a tool can "hop" from one component to another through pointer arithmetic.

[2] But note that a system using conservative garbage collection can be vulnerable to denial-of-service attacks if an attacker has an opportunity to feed it data (e.g., strings) that contains many bit patterns that might be misidentified as pointers.

FFICULTIES

The Nix store introduced in Section 2.1 satisfies both requirements. Components are stored in isolation from each other in paths such as /nix/store/bwacc7a5c5n3...-hello-2.1.1, which include long, distinctive strings such as bwacc7a5c5n3.... The latter is suitable for pointer scanning, contrary to, say, hello. Thus, if we find the ASCII string bwacc7a5c5n3... in a different component, we can conclude that there is a dependency; and if we *don't* find the string, we can conclude that no such dependency exists between the two components.

As noted in Section 3.2, the build of a component may retain a pointer to another component, thus propagating a dependency to later points on the deployment timeline. In conventional deployment systems, these dependencies have to be specified separately, with all the risks inherent in manual specification. By analysing those files of a component that are part of a distribution (in source or binary form) or of an installation, we can automatically detect timeline dependencies, such as build-time and runtime dependencies.

Pointer hiding What are the risks of this approach, i.e., when does it fail to detect dependencies? It has exactly the same limitation as conservative garbage collection in programming languages, namely, the failure to detect pointers due to *pointer hiding*. Pointer hiding occurs when a pointer is encoded in such a way that it is not recognised by the scanner (a *false negative*). For example, in C we can do the following:

```
char * p = malloc(...);
unsigned int x = ~((unsigned int) p); /* invert all bits */
p = 0;
... /* run the collector */
p = (char *) ~x;
```

That is, we allocate a pointer, then cast it to an integer, flip all the bits in the integer, and clear the original pointer. Later, we invert the integer again, and cast it back to a pointer, thus recovering the original pointer. However, if the garbage collector runs in between, it will not be able to detect that the referenced memory object if alive, since *x* does not look like a pointer to that object. Likewise, the collector can be fooled by writing a pointer to disk and reading it back later.

With components in the file system, there are similar risks[3]. If the collector searches for ASCII representations of the hash parts of component names, then anything that causes those hash parts to be not represented as contiguous ASCII strings will break the detection. Obviously, flipping the bit representation of the ASCII string will accomplish that, but this is unlikely to happen in real software. More likely is that the file name will be split in some way. Path components are a common place to split path names: /nix/store/-bwacc7a5c5n3...-hello-2.1.1 might be represented as a list ["nix", "store", "bwacc7a5c5n3...-hello-2.1.1"]. This is why we scan for just the hash part, not the whole store path. Tools are unlikely to split inside path components since substrings of path components have no general semantic meaning.

More threatening is the use of another representation than ASCII. For instance, if paths are represented in UTF-16 [34, Section 2.5], Nix's scanner will not find them. Of course,

[3] In all fairness, the risks are potentially a bit bigger in the file system than in C. The C standard leaves the semantics of the code example above implementation-defined. So most things that can cause pointer hiding are not used by educated C programmers in general. On the plus side, the chance of a *false positive* is practically non-existent.

the scanner can easily be adapted to support such encodings. Another concern that is harder to support is compressed executables. The best solution here is not to use such facilities if they are available.

Ultimately, how well the scanning approach works is an empirical question. As we shall see in Chapter 7, scanning works very well in practice, as evidenced by the large set of components in the Nix Packages collection, which have not yielded a single example of a hidden pointer. Admittedly, this result may be dependent on the characteristics of the components being deployed; namely, they are Unix components, which do not typically feature compressed executables, and store filenames almost invariably in plain ASCII.

3.5. Persistence

The technique of pointer scanning, as described in the previous section, solves the problem of reliable identification of component dependencies. Here I describe a solution to the problem of component interference, which occurs when two components occupy the same addresses in the file system. When we deploy software by copying a file system closure to another system, we run the risk of overwriting other software. The underlying problem is similar to that in the notion of *persistence* in programming languages, which is essentially the migration of data from one address space to another (such as a later invocation of a process). We cannot simply dump the data of one address space and reload them at the same addresses in the other address space, since those may already be occupied (or may be invalid). This problem is solved by *serialising* the data, that is, by writing it to disk in a format that abstracts over memory locations.

Unfortunately, it is hard to apply an analogue of serialisation to software deployment, because it requires changing pointers, which may not be reliably possible in files[4]. For instance, a tempting approach is to rename the files in a closure to addresses not existing on the target system. To do this, we also have to change the corresponding pointers in the files. However, patching files in such a manner might cause the affected components to break, e.g., due to internal checksums on files being invalidated in the process.

Thus, we should choose addresses in such a way as to minimise the chance of address collision. To make our scanning approach work, we already need long, recognisable elements in these file names, such as the base-32 representations of 160-bit values used in Nix. Not every selection of values prevents collisions: e.g., using a combination of the name and version number of a component is insufficient, since there frequently are incompatible instances even for the same version of a component.

Another approach is to select random addresses whenever we build a component. This works, but it is extremely inefficient; components that are functionally equal would obtain different addresses on every build, and therefore might be stored many times on the same system. That is, there is a complete lack of *sharing*: equal components are not stored at the same address.

Hence, we observe a tension between the desire for sharing on the one hand, and the avoidance of collisions on the other hand. The solution is another notion from memory management. *Maximal sharing* [166] is the property of a storage system that two values

[4] But if we assume that we *can* rewrite pointers, then we can do something very similar to serialisation. The implications of this assumption (which turns out to be quite reasonable) are explored in Chapter 6.

occupy the same address in memory, if and only if they are equal (under some notion of equality). This minimises memory usage in most cases.

The notion of maximal sharing is also applicable to deployment. We define two components to be equal if and only if the inputs to their builds are equal. The inputs of a build include any file system addresses passed to it, and aspects like the processor and operating system on which it is performed. We can then use a *cryptographic hash* [145] of these inputs as the recognisable part of the file name of a component. Cryptographic hashes are used because they have good collision resistance, making the chance of a collision negligible. In essence, hashes thus act as unique "product codes" for components. This is somewhat similar to global unique identifiers in COM [16], except that these apply to interfaces, not implementations, and are not computed deterministically. In summary, this approach solves the problem of component interference at local sites *and* between sites by imposing a single global address space on components. The implementation of this approach is discussed in Chapter 5.

4. The Nix Expression Language

This chapter describes the Nix expression language, the formalism used to describe components and compositions. It first motivates why a lazy functional language is an appropriate choice. It then gives a full description of the syntax and semantics of the language. Finally, it discusses some of the more interesting aspects of the implementation of the Nix expression evaluator.

4.1. Motivation

The Nix expression language is a simple, untyped, purely functional language. It is *not* a general purpose programming language. Its only purpose is to describe components and compositions. A component is created through the derivation primitive operation, which accepts all information necessary to build the component. This includes its dependencies, which are other derivations or sources.

Purely functional language A purely functional language gives us a clean component description formalism. The notion that components can be produced by *functions* that accept values for the variation points in the component, is much more flexible than, say, a language that binds such variation points using global variables.

Consider the ATerm library [166], which is a small library for term manipulation. It has several build-time options, in particular an option that specifies whether the domain feature "maximal sharing" is enabled. This setting is vitally important, as it completely changes the semantics of the library. Thus a build *with* maximal sharing cannot be substituted by a build *without* maximal sharing, and vice versa. Now suppose that we have a component consisting of two programs, foo and bar, that require the ATerm library with and without maximal sharing respectively. In a purely functional language, this is easy to model. We just make a function for the ATerm library:

```
aterm = {maximalSharing}: derivation {...};
```

We can now call this function twice with different values, in the derivations of foo and bar respectively:

```
foo = derivation { ...
  buildInputs = [ (aterm {maximalSharing = true;}) ];
};
bar = derivation { ...
  buildInputs = [ (aterm {maximalSharing = false;}) ];
};
```

Imagine that we had a formalism that used global variables to bind variation points such as maximalSharing. If the language were a Makefile-like *declarative* formalism, then it would be quite hard to implement this example. Makefiles [56] are a formalism to describe the construction of (small-grained) components. Variability in a component is typically expressed by setting variables, e.g.,

```
MAX_SHARING = 1 # or 0
CFLAGS = -DMAX_SHARING=$(MAX_SHARING)
aterm: ...
foo: aterm
bar: aterm
```

It is not clear how to build foo and bar in both variants concurrently. In Make, it is typically very hard to build a component in multiple configurations that exist side-by-side. If it is done at all, it is usually accomplished through *ad hoc* tricks such as calling Make recursively with different flags, giving the build result a different name for each call.

If the formalism were *imperative*, then it would be possible. Such a style might look like this:

```
maxSharing = true;
foo(aterm());
maxSharing = false;
bar(aterm());
```

But this has the same problems that imperative programming has in general. Since the order in which statements are executed matters, it is hard to see exactly what goes into a build function such as aterm(). Of course, in a trivial example like this, it is doable. But once we get many functions, variables, and potential control-flow paths, it may become very hard to comprehend what is happening. Note that this is the case even if variables are not global. For instance, maxSharing could be a field of the aterm object. But even then the execution order frustrates comprehension.

Interestingly, most true component-level formalisms do not specify compositions at all, just components. (With "component-level" I mean that they work at the level of large-grained components, like RPM spec files do, rather than at the level of small-grained components, like Makefiles.) An RPM spec file for instance specifies a component build action, which can include variables that can be set when the rpmbuild tool is called, but there is no way for one RPM spec file to declare that it needs the result of another RPM spec file *with certain variables set to certain values*. In any case RPM will not recursively build those dependencies, let alone with the desired variable values (since it has no way to prevent collisions between the various builds).

TODO This was 16 years ago. See what changed.

Other essentially functional build formalisms are Amake and Odin, discussed in Section 10.3.

Laziness A lazy language only evaluates values when they are needed. This is a very useful property for a component description language. Here are some concrete examples of why this is useful.

It is possible to define large sets of components concurrently (i.e., in a single file or expression), without having to build them all. For instance, the Nix expression pkgs/system-

NOTE Nix pills gets a bad press when it comes to introduce Nix in simple terms, but it is great in other aspects; in this case, to show why "laziness" is important with tons of examples.

/all-packages-generic.nix in the Nix Packages collection returns a large attribute set of derivations:

```
rec {
  stdenv = derivation { ... };
  glibc = derivation { ... };
  gcc = derivation { ... };
  hello = derivation { ... };
  subversion derivation { ... };
  firefox = derivation { ... };
}
```

and hundreds more. Since the language is lazy, the right-hand sides of the attributes will not be evaluated until they are actually needed, if at all. For instance, if we install any specific component from this set, say, nix-env -i hello, then only the hello value will be evaluated (although it may in turn require other values to be evaluated, such as stdenv).

Laziness also allows greater efficiency if only parts of a complex data structure are needed. Take the operation nix-env -qa, which prints the name attribute of all top-level derivations in a Nix expression:

```
$ nix-env -f ./pkgs/system/i686-linux.nix -qa
a52dec-0.7.4
acrobat-reader-7.0
alsa-lib-1.0.9
ant-blackdown-1.4.2
ant-j2sdk-1.4.2
ant-j2sdk-1.5.0
...
```

NOTE/TODO .. and it is established: Nix expressions that evaluate to derivations are consistently called "derivations". Which introduces ambiguity as the purpose of the Nix expression language is to produce derivations...

What about calling helper modules, snippets that **do not** produce derivations by themselves "Nix expressions" and ones that do, "Nix derivation expressions". Leave the term "derivation" mean the drv files themselves.

(But then again, the drv files are ATerm expressions, and there is also the pretty-printed version that is a representation of a derivation using the Nix expression language...)

This operation is quite fast: it takes around 0.6 seconds on an Athlon XP 2600 on an expression containing 417 top-level derivations. But the derivation calls for those 417 derivations potentially need to do a lot of work, as we will see in Section 5.4: they have to copy sources to the Nix store, compute cryptographic hashes, and so on. But that work is actually attached to two attributes *returned* from derivation, namely drvPath and outPath, which compute the derivation and output store paths. As long as those attributes are not used, they are not computed. Above, this was the case, since nix-env -qa only uses the name attribute. But if we force the computation of either of those two attributes:

```
$ nix-env -f ./pkgs/system/i686-linux.nix -qa --drv-path
a52dec-0.7.4       /nix/store/spkk...-a52dec-0.7.4.drv
acrobat-reader-7.0 /nix/store/1fdl...-acrobat-reader-7.0.drv
alsa-lib-1.0.9     /nix/store/5a7h...-alsa-lib-1.0.9.drv
...
```

then the operation takes much longer: 2.7 seconds.

Laziness allows arguments to be passed to functions that may not be used. In the Subversion example (Figure 2.9), the Subversion function passes dependencies such as openssl and httpd that are only passed to the derivation if the corresponding domain feature is set, e.g.,

WHERE ARE THESE ATTRIBUTES (drvPath, outPath, name) TAKEN FROM?

QUESTION (purple wiggly line) "**attributes returned from derivation**" - starting to have an epiphany: *derivation* (by itself, mkDerivation, etc.) will produce a **Nix expression derivation** (a derivation expressed with the Nix expression language) that will get store in **drv** files (using ATerms, and it can always converted back using *show-derivation*; see notes at figure 2.13).

There is a slight problem here: *nix show-derivation* on a drv does not have **drvPath** or **outPath** attributes... So where are these being fetched from? The text mentions "derivation" but it nothing of it is mentioned in the drv representation then it must be in the Nix expressions themselves **before** the evaluation gets there. (Therefore "derivation" here may refer to an attribute of the Nix expression in Nixpkgs - or local files - that **will** eventually evaluate to a derivation ... and that brings us back to the topic of how detrimental ambiguous terminology can become; see rant above regarding **Nix derivation expression =/= derivation**).

QUESTION **drvPath** and **outPath**
These seemed to be defined on page 101 at 5.4. Translating Nix expressions to store derivations, but, confusingly, they are used here and a bit later when defining the formal principles behind Nix. Even at section 5.4, I couldn't figure out where these come from at first glance...

```
{ ..., sslSupport, openssl, ... }:
derivation {
  openssl = if sslSupport then openssl else null;
}
```

That is, the derivation's openssl attribute is set to null if SSL support is disabled, regardless of the value of the openssl function parameter. This allows us to unconditionally pass an instance of openssl in the function call in all-packages-generic.nix:

```
subversion = import .../subversion {
  inherit openssl;
  sslSupport = false; # set to 'true' if you want SSL support
};
```

Thanks to laziness, OpenSSL will only be built if SSL support is enabled, despite the fact that it is always passed as an argument. This simplifies the call sites. Without laziness, the caller would have the responsibility to ensure that no unnecessary dependencies are passed to the derivation, thus breaking abstraction.

NOTE **Nix does not use SDF anymore.** "src/libexpr/nix.sdf" has been remove in commit https://github.com/NixOS/nix/commit/62ad3dfc4335e6b3c8acff5e7de9b8cf19c9cdbe
Dolstra's reasoning is in **PR#1102**
BNFC-based parser; see comment at **https://github.com/NixOS/nix/issues/1102#issuecomment-259645906**
Couple comments below there is the actual commit from 2004: "**Replaced the SDF parser by a substantially faster Bison/Flex**" **https://github.com/NixOS/nix/commit/c5baaafae69394082817ede9e6eb3910c4601a72**

4.2. Syntax

This section presents the syntax of the Nix expression language using the Syntax Definition Formalism [175, 90], which has the following useful properties:

NOTE The formal syntax is described in **src/libexpr/lexer.l** (https://stackoverflow.com/questions/56198420/what-is-the-syntax-of-a-valid-identifier-in-the-nix-language)

See note for reasons and commits

NOTE Look at the top of the next page; so Nix **does not** use SDF, but because SDF beneficial properties, it is used in this paper to describe the Nix expression

- It is *scannerless*: the lexical syntax and the high-level "context-free syntax" are specified in the same formalism. Thus, no separate lexical scanner is necessary. The result is a single context-free grammar.
- It supports *generalised LR (GLR) parsing*. Most context-free parsers (e.g., $LL(k)$ and $LR(k)$ parsers, for fixed k) suffer from bounded look-ahead: they must choose between different productions on the basis of a fixed number of tokens. This is inconvenient for language implementors, since they must deal with the resulting parse *conflicts*. For instance, the input fragment "{x" could be start of an attribute set (e.g., {x=123;}), or a function definition (e.g., {x}: x)[1]. This requires the programmer to introduce additional production rules to resolve the conflict. Since generalised LR parsing has unbounded look-ahead, such grammar transformations are unnecessary. Another advantage of GLR is that since it supports arbitrary context-free grammars (including ambiguous ones), it allows grammars to be composed. This makes it easy to extend programming languages [19].
- It is a declarative formalism that specifies a grammar separate from any particular programming language (contrary to, say, Yacc [103]). This is the main reason why I use SDF in this chapter: it allows the entire language to be described in a concise, readable and directly usable formalism, with little meta-syntactic overhead.

TODO: Again, SDF is not used by Nix, but this paper **does** explain the Nix expression language using SDF. So it's still worth understanding it.

Not sure why the whole section wasn't started with this information though...

[1] Actually, this is the *only* shift/reduce conflict in the language, so generalised LR parsing is not that important here, except that it enables scannerless parsing.

The current Nix implementation unfortunately does not use the SDF parser (sglr) for performance reasons (scannerless parsing in particular is expensive). Rather, it uses Bison [66], which is the GNU implementation of Yacc [103]. It recently added support for GLR parsing, but still requires a separate lexical scanner. The Flex lexical scanner generator [130], an implementation of Lex [113], is used for lexical analysis.

The following is a brief overview of SDF necessary to understand the Nix grammar. A full specification of SDF can be found in [175]. An SDF grammar contains *production rules* of the form

$$symbols \rightarrow nt$$

where *symbols* is a sequence of terminals and non-terminals, and *nt* is the non-terminal produced by the production rule. This notation is somewhat untraditional: most context-free grammar formalisms are written the other way around ($nt \leftarrow symbols$ or $nt \rightarrow symbols$). The left-hand side can contain various types of syntactic sugar:

- Regular expressions to denote lexical elements.
- The usual EBNF constructs: repetition (S*), choice (S?), and grouping ((S)).

SDF has two kinds of productions: those defining the *lexical syntax*, and those defining the *context-free syntax*. Lexical syntax is used to define the tokens of the language. An example of the first is the definition of the non-terminal for identifiers:

```
lexical syntax
  [a-zA-Z\_][a-zA-Z0-9\_\']* -> Id
```

This defines identifiers using a regular expression: they start with a letter or underscore, and are followed by zero or more letters, digits, underscores, and accents.

Layout—whitespace and comments that can occur anywhere between tokens—is also defined using lexical syntax productions. For example, the production

```
  [\ \t\n] -> LAYOUT
```

says that spaces, tabs and newlines are all layout. There is a built-in production that says that sequences of layout are also layout. The non-terminal LAYOUT has a special status, as it is automatically inserted between all symbols in the productions of the context-free syntax.

For example, the following context-free syntax[2] rule defines a production for a simple expression language:

```
context-free syntax
  Expr "+" Expr -> Expr
```

This rule is (essentially) desugared to

```
  LAYOUT? Expr LAYOUT? "+" LAYOUT? Expr LAYOUT? -> Expr
```

[2] The term “context-free syntax” in SDF is a bit confusing, as both its “lexical syntax” and “context-free syntax” are combined to a single context-free grammar.

QUESTION **lexical syntax** vs **context-free syntax**

SDF allows priorities and associativities to be defined to disambiguate otherwise ambiguous grammars. The following example defines left-associative productions for addition and multiplication in a simple expression language, and then defines that multiplication takes precedence over addition:

```
context-free syntax
  Expr "+" Expr -> Expr {left}
  Expr "*" Expr -> Expr {left}
context-free priorities
  Expr "*" Expr -> Expr
> Expr "+" Expr -> Expr
```

Finally, there are typically all sorts of ambiguities between the lexical elements of a language. For instance, the string if can be parsed as the identifier "if", a sequence of identifiers i and f, and the keyword if (assuming that the language has a production containing such a keyword). The SDF features of *rejects* and *follow restrictions* enforce the correct parsing. This declaration says that if should not be parsed as an identifier:

```
lexical syntax
  "if" -> Id {reject}
```

Technically, this defines a production that whenever it matches at some point with the input, it causes any other Id production for those same characters to be rejected. To prevent a sequence of identifier characters from being interpreted as a sequence of identifiers, a *follow restriction* can be used:

```
lexical restrictions
  Id -/- [a-zA-Z0-9\_\’]
```

This says that an Id cannot be followed by a character matching the given character class. Thus, xy cannot be parsed as the identifiers x and y, since x is followed by a letter.

TODO How much has changed?

4.2.1. Lexical syntax

We start with the lexical syntax of Nix expressions. It is divided into two parts: the definition of the layout, and of the actual information-carrying tokens of the language.

Figure 4.1 shows the SDF *module* that defines the layout of the language. (SDF grammars are modular in order to allow the different aspects of a language to be defined separately, and to make it easier to combine grammars.) Layout consists of whitespace (spaces, tabs and newlines) and comments. It is worth noting that the language supports two styles of comments:

```
(x: x) # Single-line comments like this
"foo" + /* Multi-line
  comments like this */ "bar"
```

Figure 4.2 defines the remainder of the lexical syntax of the language. The token types defined here are identifiers (which are exactly as described in the Id example above), and various kinds of constants defined using regular expressions. These are:

- Natural numbers (Int).

```
module Nix-Layout
exports
  sorts HashComment Asterisk Comment
  lexical syntax
    [\ \t\n]     -> LAYOUT
    HashComment -> LAYOUT
    Comment      -> LAYOUT
    "#" ~[\n]* -> HashComment
    "/*" ( ~[\*] | Asterisk )* "*/" -> Comment
    [\*] -> Asterisk
  lexical restrictions
    Asterisk -/- [\/]
    HashComment -/- ~[\n]
  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

Figure 4.1.: Layout in Nix expressions

- Strings (Str) enclosed between double quotes. Characters are escaped by prefixing them with a backslash.

- Paths (Path) are essentially sequences of characters with at least one forward slash in them. To be precise, they are sequences of *path components* separated by slashes. A path component consists of one or more letters, digits, and some other characters[3]. The initial path component, i.e., the one before the first slash, may be omitted. If it is omitted, we have an *absolute path*, e.g., /home/alice/.profile. If it is present, we have a *relative path*, e.g., ../../foo/builder.sh.

- URIs (Uri) are a convenience for the specification of URIs in, e.g., calls to the function fetchurl. The advantage of having URIs as a language construct over simply using strings is that the user gets syntax checking, and can omit the enclosing quotes. The regular expression for URIs used here follows [12, Appendix A].

The remainder of Figure 4.2 is concerned with rejects and follow restrictions. Note that the keywords of the language (used in the context-free syntax below) must be given twice, first to reject them as identifiers, and second to prevent them from being followed by identifier characters (e.g., ifx should not be parsed as the keyword if followed by the variable x). That we have to specify keywords twice is a current infelicity of SDF.

Also note that *Booleans* (true and false) are not specially defined here. Rather, true and false are parsed as identifiers and interpreted as built-in nullary functions during execution, as discussed in the semantics below.

4.2.2. Context-free syntax

Figure 4.3 shows the context-free syntax of the language. A parse of a Nix expression must match an Expr non-terminal $\boxed{34}$. Almost all productions here produce Exprs. In that

[3] So currently there is no way to define paths that contain characters not in this set.

```
module Nix-Lexicals
exports
  sorts Id Int Str Path Uri
  lexical syntax
    [a-zA-Z\_][a-zA-Z0-9\_\']* -> Id
    "rec" | "let" | "if" | "then" | "else" |
    "assert" | "with" | "inherit" -> Id {reject}

    [0-9]+ -> Int

    "\"" ~[\n\"]* "\"" -> Str

    [a-zA-Z0-9\.\_\-\+]* ("/"[a-zA-Z0-9\.\_\-\+]+)+ -> Path

    [a-zA-Z] [a-zA-Z0-9\+\-\.]* ":"
      [a-zA-Z0-9\%\/\?\:\@\&\=\+\$\,\-\_\.\!\~\*\']*
      -> Uri

  lexical restrictions
    Id -/- [a-zA-Z0-9\_\']
    Int -/- [0-9]
    Path -/- [a-zA-Z0-9\.\_\-\+\/]
    Uri -/- [a-zA-Z0-9\%\/\?\:\@\&\=\+\$\,\-\_\.\!\~\*\']
    "rec" "let" "if" "then" "else"
    "assert" "with" "inherit" -/- [A-Za-z0-9\_\']
```

Figure 4.2.: Lexical syntax of Nix expressions

respect the Nix expression grammar is quite flat: for instance, there is no separate grammatical level for concepts such as modules, function definitions, and so on. Everything is an expression.

The first production just injects the non-terminals for constants into Expr $\boxed{35}$. Of course, expressions can be enclosed in parentheses to override the normal precedence rules $\boxed{36}$.

The language has two types of functions. The first $\boxed{37}$ takes a single argument. For instance, the (anonymous) identity function can be defined as follows:

```
x: x
```

Of course, this style of function is just a plain λ-abstraction from the λ-calculus [10], as we will see in the discussion of the semantics below. Though this style only allows functions with a single argument, since this is a functional language we can still define (in a sense) functions with multiple arguments, e.g.,

```
x: y: x + y
```

which is a function taking an argument x that returns *another function* that accepts an argument y.

The second style of function definition $\boxed{38}$, which we have seen in Section 2.2, is more important in this language. It takes an attribute set and binds the attributes defined therein to local variables. Thus,

```
module Nix-Exprs
imports Nix-Lexicals
exports
  sorts Expr Formal Bind ExprList
  context-free start-symbols Expr 34
  context-free syntax

    Id | Int | Str | Uri | Path -> Expr 35

    "(" Expr ")" -> Expr 36

    Id ":" Expr -> Expr 37

    "{" {Formal ","}* "}" ":" Expr -> Expr 38
    Id          -> Formal
    Id "?" Expr -> Formal

    Expr Expr -> Expr 39

    "assert" Expr ";" Expr -> Expr

    "with" Expr ";" Expr -> Expr 40

    "{" Bind* "}" -> Expr 41
    "rec" "{" Bind* "}" -> Expr
    "let" "{" Bind* "}" -> Expr

    Expr "." Id -> Expr

    Id "=" Expr ";"                  -> Bind 42
    "inherit" ("(" Expr ")")? Id* ";" -> Bind

    "[" ExprList "]" -> Expr 43
    "" -> ExprList
    Expr ExprList -> ExprList

    "if" Expr "then" Expr "else" Expr -> Expr 44
```

Figure 4.3.: Context-free syntax of Nix expressions

```
{x, y}: x + y
```

declares a function that accepts an attribute set with attributes x and y (and nothing else), and the expression

```
({x, y}: x + y) {y = "bar"; x = "foo";}
```

yields "foobar". The formal (expected) argument can have a default value, allowing it to be omitted. So

```
({x, y ? "bar"}: x + y) {x = "foo";}
```

also evaluates to "foobar".

Following the tradition of most functional languages, function calls are written by juxtaposition, i.e., f x instead of, say, f(x). Whether the absence of an argument delimeter makes function calls more or less readable is a matter of controversy. However, most calls in Nix expressions pass argument sets, which are enclosed in curly braces and therefore delimited anyway (i.e., f {*attrs*}).

Assertions have been discussed in Section 2.2 (page 33). *With expressions* $\boxed{40}$ allow *all attributes* in an attribute set to be added to the lexical scope. For example,

```
with {y = "bar"; x = "foo";}; x + y
```

evaluates to "foobar"; the variables x and y defined in the attribute set are in scope in the expression x + y. This construct is primarily useful in conjunction with import, allowing values to be "included" from an attribute set defined in a file:

```
with (import ./definitions.nix); x + y
```

In essence, this enables a "conventional" module system: commonly used definitions can be placed in a separate file, and imported into the lexical scope of an expression in another file. But note that contrary to most module systems, with and import enable a "policy-free" module system: for instance, imports can occur in any expression, not just at top level.

The various types of attribute set definitions $\boxed{41}$—plain, recursive, and lets—are discussed in more detail below. All three contain lists of attributes enclosed in curly braces. There are two kinds of attribute definitions $\boxed{42}$: plain name/value pairs, and inherits, which copy the value of an attribute from the surrounding lexical scope or an optional expression.

Lists $\boxed{43}$ are enclosed in square brackets, and the list elements are juxtaposed, *not* separated by an explicit delimiter. For instance,

```
[1 2 3]
```

is a list of three elements. More importantly,

```
[a b c]
```

is also a list of three elements, *not* a call to function a with arguments b and c[4].

Finally, the language has conditionals $\boxed{44}$. There is only an if...then...else construct and no if...then, since the latter does not make sense in a functional setting as it leaves the value of the expression undefined in case of a false condition. Assertions already serve this

```
    Expr "==" Expr -> Expr {non-assoc}
    Expr "!=" Expr -> Expr {non-assoc}

    "!" Expr -> Expr
    Expr "&&" Expr -> Expr {right}
    Expr "||" Expr -> Expr {right}
    Expr "->" Expr -> Expr {right}

    Expr "//" Expr -> Expr {right}
    Expr "~" Expr -> Expr {non-assoc}
    Expr "?" Id -> Expr
    Expr "+" Expr -> Expr {left}
```

Figure 4.4.: Context-free syntax of Nix expressions: Operators

purpose—they abort evaluation if the condition is false. Note that due to the absence of an if...then, there is no “dangling else” problem.

Figure 4.4 continues the context-free syntax. It defines operators, along with their associativities. The meaning of the operators is given below.

Figure 4.5 defines the relative precedences of all language constructs. Functions bind the weakest, and thus the body of a function extends maximally to the right unless the function is enclosed in parentheses. Assertions, with-expressions and conditions are the next level, followed by the operators. Function application binds very strongly. These priorities cause the following example expression:

```
{x}: assert true; f x != !g y
```

to be parsed as:

```
({x}: (assert (true); ((f x) != (!(g y)))))
```

4.3. Semantics

This section gives the formal semantics of the Nix expression language.

4.3.1. Basic values

The basic (data) values of the Nix expression language, in addition to natural numbers, strings, paths, and URIs described above, are the following:

- Null values, denoted as the built-in nullary function null.
- Booleans, denoted as the built-in nullary functions true and false.

[4] The reason that there is a ExprList non-terminal is actually for this very reason: we cannot write Expr* in the production rule for lists, since then we are not able to give it a priority relative to function calls.

```
context-free priorities
  Expr "." Id -> Expr
> Expr ExprList -> ExprList
> Expr Expr -> Expr
> Expr "~" Expr -> Expr
> Expr "?" Id -> Expr
> Expr "+" Expr -> Expr
> "!" Expr -> Expr
> Expr "//" Expr -> Expr
> Expr "==" Expr -> Expr
> Expr "!=" Expr -> Expr
> Expr "&&" Expr -> Expr
> Expr "||" Expr -> Expr
> Expr "->" Expr -> Expr
> "if" Expr "then" Expr "else" Expr -> Expr
> "assert" Expr ";" Expr -> Expr
> "with" Expr ";" Expr -> Expr
> Id ":" Expr -> Expr
> "{" {Formal ","}* "}" ":" Expr -> Expr
```

Figure 4.5.: Context-free syntax of Nix expressions: Priorities

NOTE the ~ operator has been deprecated: https://nixos.org/manual/nix/stable/release-notes/rl-0.10.html?highlight=subpath#release-010-2006-10-06

The + operator can now fulfil this function too.

- *Subpaths* are a somewhat obscure language feature that allows files in derivations to be referenced from other derivations. This is useful if a derivation (as is typical) produces a directory tree, and we are interested in a particular file in that tree. Suppose that we have a variable perl that refers to a derivation that builds Perl. The output of this derivation is a directory at some path p. Suppose now that we want to use the Perl interpreter as the builder of some other derivation. However, the interpreter is a program stored not at p but at $p+$"/bin/perl". With subpaths, we can write the following:

NOTE Again, "derivation" here refers to the derivation represented by Nix terms.

```
    derivation { ...
      builder = perl + /bin/perl;
    }
```

also

The operator + is the constructor of subpath values. We need subpaths in order to maintain the proper dependency relation between derivations. The above might have been written as:

TODO yet again, no clue where **outPath** and **outDrv** come from

```
    derivation { ...
      builder = perl.outPath + "/bin/perl";
    }
```

That is, we compute the path of the builder through ordinary string concatenation (the outPath attribute contains the computed store path of the perl derivation). As we shall see in Section 5.4, this derivation will not properly identify perl as one of its dependencies. It will just have an opaque string value for its builder attribute[5].

[5] Currently, direct references to outPath are not disallowed, so this unsafe example is actually possible. It can be disallowed by restricting access to the outPath and drvPath attributes described in Section 5.4.

TODO What? Has this been resolved since then?

All path values are internally in *canonical form*, meaning that they are not relative to the current directory (i.e., start with /), do not contain . or .. elements, do not contain redundant separators (e.g., //), and do not end in a separator. Any relative paths in a Nix expression are *absolutised* relative to the directory that contained the expression. For instance, if the expression /foo/bar/expr.nix contains a path ../bla/builder.sh, it is absolutised to /foo/bla/builder.sh. Sometimes Nix expressions are not specified in a file but given on the command line or passed through standard input, e.g., in nix-env and nix-instantiate. Such paths are absolutised relative to the current directory. In the semantic rules below we will make use of a function canonicalise(p,d) (whose definition is omitted on grounds of dullness) that yields a canonical representation of path p, absolutised relative to directory d if necessary.

4.3.2. Compound values

The language has two ways to form compound data structures: lists and attribute sets.

Lists Lists are enclosed in square brackets, as described above. List elements are lazy, so they are only evaluated when needed. Currently, the language has very limited facilities for list manipulation. There is a built-in function map that applies a function to all elements in a list. Lists can be included in derivations, as we will see in Section 5.4. Other than that, there are no operations on lists.

Attribute sets The most important data type in the language is the attribute set, which is a set of name/value pairs, e.g.,

```
{ x = "foo"; y = 123; }
```

Attribute names are identifiers, and attribute values are arbitrary expressions. The order of attributes is irrelevant, but any atttribute name can occur only once in a set. Attributes can be selected using the . operator:

```
{ x = "foo"; y = 123; }.y
```

This expression evaluates to 123.

Recursive attribute sets allow attribute values to refer to each other. They are constructed using the rec keyword. Formally, each attribute in the set is added to the scope of the entire attribute set. Hence,

```
rec { x = y; y = 123; }.x
```

evaluates to 123. If rec were omitted, the identifier y in the definition of the attribute x would refer to some y bound in the surrounding scope. Recursive attribute sets introduce the possibility of recursion, including non-termination:

```
rec { x = x; }.x
```

~~A *let-expression*, constructed using the let keyword, is syntactic sugar for a recursive attribute set that automatically selects the special attribute body from the set. Thus,~~

TODO These are not a thing anymore, but check when it was removed

```
~~let { body = a ++ b; a = "foo"; b = "bar"; }~~
```

~~evaluates to "foobar".~~

As we saw above, when defining an attribute set, attributes values can be *inherited* from the surrounding lexical scope or from other attribute sets. The expression

```
x: { inherit x; y = 123; }
```

defines a function that returns an attribute set with two attributes: x which is inherited from the function argument named x, and y which is declared normally. The inherit construct is just syntactic sugar. The example above could also be written as

```
x: { x = x; y = 123; }
```

Note that the right-hand side of the attribute x = x refers to the function argument x, *not* to the attribute x. Thus, x = x is not a recursive definition.

Likewise, attributes can be inherited from other attribute sets:

```
rec {
  as1 = {x = 1; y = 2; z = 3;};
  as2 = {inherit (as1) x y; z = 4;};
}
```

Here the set as2 copies attributes x and y from set as1. It desugars to

```
rec {
  as1 = {x = 1; y = 2; z = 3;};
  as2 = {x = as1.x; y = as1.y; z = 4;};
}
```

However, the situation is a bit more complicated for rec attribute sets, whose attributes are mutually recursive, i.e., are in each other's scope. The intended semantics of inherit is still the same: values are inherited from the surrounding scope. But simply desugaring is no longer enough. So if we desugar

```
x: rec { inherit x; y = 123; }
```

to

```
x: rec { x = x; y = 123; }
```

we have incorrectly created an infinite recursion: the attribute x now evaluates to itself, rather than the value of the function argument x. For this reason rec is actually internally stored as *two* sets of attributes: the recursive attributes, and the non-recursive attributes. The latter are simply the inherited attributes. We denote this internally used representation of rec sets as rec $\{as_1/as_2\}$, where as_1 and as_2 are the recursive and non-recursive attributes, respectively.

$$
\begin{aligned}
\mathsf{subst}(subs, x) &= \begin{cases} e & \text{if } (x \rightsquigarrow e) \in subs \\ x & \text{otherwise} \end{cases} \\
\mathsf{subst}(subs, \{as\}) &= \{\mathsf{map}(\lambda \langle n = e\rangle . \langle n = \mathsf{subst}(subs, e)\rangle, as)\} \\
\mathsf{subst}(subs, \mathsf{rec}\ \{as_1 / as_2\}) &= \mathsf{rec}\ \{\ \mathsf{map}(\lambda \langle n = e\rangle . \langle n = \mathsf{subst}(subs', e)\rangle, as_1) \\
&\qquad / \ \mathsf{map}(\lambda \langle n = e\rangle . \langle n = \mathsf{subst}(subs, e)\rangle, as_2)\} \\
&\text{where } subs' = \{x \rightsquigarrow e | x \rightsquigarrow e \in subs \wedge x \notin \mathsf{names}(as)\} \\
\mathsf{subst}(subs, \mathsf{let}\ \{as_1 / as_2\}) &= \text{analogous to the rec case} \\
\mathsf{subst}(subs, x\text{: } e) &= x\text{: } \mathsf{subst}(subs', e) \\
&\text{where } subs' = \{x_2 \rightsquigarrow e | x_2 \rightsquigarrow e \in subs \wedge x \neq x_2\} \\
\mathsf{subst}(subs, \{fs\}\text{: } e) &= \{fs\}\text{: } \mathsf{subst}(subs', e) \\
&\text{where } subs' = \{x \rightsquigarrow e | x \rightsquigarrow e \in subs \wedge x \notin \mathsf{argNames}(fs)\}
\end{aligned}
$$

Figure 4.6.: subst: Substitutions

4.3.3. Substitutions

Variable substitution is an important operation in the evaluation process[6]. The substitution function $\mathsf{subst}(subs, e)$ performs a set of substitutions $subs$ in the expression e. The set $subs$ consists of substitutions of the form $x \rightsquigarrow e$ that replace a variable x with an expression e.

x⇝e

Here and in the semantic rules below, the variable e ranges over expressions, x over variables, p over paths, s over strings, and n over attribute names, with subscripts as appropriate. The members of attribute sets are denoted collectively as as, and we write $\langle n = e\rangle \in as$ to denote that the attribute set as has an attribute named n with value e. The arguments of a multi-argument function are denoted as fs (for "formals").

The auxiliary function $\mathsf{names}(as)$ yields the set of attribute names occurring in the left hand side of a set of attributes as. Likewise, $\mathsf{argNames}(fs)$ yields the set of *names* of formal arguments from a set of formal arguments (note that formal arguments can also contain defaults, which are left out by this function).

The function subst replaces all *free variables* for which there is a substitution. A variable is free in a subexpression if it is not *bound* by any of its enclosing expressions. Variables are bound in functions and in recursive attribute sets. In recursive attribute sets, only the *recursive* attributes (as_1) bind variables; the non-recursive attributes (as_2) do not. The function subst is shown in Figure 4.6. For all cases not specifically mentioned here, the substitution is recursively applied to all subexpressions. For instance, for function calls the following substitution is applied:

$$\mathsf{subst}(subs, e_1\ e_2) = \mathsf{subst}(subs, e_1)\ \ \mathsf{subst}(subs, e_1)$$

!

[6] The term *substitution* in this section has no relation to Nix's substitute mechanism.

QUESTION: what is **name capture**? Below the term "name capture" is linked to [10] but Barendregt's tome has 1 entry for "captur" search term and it isn't even related.

It is assumed that the substitution terms (i.e., the expressions in *subs*) contain no free variables, so subst does not have to perform renaming to prevent name capture; the style of evaluation used below allows us to get away with this.

https://stackoverflow.com/questions/11239262/what-is-meant-by-capture-avoiding-substitutions refers to the same thing as "*capture-avoiding substitutions*" and I think I get the gist:
alpha equivalence: (λx.(λy.yx)) is equivalent to (λa.(λb.ba))
but this is not: (λy.yx) =/= (λx.xx)
So I guess if during the substitution rules a variable is placed in an expr with the same name as a parameter then it will be a "collision" or that variable will be "captured".

4.3.4. Evaluation rules

The operational semantics of the language is specified using semantic rules of the form $e_1 \mapsto e_2$ that transform expression e_1 into e_2. Rules may only be applied to closed terms, i.e., terms that have no free variables. Thus it is not allowed to arbitrary apply rules to subterms.

An expression e is in *normal form* if no rules are applicable. Not all normal forms are acceptable evaluation results. For example, no rule applies to the following expressions:

```
x
123 x
assert false; 123
{x = 123;}.y
({x}: x) {y = 123;}
```

good(e) == false

The predicate *good*(e) defines whether an expression is a valid evaluation result. It is true if e is a basic or compound value or a function (lambda), and false otherwise. Since rules are only allowed to be applied to an expression at top level (i.e., not to subexpressions), a good normal form is in weak head normal form (WHNF) [133, Section 11.3.1]. Weak head normal form differs from the notion of *head normal form* in that right-hand sides of functions need not be normalised. A nice property of this style of evaluation is that there can be no *name capture* [10], which simplifies the evaluation machinery.

An expression e_1 is said to *evaluate* to e_2, notation $e_1 \overset{*}{\mapsto} e_2$, if there exists a sequence of zero or more applications of semantic rules to e_1 that transform it into e_2 such that *good*(e_2) is true; i.e., the normal form must be good. In the implementation, if the normal form of e_1 is not good, its evaluation triggers a runtime error (e.g., "undefined variable" or "assertion failed").

Not all expressions have a normal form. For instance, the expression

```
(rec {x = x;}).x
```

does not terminate. But if evaluation does terminate, there must be a single normal form. That is, evaluation is *confluent* [5]. The confluence property follows from the fact that at most one rule applies to any expression. The implementation detects some types of infinite recursion, as discussed below.

The semantic rules are stated below in the following general form:

$$\textsc{Rule-Name} : \frac{\textit{condition}}{e \mapsto e'}$$

NOTE Why come up with a notation if it is not used correctly...

(or, if the formula is wrong, then a dot is missing from the arrows)

That is, we can conclude that e transforms into ~~evaluates to~~ e' ($e \mapsto e'$), if the proposition *condition* holds. If there are no conditions, the rule is simply written as $\textsc{Rule-Name} : e \mapsto e'$.

Attribute sets The SELECT rule implements attribute selection. This rule governs *successful* selection, i.e., it applies only if the given attribute name exists in the attribute set.

$$\textsc{Select}: \frac{e \overset{*}{\mapsto} \{as\} \wedge \langle n = e' \rangle \in as}{e.n \mapsto e'}$$

Note that there is no rule for failure. If attribute n is not in as, evaluation fails and a nice error message is printed in the actual implementation.

A recursive attribute set is desugared to a normal attribute set by replacing all occurrences of references to the attributes with the recursive attribute set. For instance, if $e =$ rec {x = f x y; y = x;}, then e is desugared to

```
{ x = f (e.x) (e.y);
  y = e.x;
}
```

or in full,

NOTE This is super-important to understand overrides (among others)

TODO As an example of how recursive attribute sets evaluate, it's perfect. But can't figure out what would happen if I actually typed it in the nix repl. Looks mind-breakingly similar to the override stuff in Nix pills so play with this, just out of curiosity.

```
{ x = f ((rec {x = f x y; y = x;}).x)
        ((rec {x = f x y; y = x;}).y);
  y = (rec {x = f x y; y = x;}).x;
}
```

This desugaring is implemented by the REC rule:

$$\textsc{Rec}: \mathsf{rec}\ \{as_1/as_2\} \mapsto \{\mathsf{subst}(subs, \{as_1\}) \cup as_2\}$$

where

$$subs = \{n \rightsquigarrow (\mathsf{rec}\ \{as_1/as_2\}).\mathsf{n} \mid n \in \mathsf{names}(as_1 \cup as_2)\}$$

Contrary to what the reader might expect, this substitution does not lead to a potential explosion in the size of expressions in our implementation, since we use ATerms that employ maximal sharing to store equal subterms exactly once, as discussed in Section 4.4.

~~A let-expression is just syntactic sugar that automatically selects the attribute body from a recursive set of attributes:~~

~~$\textsc{Let}: \mathsf{let}\ \{as_1/as_2\} \mapsto (\mathsf{rec}\ \{as_1/as_2\}).\mathsf{body}$~~

TODO That is, I'm not sure what the current rules are or how **let-expressions** are implemented, but this is not it.

Function calls Function calls to single-argument functions (i.e., lambdas) are just plain β-reduction in the λ-calculus [10].

$$\beta\text{-}\textsc{Reduce}: \frac{e_1 \overset{*}{\mapsto} \boxed{x\colon e_3}\ \text{function } \textbf{definition}}{\underset{\text{function } \textbf{call}}{\boxed{e_1\ e_2}} \mapsto \mathsf{subst}(\{x \rightsquigarrow e_2\}, e_3)}$$

single-argument function calls

As we shall see in Theorem 2 (page 85), the expression e_2 contains no free variables. Therefore, there is no danger of name capture in subst.

QUESTION: what is **name capture**?

Multi-argument function calls, i.e., functions that accept and open an attribute set, are a bit more complicated:

$$\beta\text{-}\textsc{Reduce'}: \frac{e_1 \overset{*}{\mapsto} \{fs\}\colon e_3 \ \wedge\ e_2 \overset{*}{\mapsto} \{as\} \ \wedge\ match}{e_1\ e_2 \mapsto \mathsf{subst}(subs, e_3)}$$

multi-argument function calls

where

$$\begin{aligned} \mathit{match} &= (\forall n \in \mathsf{names}(as) : n \in \mathsf{argNames}(fs)) \;\wedge\; (\forall n \in fs : n \in \mathsf{names}(as)) \\ \mathit{subs} &= \{n \rightsquigarrow \mathsf{arg}(n) \mid n \in \mathsf{argNames}(fs)\} \\ \mathsf{arg}(n) &= \begin{cases} e & \text{if } \langle n = e\rangle \in as \\ e & \text{if } n \notin \mathsf{names}(as) \;\wedge\; n \mathbin{?} e \in fs \end{cases} \end{aligned}$$

TODO Get to the bottom of this; I think I understand it on an intuitive level but it is not clear.

Note that a multi-argument function call is strict in its argument—the attribute set—but not in the values of the attributes. The Boolean value *match* ascertains whether each actual argument matches a formal argument, and whether each formal argument *without a default* matches an actual argument (note that $\forall n \in fs$ does not apply to the formal arguments with defaults, which would be $\forall n \mathbin{?} e \in fs$). The function $\mathsf{arg}(n)$ determines the actual value to use for formal argument n, taking it from as if it has an attribute n, and using the default in fs otherwise.

Conditionals Conditional expressions first evaluate the condition expression. It must evaluate to a Boolean. (Evaluation fails if it is not.) The conditional then evaluates to one of its alternatives.

$$\textsc{IfThen}: \frac{e_1 \overset{*}{\mapsto} \mathsf{true}}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mapsto e_2} \qquad \textsc{IfElse}: \frac{e_1 \overset{*}{\mapsto} \mathsf{false}}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mapsto e_3}$$

Assertions An assertion $\mathsf{assert}\ e_1;\ e_2$ evaluates to e_2 if e_1 evaluates to true. Otherwise, evaluation fails.

$$\textsc{Assert}: \frac{e_1 \overset{*}{\mapsto} \mathsf{true}}{\mathsf{assert}\ e_1;\ e_2 \mapsto e_2}$$

Withs With-expressions $\mathsf{with}\ e_1;\ e_2$ substitute the attributes defined in the attribute set e_1 into the resulting expression e_2.

$$\textsc{With}: \frac{e_1 \overset{*}{\mapsto} \{as\}}{\mathsf{with}\ e_1;\ e_2 \mapsto \mathsf{subst}(\{n \rightsquigarrow e \mid \langle n = e\rangle \in as\}, e_2)}$$

Operators The equality operators are defined as follows.

$$\textsc{OpEq}: \frac{e_1 \overset{*}{\mapsto} e_1' \wedge e_2 \overset{*}{\mapsto} e_2' \wedge e_1' = e_2'}{e_1\ \mathtt{==}\ e_2 \mapsto \mathsf{true}} \qquad \frac{e_1 \overset{*}{\mapsto} e_1' \wedge e_2 \overset{*}{\mapsto} e_2' \wedge e_1' \neq e_2'}{e_1\ \mathtt{==}\ e_2 \mapsto \mathsf{false}}$$

Equality between expressions ($e_1' = e_2'$) is *syntactic*. Of course, inequality ($e_1\ \mathtt{!=}\ e_2$) is defined as the negation of equality.

The Boolean operators are straightforward, although it is worth noting that conjunction and disjunction (“and” and “or”) are lazy, i.e., “short-circuited”.

$$\textsc{OpNeg}: \quad \frac{e \overset{*}{\mapsto} \mathsf{false}}{!e \mapsto \mathsf{true}} \qquad \frac{e \overset{*}{\mapsto} \mathsf{true}}{!e \mapsto \mathsf{false}}$$

$$\textsc{OpAnd}: \quad \frac{e_1 \overset{*}{\mapsto} \mathsf{false}}{e_1 \;\&\&\; e_2 \mapsto \mathsf{false}} \qquad \frac{e_1 \overset{*}{\mapsto} \mathsf{true} \wedge e_2 \overset{*}{\mapsto} \mathsf{false}}{e_1 \;\&\&\; e_2 \mapsto \mathsf{false}} \qquad \frac{e_1 \overset{*}{\mapsto} \mathsf{true} \wedge e_2 \overset{*}{\mapsto} \mathsf{true}}{e_1 \;\&\&\; e_2 \mapsto \mathsf{true}}$$

$$\textsc{OpOr}: \quad \frac{e_1 \overset{*}{\mapsto} \mathsf{true}}{e_1 \;||\; e_2 \mapsto \mathsf{true}} \qquad \frac{e_1 \overset{*}{\mapsto} \mathsf{false} \wedge e_2 \overset{*}{\mapsto} \mathsf{true}}{e_1 \;||\; e_2 \mapsto \mathsf{true}} \qquad \frac{e_1 \overset{*}{\mapsto} \mathsf{false} \wedge e_2 \overset{*}{\mapsto} \mathsf{false}}{e_1 \;||\; e_2 \mapsto \mathsf{false}}$$

$$\textsc{OpImpl}: \quad \frac{e_1 \overset{*}{\mapsto} \mathsf{false}}{e_1 \rightarrow e_2 \mapsto \mathsf{true}} \qquad \frac{e_1 \overset{*}{\mapsto} \mathsf{true} \wedge e_2 \overset{*}{\mapsto} \mathsf{true}}{e_1 \rightarrow e_2 \mapsto \mathsf{true}} \qquad \frac{e_1 \overset{*}{\mapsto} \mathsf{true} \wedge e_2 \overset{*}{\mapsto} \mathsf{false}}{e_1 \rightarrow e_2 \mapsto \mathsf{false}}$$

The *update operator* e_1 // e_2 yields the *right-biased union* of the attribute sets e_1 and e_2, that is, the set consisting of the attributes in e_1 and e_2, with attributes in e_2 overriding those in e_1 if there are name conflicts.

$$\textsc{OpUpdate}: \quad \frac{e_1 \overset{*}{\mapsto} \{as_1\} \wedge e_2 \overset{*}{\mapsto} \{as_2\}}{e_1 \;/\!/\; e_2 \mapsto \{\mathsf{rightUnion}(as_1, as_2)\}}$$

where

$$\boxed{\mathsf{rightUnion}(as_1, as_2) = as_2 \cup \{\langle n = e\rangle \mid \langle n = e\rangle \in as_1 \wedge n \notin \mathsf{names}(as_2)\}} \qquad \text{|| operator}$$

Addition is defined between strings and between paths.

$$\textsc{OpPlusStr}: \quad \frac{e_1 \overset{*}{\mapsto} s_1 \wedge e_2 \overset{*}{\mapsto} s_2}{e_1 + e_2 \mapsto s_1 +_s s_2}$$

$$\textsc{OpPlusPath}: \quad \frac{e_1 \overset{*}{\mapsto} p_1 \wedge e_2 \overset{*}{\mapsto} p_2}{e_1 + e_2 \mapsto \mathsf{canonicalise}(p_1 +_s \texttt{"/"} +_s p_2)}$$

where $+_s$ denotes string concatenation.

Finally, there is an operator ? to check attribute set membership. This is used to guard the use of attributes that may not be defined, e.g.,

```
builder = if args ? builder then args.builder else bash;
```

It is implemented as follows.

$$\textsc{OpHasAttr+}: \quad \frac{e \overset{*}{\mapsto} \{as\} \wedge n \in \mathsf{names}(as)}{e \;?\; n \;\mapsto\; \mathsf{true}}$$

$$\textsc{OpHasAttr-}: \quad \frac{e \overset{*}{\mapsto} \{as\} \wedge n \notin \mathsf{names}(as)}{e \;?\; n \;\mapsto\; \mathsf{false}}$$

NOTE **drvPath, outPath, type, name, etc -** the **derivation** primop (1) creates a **store derivation (drv file)** and (2) returns the same attribute set it took PLUS drvPath, outPath, type attributes. So

derivation(as) -> as || { drvPath = ...; outPath = ...; type = "derivation"; }

So **name** (mentioned on page 63 (or 71 in PDF numbering) at purple wiggly line - and other places) comes from the original attribute set that derivation (and mkDerivation and co.) gets as input; see figure 2.9 way up.

forgot to read ahead again...

Primops Primitive operations, or *primops*, are functions that are built into the language. They need not be defined or imported; they are in scope by default. But since they are normal identifiers, they *can* be overridden by local definitions.

The most important primop, indeed the *raison d'être* for the Nix expression language, is the primop derivation. It translates a set of attributes describing a build action to a store derivation, which can then be built. The translation process is performed by the function instantiate given in Section 5.4. What is relevant here is that instantiate(as) takes a set of attributes as, translates them to a store derivation, and returns an identical set of attributes, but with three attributes added: the attribute type with string value "derivation", and the attributes drvPath and outPath containing the store paths of the store derivation and its output, respectively.

So intuitively, derivation just calls instantiate:

$$\textsc{Derivation} : \frac{e \overset{*}{\mapsto} \{as\}}{\mathsf{derivation}\ e \mapsto \{\mathsf{instantiate}(as)\}}$$

However, as I argued at the beginning of this chapter (page 63), we want derivations to be very lazy: the expensive call to instantiate should only be made if we actually need drvPath and outPath. For this reason, derivation *postpones* the computation of those attributes by using an indirection:

$$\textsc{Derivation} : \frac{e \overset{*}{\mapsto} \{as\}}{\mathsf{derivation}\ e \mapsto \{\mathsf{rightUnion}(as, as')\}}$$

TODO Have there been any changes/tweaks to this model?

where

$$\begin{aligned} as' &= \{\langle \mathsf{type} = \texttt{"derivation"}\rangle, \langle \mathsf{drvPath} = e'.\mathsf{drvPath}\rangle, \langle \mathsf{outPath} = e'.\mathsf{outPath}\rangle\} \\ e' &= \mathsf{derivation!}\ \{as\} \end{aligned}$$

The internal (not user-accessible) primop derivation! performs the actual instantiation:

$$\textsc{Derivation!} : \mathsf{derivation!}\ \{as\} \mapsto \{\mathsf{instantiate}(as)\}$$

Thus, when we evaluate attributes that result from a call to derivation, there is essentially no runtime cost *unless* we evaluate the drvPath or outPath attributes.

The next primop is import. The unary operation import p reads the file p, parses it as a Nix expression, applies certain checks, and evaluates and returns the expression. The validity checks are that the expression must be *closed* (i.e., not contain any unbound variables), that no attribute set definition defines two attributes with the same name, and similarly that no multi-argument function definition has two formal arguments with the same name. The requirement that imported expressions are closed is rather important: it implies that no imported expression can refer to the local scope of the caller. Also, any relative path constants occurring in the expression are absolutised.

$$\textsc{Import} : \frac{e \overset{*}{\mapsto} p}{\mathsf{import}\ e \mapsto \mathsf{loadExpr}(p)}$$

If p denotes a directory, the path "/default.nix" is automatically appended. This is an organisational convenience for people who want to store each component's Nix expression and builder in a separate directory (as in, e.g., Figure 2.5).

The function loadExpr takes care of loading, parsing and processing the file. The astute reader will note that loadExpr depends on the contents of the file system, that is, it depends on some mapping of paths to file system contents. This mapping is left implicit in the evaluation rules. It may also appear to make the evaluation calculus *impure*, implying that equational reasoning (an important property of purely functional languages) no longer holds. For instance, it might appear that import p == import p does not always evaluate to true. However, as we will see below, due to maximal laziness this is not in fact the case. The call import p is only evaluated once for a given p.

QUESTION The impurity coming from the fact that the content of a path can be different at different times. Very curious how **maximal laziness** overcomes this fact.

The import primitive is overloaded to also work on derivations:

$$\text{IMPORT'}:\ \frac{e \overset{*}{\mapsto} \{as\} \wedge \langle\text{"type"} = e_t\rangle \in as \wedge e_t \overset{*}{\mapsto} \text{"derivation"} \wedge \langle\text{"drvPath"} = e_p\rangle \in as \wedge e_p \overset{*}{\mapsto} p}{\text{import } e \mapsto \text{loadExpr}(\text{build}(p))}$$

QUESTION This is very important (and answers some of my questions) but it is important to clarify: what does derivation mean in this context?

I'm starting to think that **derivation** always means **the output of *derivation* primop**, that is **the input attribute set with the 3 extra attributes added: type, drvPath, outPath.**

The function build builds a store derivation. It is defined in Section 5.5 and returns the output path of the derivation. Overloading import to work on derivations allows Nix expressions to *generate* and use other Nix expressions. We will use this feature in Chapter 10 to implement build management with Nix, as it allows the automatic generation of lists of inputs from other inputs (e.g., the list of header files imported by a C source file).

The **drv** files are the ATerm representations of derivations (or something). **outPath**, **drvPath**, and **type** attributes are not present because they can be figured out from context (e.g., **type** is always "derivation" for a derivation, and the paths are the paths of the **drv** file itself.

|
V
QUESTION But what if there are multiple outputs? (it's possible, right?)

Can't wait to understand this one.

Finally, the nullary primops true, false, and null are constant symbols: they represent themselves, and have no evaluation rules. There is also an ever-growing set of utility functions (added in an admittedly *ad hoc* fashion) such as map, baseNameOf, and so on; but the semantics of those is not very relevant to a discussion of the foundations of the Nix expression language.

4.4. Implementation

Maximal laziness Nix expression evaluation is implemented using the *ATerm library* [166], which is a library for the efficient storage and runtime manipulation of terms. The Nix expression parser yields an ATerm encoding of the term. For example, the expression

```
(x: x) 123
```

yields the ATerm

```
Call(Function1("x",Var("x"),Pos("(string)",1,3)),Int(123))
```

The Pos node indicates position information, which is stored for certain language constructs (notably functions and attributes) to provide better error messages.

A very nice property of the ATerm library is its *maximal sharing*: if two terms are syntactically equal, then they occupy the same location in memory. This means that a shallow pointer equality test is sufficient to perform a deep syntactic equality test. Maximal sharing is implemented through a hash table. Whenever a new term is created, the term is looked up in the hash table. If the term already exists, the address of the term obtained from

Why?

```
eval($e$) :
    if cache[$e$] $\neq \varepsilon$ :
        if cache[$e$] $=$ blackhole :
            Abort; infinite recursion detected.
        return cache[$e$]
    else :
        cache[$e$] $\leftarrow$ blackhole
        $e' \leftarrow$ eval$'$($e$)
        cache[$e$] $\leftarrow e'$
        return $e'$
```

Figure 4.7.: Evaluation caching

the hash table is returned. Otherwise, the term is allocated, initialised, and added to the hash table. A garbage collector takes care of freeing terms that are no longer referenced.

Maximal sharing is extremely useful in the implementation of a Nix expression interpreter since it allows easy *caching* of evaluation results, which speeds up expression evaluation by removing unnecessary evaluation of identical terms. The interpreter maintains a hash lookup table cache : ATerm → ATerm that maps ATerms representing Nix expressions to their normal form. Figure 4.7 shows pseudo-code for the caching evaluation function eval, which "wraps" the evaluation rules defined in Section 4.3.4 in a caching layer. The function eval$'$ simply implements those evaluation rules. It is assumed that eval$'$ calls back into eval to evaluate subterms (i.e., every time a rule uses the relation $\overset{*}{\mapsto}$ in a condition), and that it aborts with an appropriate error message if e does not evaluate to a good normal form. Thus we obtain the desired caching. The special value ε denotes that no mapping exists in the cache for the expression[7]. Note that thanks to maximal sharing, the lookup cache[e] is very cheap: it is a lookup of a pointer in a hash table.

The function eval also perform a trick known as *blackholing* [134, 110] that allows detection of certain simple kinds of infinite recursion. When we evaluate an expression e, we store in the cache a preliminary "fake" normal form blackhole. If, during the evaluation of e, we need to evaluate e *again*, the cache will contain blackhole as the normal form for e. Due to the determinism and purity of the language, this necessarily indicates an infinite loop, since if we start evaluating e again, we will eventually encounter it another time, and so on.

TODO Yeah, a bit lightweight for this one.

Note that blackholing as implemented here differs from conventional blackholing, which overwrites a value being evaluated with a black hole. This allows discovery of self-referential values, e.g., x = ... x ...;. But it does *not* detect infinite recursions like this:

```
(rec {f = x: f x;}).f 10
```

TODO I don't even know how to start evaluating this one.

since every recursive call to f creates a *new* value of x, and so blackholing will not catch the infinite recursion. In contrast, our blackholing *does* detect it, since it is keyed on maximally shared ATerms that represent *syntactically equal* expressions. The example

[7] In the ATerm library, this means that the table lookup returned a null pointer.

above is evaluated as follows:

There you go...

$$\begin{array}{rcl} & & \texttt{(rec \{f = x: f x;\}).f 10} \\ (\textsc{Rec}) & \mapsto & \texttt{\{f = x: (rec \{f = x: f x;\}).f x;\}.f 10} \\ (\textsc{Select}) & \mapsto & \texttt{(x: (rec \{f = x: f x;\}).f x) 10} \\ (\beta\text{-}\textsc{Reduce}) & \mapsto & \texttt{(rec \{f = x: f x;\}).f 10} \end{array}$$

This final expression is equal to the first (which is blackholed at this time), and so an infinite recursion is signalled.

The current evaluation cache never forgets the evaluation result of any term. This obviously does not scale very well, so one might want to clear or prune the cache eventually, possibly using a least-recently used (LRU) eviction scheme. However, the size of current Nix expressions (such as those produced during the evaluation of Nixpkgs) has not compelled me to implement cache pruning yet. It should also be noted that due to maximal sharing, cached values are stored quite efficiently.

TODO How about 16 years later?

The expression caching scheme described here makes the Nix expression evaluator *maximally lazy*. Languages such as Haskell are *non-strict*, meaning that values such as function arguments or let-bindings are evaluated only when necessary. A stronger property is *laziness*, which means that these values are evaluated at most once. (Even though the semantics of Haskell only requires non-strictness, in practice all implementations are lazy.) Finally, *maximal laziness* means that syntactically identical terms are evaluated at most once. This is not the case with mere laziness: a compiler is not obliged (or generally capable) to arrange for the evaluation result of two terms occurring in different locations in the program, or in different invocations of a function, to be shared. For instance, there is no guarantee that a lazy Haskell implementation will evaluate the expression product [1..1000] only once when evaluating variable z in the following program:

```
x = product [1..1000]
y = product [1..1000]
z = x + y
```

Maximal laziness simplifies the implementation of the Nix expression evaluator. For instance, it is not necessary to implement sharing of β-redexes. Without sharing, it is typically necessary to implement the rule β-Reduce as follows:

TODO This is greek right now.

$$\beta\text{-}\textsc{Reduce}: \frac{e_1 \overset{*}{\mapsto} x\colon e_3}{e_1\ e_2 \mapsto \mathsf{let}\ x = e_2\ \mathsf{in}\ e_3}$$

where we give let a special "destructive update" semantics so that the evaluation result of x is written back into the right-hand side of the let-binding [51]. This is typically how laziness is implemented in functional languages: x is a pointer that points to a piece of memory containing code and environment pointers (the *closure* or *thunk* [134]), which after evaluation is overwritten with the actual result.

Error messages If Nix expression evaluation fails, or we import a syntactically or otherwise invalid Nix expression, a backtrace is printed of the evaluation stack to give the user some clue as to the cause of the problem. For instance, if we try to install Subversion from a Nix expression that calls the one in Figure 2.9 with compressionSupport set to true, but with zlib set to null, we get the following error message:

```
$ nix-env -f ./system/populate-cache.nix -i subversion
error: while evaluating the attribute ‘subversion’
  at ‘./system/all-packages-generic.nix’, line 1062:
while evaluating the function
  at ‘(...)/subversion-1.2.x/default.nix’, line 1:
assertion failed
  at ‘(...)/subversion-1.2.x/default.nix’, line 19
```

This tells the user not only what assertion triggered the error, but also where the incorrect call to the Subversion function was made (namely, in all-packages-generic.nix at line 1062).

Backtraces in lazy languages are tricky, as a failing value may be finally evaluated in a piece of code far removed from where the value was “produced” [55]. Consider for example:

```
let {
  f = b: {x = assert b; 123;};
  body = (f false).x;
}
```

Its evaluation will yield the following backtrace:

```
error: while evaluating the file ‘foo.nix’:
while evaluating the attribute ‘body’ at ‘foo.nix’, line 3:
while evaluating the attribute ‘x’ at ‘foo.nix’, line 2:
assertion failed at ‘foo.nix’, line 2
```

Note the absense of a reference to the function f. In a strict language, f would appear in the backtrace. But also note that since attributes are annotated with position information, we still get the position of the definition of the attribute x in the body of f.

Another problem with backtraces in lazy languages is that tail-call optimisation [151] may remove useful stack frames. However, the current Nix expression evaluator does not optimise tail-calls. TODO Any change here?

In practice, however, the stack traces printed by Nix are very helpful in everyday use, as the example above shows. The trace may not show every pertinent code location, and it may not show them in a meaningful order, but the locations that it does give usually give enough of a clue to figure out the cause of the problem. A useful improvement is a kind of *slicing* for assertion failures (which are the most common type of errors, apart from missing attributes or function arguments). Currently, the interpreter simply prints that an assertion has failed, i.e., its condition expression evaluated to false. So it is useful to print an explanation as to *why* it is false. For example, if a guard expression p && x != null evaluates to false, Nix should print whether p is false, or x != null is false. If p is false, it should give an explanation as to *how* p obtained that value. In an interpreter (as opposed to a compiled implementation), this is not hard to implement since the original expression is still known, not just its normal form.

Optimising substitution There is an important opportunity for optimisation in the implementation of β-reduction: if we substitute a term, e.g., replacing free occurrences of

TODO the usual

x in e_1 by e_2 in the β-reduction of $(x:\ e_1)\ e_2$, we never need to substitute under the replacements of x. Consider the expression $(x:\ y:\ e_1)\ e_2\ e_3$, where e_2 is a large expression. With normal substitution, we first replace all occurrences of x in e_1 with e_2. Then, we replace all occurrences of y in the resulting term with e_3. This substitution also descends into the e_2 replacements of x, even though those subterms are closed. Since e_2 is large, this is inefficient.

The optimisation is that we can *mark* replacement terms to indicate to the substitution function that it need not descend into such subterms. This is possible because of the following theorems.

Lemma 1. *Every term to which a semantic rule is applied during evaluation is closed.*

Proof. The property follows by induction on the application of semantic rules. The base case is the initial terms produced by the parser. Here the induction hypothesis hold trivially since the parser checks that these terms are closed and aborts with an error otherwise.

For the inductive step, we must show two things.

First, we must show that each rule that takes a closed term produces a closed term. This can be easily verified. For instance, β-REDUCE yields the body of the function, e_3, with its sole free variable x substituted by the argument. e_3 cannot have additional free variables because the function call $e_1\ e_2$ is closed, and function calls do not bind variables. Likewise, the expression yielded by SELECT must be closed because the attribute set in which it was contained is closed, and attribute sets do not bind variables.

Second, we must show that subterms evaluated in the conditions of rules are closed. This can also be easily verified. For instance, β-REDUCE recursively evaluates the expression e_1 in a function call $e_1\ e_2$. Since by the induction hypothesis $e_1\ e_2$ is closed, it follows that e_1 is also closed, as function calls do not bind variables. □

Theorem 2. *All substitution terms occurring during evaluation are closed.*

Proof. This property follows by inspecting all calls to subst in the evaluation rules and observing that the substitution terms (i.e., the expressions in *subs*) are always closed. For instance, in the rule β-REDUCE, $subs = \{x \rightsquigarrow e_2\}$. The term e_2 is closed because by Lemma 1 the call $e_1\ e_2$ is closed, and function calls bind no variables. A similar argument holds for the subst calls in REC, β-REDUCE' and WITH. □

Since substitution terms are always closed, we can adapt the variable case of the substitution function subst in Figure 4.6 as follows:

$$\mathsf{subst}(subs, x) = \begin{cases} \mathsf{closed}(e) & \text{if } (x \rightsquigarrow \mathsf{closed}(e)) \in subs \\ \mathsf{closed}(e) & \text{if } (x \rightsquigarrow e) \in subs \\ x & \text{otherwise} \end{cases}$$

That is, replacement terms e are placed inside a wrapper $\mathsf{closed}(e)$. (The first case merely prevents redundant wrapping, e.g., $\mathsf{closed}(\mathsf{closed}(e))$, which reduces the effectiveness of caching and blackholing.) The wrapper denotes that e is a closed subterm under which no substitution is necessary, since it has no free variables. To actually make use of this optimisation, we also add a case to subst to stop at closed terms:

$$\mathsf{subst}(subs, \mathsf{closed}(e)) = \mathsf{closed}(e)$$

Of course, during evaluation we must get rid of closed eventually. That's easy:

$$\textsc{Closed} : \mathsf{closed}(e) \mapsto e$$

as a closed term is semantically equivalent to the term that it wraps.

5. The Extensional Model

In Chapter 2, we have seen that Nix offers many advantages over existing deployment systems, such as reliable dependency information, side-by-side deployment of versions and variants, atomic upgrades and rollbacks, and so on. And as explained in Chapter 3, it obtains these properties by imposing a "pointer discipline" on file systems, thus treating the component deployment problem similar to memory management in programming languages.

The current and subsequent chapter formalise the operation of the Nix system. This includes the translation of Nix expressions into store derivations, building store derivations, the substitute mechanism, garbage collection, the invariants of the Nix store that must hold to ensure correct deployment, and so on.

There are actually two different variants or *models* of Nix. The original model is the *extensional model*, described in this chapter. It is the model with which we have had the most experience. It implements all of the deployment properties described in Chapter 2. The next chapter describes the *intensional model* which is more powerful in that it allows the sharing of a Nix store between mutually untrusted users; however, it also places slightly stricter requirements on components. Most of the aspects of the extensional model carry over without modification to the intensional model.

This chapter discusses all aspects of the semantics of the extensional model: the basics of cryptographic hashing (Section 5.1), the notion of File System Objects and the organisation of the Nix store (Section 5.2), the operation of adding sources and other "atomic" values to the store (Section 5.3), translating Nix expressions to store derivations (Section 5.4), building store derivations (Section 5.5), and garbage collection (Section 5.6). It concludes with an explanation of the notion of extensionality (Section 5.7), which sets up the next chapter on the intensional model.

5.1. Cryptographic hashes

Since Nix heavily uses cryptographic hash functions in store paths and in other places, this section introduces some properties and notations that will be used in the remainder of Part II.

A *hash function* [36] is a function $h : A \rightarrow B$ that maps values from a possibly infinite domain A onto a finite range B. For instance, $h(x \in N) = x \bmod 27$ is a hash function that maps natural numbers to the numbers $[0..26]$. Given value $x \in A$ and $y = h(x) \in B$, the value x is called the *preimage* and y is called the *hash* of x. Since A is typically larger than B, there must necessarily exist values $x_1, x_2 \in A$ such that $h(x) = h(y)$. These are *collisions* of the hash function h. A good hash function will have the property that collisions can be expected to occur with low probability given the sets of values that will be fed into the hash function in typical usage patterns.

A *cryptographic hash function* [145] is a hash function that maps arbitrary-length byte sequences onto fixed-length byte sequences, with a number of special properties:

- *Preimage resistance*: given a value $y \in B$, it should be computationally infeasible to find an $x \in A$ such that $y = h(x)$. Computational infeasibility means that ideally the most efficient way to find x is a brute-force attack, that is, trying all possible values in A. This means computing $|A|/2$ hashes on average.

- *Collision resistance*: it should be infeasible to find any $x_1, x_2 \in A$ such that $h(x_1) = h(x_2)$. Due to the birthday paradox [145], a brute-force attack will find a collision with probability $\frac{1}{2}$ after computing approximately $\sqrt{|B|}$ hashes.

- *Second preimage resistance*: given an $x_1 \in A$, it should be infeasible to find another $x_2 \in A$ such that $h(x_1) = h(x_2)$. The difference with collision resistance is that here x_1 is given.

For our purposes collision resistance and second preimage resistance are what matters—it should not be feasible to find collisions. This is since we do not ever want two different components in the Nix store to be stored in the same store path. Preimage resistance—infeasibility of inverting the hash—is not important. We do not care if it is possible to find a component given a store path.

Prominent examples of cryptographic hash functions are MD5 [141] and SHA-1 [127]. Nix initially used MD5, which produces 128-bit hashes. However, MD5 has recently been broken [178] in the sense that it is now possible to find collisions with a work effort of a few hours on conventional hardware. That is, collision resistance has been broken, but this technique does not find preimages or second preimages.

In the context of Nix, a collision attack such as the one against MD5 means that it is possible to construct two components that hash to the same store path. Specifically, an attacker could create two components, one benevolent and one malevolent (e.g., containing a Trojan horse—a malicious component masquerading as legitimate software), with the same MD5 hash. The attacker would then publish a Nix expression that uses fetchurl to download the benevolent component from a server, and finally replace the benevolent component with the malevolent one on the server. Since on download the hash continues to match, the deception would not be noticed. Constructing such components is quite easy (see, e.g., [156, Section 8.4.4], and [120, 105] for examples of the construction of harmless and harmful executables with matching MD5 hashes). Thus collisions of the underlying hash are a serious concern.

NOTE Bottom line is, SHA-1 is broken: https://shattered.io/

SHA-1 is a very widely used hash function. It produces 160 bit hashes, making it more resistant to birthday attacks in theory as it requires 2^{80} hashes on average in a brute-force attack. However, recent attacks [177, 146] have reduced the effort of finding a collision to less than 2^{63} hashes. This is still a substantial effort, but it creates enough doubt regarding the future security of SHA-1 to require a transition away from it ("Attacks always get better; they never get worse." [147]).

There are a number of strengthened variants of SHA-1 that produce larger hashes. These are SHA-224, SHA-256, SHA-384, and SHA-512 [127]. While there are strong doubts about the future security of these hashes, they represent the best choice until a next-generation hash function becomes available (e.g., through a similar process as the AES selection [126]).

The Nix system uses SHA-256 hashes. However, MD5 and SHA-1 are available in functions such as fetchurl for compatibility. The use of these hashes is still safe if they apply to components from a trusted source, since it remains infeasible for a third-party attacker to find second preimages. However, we should migrate away from these hashes to SHA-256.

Notation *Byte sequences* are finite-length sequences of bytes. The size of byte sequence c is denoted $|c|$. Given a byte sequence c of size $|c|$, we write $c[i]$, $0 \le i < |c|$ to denote the $(i+1)$'th byte in the sequence. In examples, we write byte sequences initialised from ASCII characters between quotes, e.g., "Hello" is a byte sequence of length 5.

Given a byte sequence c, we write $\mathsf{h} = \mathsf{hash}_t(c)$ to denote the byte sequence obtained from the cryptographic hash function algorithm t, where $t \in \{\mathsf{md5}, \mathsf{sha1}, \mathsf{sha256}\}$.

QUESTION What are the output of the hash functions (e.g., MD5, SHA-1, SHA-256, etc.)?

The function $\mathsf{printHash}_{16}$ returns a hexadecimal string representation of a byte sequence, useful for printing hashes. It is defined as follows:

$$\mathsf{printHash}_{16}(c) = \sum_{0 \le i < |c|} (\mathsf{digits}_{16}[c[i] \operatorname{div} 16] + \mathsf{digits}_{16}[c[i] \bmod 16])$$

TODO Get to the bottom of this. I presume that **c** is the resutl of a hash function (i.e., **h** above).

where $+$ and $\sum$ denote string concatenation, and digits_{16} is the byte sequence of hexadecimal ASCII characters, i.e., "0123456789abcdef". For example,

$$\mathsf{printHash}_{16}(\mathsf{hash}_{\mathsf{md5}}(\text{"Hello World"})) = \text{"b10a8db164e0754105b7a99be72e3fe5"}$$

There is also a function $\mathsf{printHash}_{32}$ that returns a base-32 string representation of a byte sequence. It produces representations of length $\lceil |c| \times 8/\lg 32 \rceil = \lceil |c| \times \frac{8}{5} \rceil$. Since this is shorter than the representations produced by $\mathsf{printHash}_{16}$ (which have length $|c| \times 2$), it is useful *inter alia* for the representation of hashes in store paths. It is defined as follows:

$$\mathsf{printHash}_{32}(c) = \sum_{\lceil |c| \times \frac{8}{5} \rceil > i \ge 0} \mathsf{digits}_{32}[(\sum_{0 \le j < |c|} c[j] \times 256^j)/32^i \bmod 32]$$

TODO It wasn't just the mention of encodings that threw me of; simply couldn't parse what this sentence is trying to say.

That is, interpreting c as a base-256 encoding of some number, with digits from least significant to most significant (i.e., a little-endian number), we print the number in base-32 with digits from most significant to least significant. (Note that the outer summation denotes string concatenation, while the inner summation denotes integer addition.) The set of digits is

NOTE Documenting my confusing regarding BaseN encodings: https://stackoverflow.com/questions/71933607/what-where-is-the-formal-specification-of-basen-e-g-base64-encodings

$$\mathsf{digits}_{32} = \text{"0123456789abcdfghijklmnpqrsvwxyz"}$$

i.e., the alphanumerics excepting the letters e, o, u, and t. This is to reduce the possibility that hash representations contain character sequences that are potentially offensive to some users (a known possibility with alphanumeric representations of numbers [11]). Here is an example of $\mathsf{printHash}_{32}$:

$$\mathsf{printHash}_{32}(\mathsf{hash}_{\mathsf{sha1}}(\text{"Hello World"})) = \text{"s23c9fs0v32pf6bhmcph5rbqsyl5ak8a"}$$

QUESTION Why?

It is sometimes necessary to truncate a hash to a specific number of bytes n. For instance, below we will truncate 32-byte SHA-256 hashes to 20 bytes (of course, this reduces the

TODO But more importantly, did it mention **how store hashes are generated and how long they are**? This section is supposed to be an intro to cryptographic hashing, so I presume no, but it also went into some specific details about the store. I probably missed something crucial but couldn't find it.

security of the hash). This truncation is done by cyclically XORing the input with an initially zero-filled byte array of length n. Formally, the result of $\mathsf{truncate}(n,c)$ is defined by the equation

$$\mathsf{truncate}(n,c)[i] = \bigoplus_{0 \leq j < |c|,\, j \bmod n = i} c[j]$$

where $\oplus$ denotes the XOR operator.

5.2. The Nix store

5.2.1. File system objects

The Nix store is a directory in the file system that contains *file system objects* (FSOs)[1], such as files and directories. In the context of the Nix store, FSOs typically represent software components, user environments and store derivations. As we shall see below, Nix must perform various operations on FSOs, such as computing cryptographic hashes over their contents, scanning for strings, rewriting strings, and so on. Thus, it is useful to formalise the notion of FSOs.

Operating systems have different kinds of objects that live in a file system. All modern operating systems at the very least have hierarchical directories and files that consist of unstructured sequences of bytes (as opposed to, say, files consisting of records). Beyond that, there are many extensions:

- Symbolic links on Unix.
- Permissions, ranging from MS-DOS's read-only attribute, through Unix's access rights, to complex Access Control Lists (Windows NT, various Unixes).
- Extended attributes (OS/2, Windows NT) that allow arbitrary name/value pairs to be associated with files.
- Streams (Windows NT) or resource forks (Mac OS) that extend the notion of a file as a sequence of bytes to being *sequences* of bytes.
- Forests of directory hierarchies (e.g., as formed by drive letters in Windows).
- Hard links that turn file systems into directed acyclic graphs instead of trees.
- Device nodes, FIFOs, named pipes, sockets, and other types of files with special semantics.

QUESTION & ANSWER
https://www.quora.com/How-are-directed-acyclic-graphs-different-from-trees
Beautifully simple answer to my confusion.

Fortunately, for our purposes—which is storing components in the Nix store—most of these features are either not relevant (drive letters, device nodes, etc.), can be ignored (hard links), or are never used in practice (streams). Some others, such as permissions, we will ignore to make life simpler.

[1] In the Unix tradition, the term *file* refers to any file system object (i.e., an FSO), while the term *regular file* refers to what most people call a file. This thesis uses the term *FSO* to avoid the ambiguity.

```
data FSO = Regular Executable ByteStream
         | Directory [(FileName, FSO)]
         | SymLink ByteStream

data Executable = Executable
                | NonExecutable
```

Figure 5.1.: Abstract syntax for file system objects

What we absolutely cannot ignore for deployment of Unix software is regular files, directories, and symbolic links. Also, while we can get away with disregarding most file permissions, the executable bit must be maintained[2]. This leads to the grammar for FSOs shown in Figure 5.1. (The data type for FSOs shown in this figure uses the Haskell-like notation described in Section 1.7.) A ByteStream is a sequence of 0 or more bytes, and a FileName is a sequence of 1 or more bytes not equal to 0. For example, the file system object

```
Directory [
    ("foo", Regular NonExecutable "Hello World"),
    ("bar", Directory [
        ("xyzzy", Symlink "../foo")
    ])
]
```

defines a directory with two files: a non-executable file foo with contents Hello World, and a subdirectory bar containing a symlink called xyzzy to foo in the parent directory. Note that the top-level directory is anonymous. Also note that we do not keep any meta-information about files other than the executable bit; notably, time stamps are discarded.

The type of FSOs defined in Figure 5.1 is not actually used in the implementation of Nix. They are merely a device used in this chapter to formalise certain aspects of the operation of Nix. In the pseudo-code algorithms in this chapter, I will use the imperative functions $fso \leftarrow \mathsf{readPath}(p)$ to denote reading the FSO fso stored at path p, and $\mathsf{writePath}(p, fso)$ to denote writing FSO fso to path p. For presentational simplicity, I ignore error handling aspects.

To perform operations on FSOs such as computing cryptographic hashes, scanning for references, and so on, it is useful to be able to *serialise* FSOs into byte sequences, which can then be *deserialised* back into FSOs that are stored in the file system. Examples of such serialisations are the ZIP and TAR file formats. However, for our purposes these formats have two problems:

- They do not have a *canonical serialisation*, meaning that given an FSO, there can be many different serialisations. For instance, TAR files can have variable amounts of padding between archive members; and some archive formats leave the order

[2] Nix should also support setuid executables, which are programs that are executed under a different user ID than the caller. However, it is not entirely clear how to support these in a clean way.

TODO I remember having seen discussions about this but not where (Discourse? Github issues?) and any of the specifics.

of directory entries undefined. This is bad because we use serialisation to compute cryptographic hashes over FSOs, and therefore require the serialisation to be unique. Otherwise, the hash value can depend on implementation details or environment settings of the serialiser. The canonicity of archives will be particularly important in Section 7.5, when we apply binary patches to archives.

- They store more information than we have in our notion of FSOs, such as time stamps. This can cause FSOs that Nix should consider equal to hash to different values on different machines, just because the dates differ.[3]
- As a practical consideration, the TAR format is the only truly universal format in the Unix environment. It has many problems, such as an inability to deal with long file names and files larger than 2^{33} bytes. Current implementations such as GNU Tar work around these limitations in various ways.

For these reasons, Nix has its very own archive format—the *Nix Archive* (NAR) format. Figure 5.2 shows the serialisation function $\mathsf{serialise}(\mathit{fso})$ that converts an FSO to a byte sequence containing a NAR archive. The auxiliary function $\mathsf{concatMap}(f, xs)$ applies the function f to every element of the list xs and concatenates the resulting byte sequences. The function sortEntries sorts the list of entries in a directory by name, comparing the byte sequences of the names lexicographically. The function serialise is typically used in conjunction with readPath, e.g.,

$$c \leftarrow \mathsf{serialise}(\mathsf{readPath}(p)).$$

Of course, there is also a deserialisation operation deserialise(c) (typically used with writePath) that converts a byte sequence c containing a NAR archive to an FSO. It is the inverse of serialise and is not shown here. The following identity holds:

$$\mathsf{deserialise}(\mathsf{serialise}(\mathit{fso})) = \mathit{fso}$$

Note that deserialise is a partial function, since most byte sequences are not valid NAR archives.

5.2.2. Store paths

A *store object* is a top-level file system object in the Nix store, that is, a direct child of the Nix store directory. Indirect subdirectories of the Nix store are not store objects (but contained in FSOs that are store objects).

A *store path* is the full path of a store object. It has the following anatomy:

storeDir/*hashPart*-*name*

The first part is the path of the Nix store, denoted *storeDir*. The second is a 160-bit cryptographic hash in base-32 representation. Finally, there is a symbolic name intended for human consumption. An example of a store path is:

/nix/store/bwacc7a5c5n3qx37nz5drwcgd2lv89w6-hello-2.1.1

[3] This is no longer a big issue, since Nix nowadays *canonicalises* all FSOs added to the store by setting irrelevant metadata to fixed values (see page 112). For instance, it sets the time stamps on all files to 0 (1 Jan 1970, 00:00:00 UTC). However, metadata such as the last-accessed time stamp might still cause non-canonicity.

$$\mathsf{serialise}(\mathit{fso}) = \mathsf{str}(\texttt{"nix-archive-1"}) + \mathsf{serialise}'(\mathit{fso})$$

$$\mathsf{serialise}'(\mathit{fso}) = \mathsf{str}(\texttt{"("}) + \mathsf{seralise}''(\mathit{fso}) + \mathsf{str}(\texttt{")"})$$

$$\begin{aligned}
&\mathsf{serialise}''(\mathsf{Regular}\ \mathit{exec}\ \mathit{contents}) = \\
&\quad \mathsf{str}(\texttt{"type"}) + \mathsf{str}(\texttt{"regular"}) \\
&\quad + \begin{cases} \mathsf{str}(\texttt{"executable"}) + \mathsf{str}(\texttt{""}), & \text{if } \mathit{exec} = \mathsf{Executable} \\ \texttt{""}, & \text{if } \mathit{exec} = \mathsf{NonExecutable} \end{cases} \\
&\quad + \mathsf{str}(\texttt{"contents"}) + \mathsf{str}(\mathit{contents})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{serialise}''(\mathsf{SymLink}\ \mathit{target}) = \\
&\quad \mathsf{str}(\texttt{"type"}) + \mathsf{str}(\texttt{"symlink"}) \\
&\quad + \mathsf{str}(\texttt{"target"}) + \mathsf{str}(\mathit{target})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{serialise}''(\mathsf{Directory}\ \mathit{entries}) = \\
&\quad \mathsf{str}(\texttt{"type"}) + \mathsf{str}(\texttt{"directory"}) \\
&\quad + \mathsf{concatMap}(\mathsf{serialiseEntry}, \mathsf{sortEntries}(\mathit{entries}))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{serialiseEntry}((\mathit{name}, \mathit{fso})) = \\
&\quad \mathsf{str}(\texttt{"entry"}) + \mathsf{str}(\texttt{"("}) \\
&\quad + \mathsf{str}(\texttt{"name"}) + \mathsf{str}(\mathit{name}) \\
&\quad + \mathsf{str}(\texttt{"node"}) + \mathsf{serialise}'(\mathit{fso}) \\
&\quad + \mathsf{str}(\texttt{")"})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{str}(s) = \mathsf{int}(|s|) + \mathsf{pad}(s) \\
&\mathsf{int}(n) = \text{the 64-bit little endian representation of the number } n \\
&\mathsf{pad}(s) = \text{the byte sequence } s\text{, padded with 0s to a multiple of 8 bytes}
\end{aligned}$$

Figure 5.2.: serialise: Serialising file system objects into NAR archive

On the other hand,

```
/nix/store/bwacc7a5c5n3qx37nz5drwcgd2lv89w6-hello-2.1.1/bin/hello
```

is not a store path (but it has a prefix that is a store path).

Usually, *storeDir* is /nix/store, and in this thesis we will tacitly assume that it is. For binary deployment purposes (Section 5.5.3), it is important that Nix store locations on the source and target machines match.

The symbolic name consists of one or more characters drawn from the set of letters (in either case), digits, and the special characters in the set "+-._?=".

We will assume that store paths are always fully *canonicalised* (see page 73), i.e., they are absolute, do not contain . or .. elements, and do not have redundant separators (/). The store path above is in canonical form; a non-canonical example of the same path is:

```
/nix/../nix//./store/bwacc7a5c5n3qx37nz5drwcgd2lv89w6-hello-2.1.1/
```

No component of the store location *storeDir* is allowed to be a symbolic link. The reason is that some builders canonicalise paths by resolving symlink components. These paths may then be stored in derivation outputs as retained dependencies. Then, if *storeDir* has different symlink components on different machines, builds on these machines might not be interchangeable.

In the remainder, we denote the infinite set of syntactically correct store paths as Path. Syntactic correctness depends on the value of *storeDir*—the location of the Nix store—but we leave that implicit. We will use the convenience function $\mathsf{hashPart}(p)$ to extract the hash part of a store path. Likewise, $\mathsf{namePart}(p)$ yields the symbolic name part of p.

How do we compute the hash part of store paths? The hash computation has some important properties. First, both the Nix store path and the symbolic name are part of the hash. This means that Nix stores in different locations yield different hash parts, and that sources or outputs that are identical except for their symbolic names have different hash parts. This is important because the hash scanning approach to dependency determination described in Section 3.4 only looks at the hash parts of paths, not the symbolic name.

Second, different types of store objects—notably sources and outputs—have different hashes. This prevents users from adding sources that "impersonate" outputs. In unshared Nix stores, this is not a major issue, but it is important for security in a shared store that uses the technique described in Section 6.2 to enable sharing of locally built outputs.

These properties are achieved in the function makePath, which computes store paths as follows:

$$\begin{array}{l}\mathsf{makePath}(\mathit{type},\ \mathit{descr},\ \mathit{name}) = \\ \quad \mathit{nixStore} + \texttt{"/"} + \mathsf{printHash}_{32}(\mathsf{truncate}(20,\ \mathsf{hash}_{\mathsf{sha256}}(s))) + \texttt{"-"} + \mathit{name}\end{array}$$

where

$$s = \mathit{type} + \texttt{":sha256:"} + \mathit{descr} + \texttt{":"} + \mathit{nixStore} + \texttt{":"} + \mathit{name}$$

(Examples will be shown in the remainder of this chapter.) Thus, the hash part of the store path is a base-32 representation of a 160-bit truncation of a SHA-256 hash of the variable elements that must be represented in the hash, namely, the location of the Nix store *nixStore*, the type of the store object (e.g., "source"), a unique descriptor string describing the store object (e.g., a cryptographic hash of the contents of the FSO in case of sources), and the symbolic name.

Why do we truncate the SHA-256 hash to 160 bits? Nix originally used base-16 (hexadecimal) representations of 128-bit MD5 hashes, giving a hash part of $128/\lg 16 = 32$ characters. Since MD5 was recently broken (see Section 5.1), a decision was made to switch to 160-bit SHA-1 hashes. The hexadecimal representation was at that time replaced with a base-32 representation, which has the pleasant property of keeping the length at $160/\lg 32 = 32$ characters. (Some builders strip off hash parts from store paths using a hard-coded constant of 32 characters (clearly a bad practice), so these did not need to be changed.) However, as attacks against SHA-1 were also published, SHA-256 was used instead. Unfortunately, hash parts of $\lceil 256/\lg 32 \rceil = 52$ characters are a bit too long: together with the symbolic name, they exceed the maximum file name lengths of some file systems. In particular, the Joliet extension to ISO-9660 has a maximum file name length of 64 characters [38]. This would make it difficult to burn Nix stores on a CD-ROM, a potentially useful distribution method. For this reason the SHA-256 hash is truncated to 160 bits. The assumption is that this is no less secure than 160-bit SHA-1, and possibly more secure due to its different structure. However, given that most modern hashes share the same basic design, the security of all of them might be ephemeral.

5.2.3. Path validity and the closure invariant

Nix maintains meta-information about store paths in a number of database tables. This information includes the following:

- The set of *valid paths*, which are the paths that are “finished”, i.e., whose contents will no longer change. For instance, the output path of a derivation isn’t marked as valid until the builder that produces it finishes successfully. The contents of an invalid path are undefined (except when the FSO is under construction, in which case a lock is used to indicate this fact, as we shall see below).
- The references graph (i.e., the deployment analogue of the pointer graph in programming languages). For each valid path, Nix maintains the set of references to other valid paths discovered by scanning for hash parts.
- The substitutes that have been registered by tools such as nix-pull (Section 2.6).
- Traceability information: for each valid output path, Nix remembers the derivation that built it.

The current Nix implementation stores this information in a transactional Berkeley DB database. As we shall see, transactions are important to ensure that the store invariants described below always hold, that interrupted operations can be restarted, and that multiple operations can be safely executed in parallel.

The most important pieces of information in the database are the set of valid paths and the references graph, which are defined here. The other database tables are defined below. The type of a database table t is given as $t : A \rightarrow B$, where A is the type of the key and B is the type of the value corresponding to a key; $t[x]$ denotes the value corresponding to key x. The special value ε indicates that no entry occurs for a value in the given mapping. Setting of table entries is denoted by $t[x] \leftarrow y$.

The mapping $\mathsf{valid} : \mathsf{Path} \rightarrow \mathsf{Hash}$ serves two purposes. First, it lists the set of valid paths—the paths that have been successfully built, added as a source, or obtained through a substitute. It does not include paths that are currently being built, or that have been left over from failed operations. Thus, $\mathsf{valid}[p] \neq \varepsilon$ means that path p is valid. The following invariant states that valid paths should exist.

Invariant 1 (Validity invariant)**.** *If a path is valid, it exists in the Nix store:*

$$\forall p \in \mathsf{Path} : \mathsf{valid}[p] \neq \varepsilon \rightarrow \mathsf{pathExists}(p)$$

Second, the valid mapping stores a SHA-256 cryptographic hash of the contents of each valid path. That is, when the path is made valid, the valid entry for the path is set as follows:

$$\mathsf{valid}[p] \leftarrow \texttt{"sha256:"} + \mathsf{printHash}_{32}(\mathsf{hash}_{\mathsf{sha256}}(\mathsf{serialise}(\mathsf{readPath}(p))))$$

(The name of the hash algorithm is incorporated to facilitate future upgrading of the hash algorithm.) For instance, given a valid path /nix/store/bwacc7a5c5n3...-hello-2.1.1 with an FSO whose serialisation has SHA-256 hash 085xkvh8plc0..., we have

$$\mathsf{valid}[\text{"/nix/store/bwacc7a5c5n3...-hello-2.1.1"}] = \text{"sha256:085xkvh8plc0..."}$$

The purpose of storing these content hashes is to detect accidental tampering with store objects, which are supposed to be read-only. The command nix-store --verify --check-contents checks that the contents of store paths are still consistent with their stored hashes. That is, the following invariant must hold.

Invariant 2 (Stored hash invariant)**.** *The contents at valid store paths correspond to the cryptographic hashes stored in the* valid *table:*

$$\begin{aligned}&\forall p \in \mathsf{Path} : \mathsf{valid}[p] \neq \varepsilon \rightarrow\\ &\quad \mathsf{valid}[p] = \text{"sha256:"} + \mathsf{printHash}_{32}(\mathsf{hash}_{\mathsf{sha256}}(\mathsf{serialise}(\mathsf{readPath}(p))))\end{aligned}$$

The mapping $\mathsf{references} : \mathsf{Path} \rightarrow \{\mathsf{Path}\}$ maintains the references graph, i.e., the set of store paths referenced by each store object. For sources, the references are usually empty. For store derivations, they are exactly the union of the store paths listed in its inputSrcs and inputDrvs fields. For output paths, they are found through scanning using the method outlined in Section 3.4. As we will see in Section 5.6.3, the references graph is acyclic except for trivial cycles defined by self-references, i.e., when $p \in \mathsf{references}[p]$. Self-references occur when a builder stores its output path in its output.

An important property of the Nix store is that at all times the following invariant holds.

Invariant 3 (Closure invariant)**.** *The set of valid paths is closed under the* references *relation:*

$$\forall p \in \mathsf{Path} : \mathsf{valid}[p] \neq \varepsilon \rightarrow \forall p' \in \mathsf{references}[p] : \mathsf{valid}[p'] \neq \varepsilon$$

The *closure* of a valid path p is the set of all paths reachable by traversing references from p:

$$\mathsf{closure}(p) = \{p\} \cup \bigcup_{p' \in \mathsf{references}[p]} \mathsf{closure}(p')$$

As I explained in Section 2.1, the closure is vitally important for correct deployment, since the closure of a component is the set of paths that might be accessed in an execution involving the component. The closure is what must be copied in its entirety when we deploy the component. Invariant 3 (closure) and Invariant 1 (validity) together give us complete deployment: if a path p is valid, then all its potential runtime dependencies are also valid; and all these paths exist in the file system.

The references graph is also maintained in the opposite direction. Nix maintains a mapping $\mathsf{referers} : \mathsf{Path} \rightarrow \{\mathsf{Path}\}$ that defines the transpose of the graph defined by the references mapping[4]. Thus, the following invariant must hold.

[4] The referers table should properly be spelled referrers. However, the Nix implementation unwittingly followed the unfortunate tradition established by the HTTP standard, which has a misspelled Referer header field [57].

Invariant 4 (Integrity invariant)**.** *The graphs defined by the* references *and* referers *mappings are each other's transposes:*

$$\forall p, p' \in \mathsf{Path} : p \in \mathsf{references}(p') \leftrightarrow p' \in \mathsf{referers}(p)$$

We can also compute the closure of a valid path under the referers relation (the *referrer closure*):

$$\mathsf{closure}^T(p) = \{p\} \cup \bigcup_{p' \in \mathsf{referers}[p]} \mathsf{closure}^T(p')$$

This is the set of all paths that can reach path p in the references graph. Note that while closure(p) is constant for a valid path (i.e., can never change due to the immutable nature of valid objects), closure$^T(p)$ is variable, since additional referrers can become valid (e.g., due to building), and existing referrers can become invalid (due to garbage collection).

The main application of referrers is to allow users to gain insight in the dependency graph. The command nix-store --query --referers allows users to query what paths refer to a given path. For instance,

```
$ nix-store -q --referers /nix/store/wa2xjf809y7k...-glibc-2.3.5
/nix/store/vcwh8w6ryn37...-libxslt-1.1.14
/nix/store/vfps6jpav2h6...-gnutar-1.15.1 [...]
```

shows the components in the store that use a particular instance of Glibc. Likewise, the command nix-store --query --referers-closure prints all paths that directly or indirectly refer to the given path.

5.3. Atoms

A basic operation on the Nix store is to add *atoms* to the store. Atoms are store objects that are not derived, that is, are not built by performing a store derivation. The foremost examples of atoms are sources referenced from derivations in Nix expressions (e.g., builder.sh in Figure 2.6), and store derivations.

The operation addToStore(*fso*, *refs*, *name*) shown in Figure 5.3 adds an FSO *fso* to the Nix store. The store path is computed from the cryptographic hash of the serialisation of *fso*, and the symbolic name *name*. The references entry for the resulting path is set to *refs*.

The operation works as follows. It computes the cryptographic hash over the serialisation of the FSO, and uses that and *name* to construct the store path p $\boxed{45}$. If path p is not valid $\boxed{47}$, the FSO is written to path p $\boxed{48}$. The path is marked as valid, with the references of p set to *refs* $\boxed{49}$. The references are set by the helper function setReferences $\boxed{50}$, which also updates the corresponding referrer mappings. The remainder of the code is concerned with concurrency.

The hash part of path p is essentially a cryptographic hash of the atom being added. Therefore there is a correspondence between the file name of the store object created by addToStore, and its contents. As a consequence, knowing the contents of an FSO (and its symbolic name) also tells one its store path. This property is known as *content-addressability*. It does not hold for all store objects: derivation outputs are not content-addressable, since their output paths are computed before they are built (as we will see in

```
addToStore($fso$, $refs$, $name$) :
    $c \leftarrow$ serialise($fso$)
    $h \leftarrow$ hash$_{\text{sha256}}$($c$)
    $p \leftarrow$ makePath("source", printHash$_{16}$($h$), $name$) $\boxed{45}$
    if valid[$p$] $= \varepsilon$ : $\boxed{46}$
        $lock \leftarrow$ acquirePathLock($p$)
        if valid[$p$] $= \varepsilon$ : $\boxed{47}$
            if pathExists($p$) :
                deletePath($p$)
            writePath($p$, $fso$) $\boxed{48}$
            in a database transaction : $\boxed{49}$
                valid[$p$] $\leftarrow h$
                setReferences($p$, $refs$)
        releasePathLock($p$, $lock$)
    return $p$

setReferences($p$, $refs$) $\boxed{50}$
    references[$p$] $\leftarrow refs$
    for each $p' \in refs$ :
        referers[$p'$] $\overset{\cup}{\leftarrow} \{p\}$
```

Figure 5.3.: addToStore: Adding atoms to the Nix store

Section 5.4). This is in fact the defining characteristic of the extensional model. In the intensional model described in Chapter 6, content-addressability is extended to *all* store objects.

Concurrency We have to take into account the remote possibility that parallel Nix processes simultaneously try to add the same atom. If this happens, the operations should be idempotent, and the contents of a path should never change after it has been marked as valid. There should be no race conditions, e.g.:

1. Process A detects that the target path p is invalid, and starts writing the atom to p.

2. Process B detects that p is invalid, and wants to write the atom to the target path, but there is an obstructing file at p. As it cannot make assumptions about invalid paths, it deletes p and starts writing the atom to p.

3. While B is half-way through, A finishes writing and marks the path as valid. Note that some of A's writes have been deleted.

4. Before B finishes writing, A starts the builder of a derivation that has p as input. The builder uses the incorrect contents of p.

There are several solutions to prevent races. The simplest one is to use a global mutex lock on the Nix store for all operations that modify the store. These operations are the

```
acquirePathLock($p$) :
  while true:
    $fd \leftarrow$ open($p +$ ".lock", O_WRONLY|O_CREAT, 0666)
    fcntl($fd$, F_SETLKW, ...F_WRLCK...) $\boxed{51}$
    if size of file $fd = 0$ : $\boxed{52}$
      return $fd$ $\boxed{53}$
    close($fd$)

releasePathLock($p$, $fd$) :
    deleteAndSignal($p +$ ".lock", $fd$)

deleteAndSignal($p$, $fd$) :
    deletePath($p$)
    write($fd$, "deleted") $\boxed{54}$
    close($fd$) $\boxed{55}$
```

Figure 5.4.: acquirePathLock and releasePathLock: Locking on Unix

addition of atoms, the building of derivations (Section 5.5), and garbage collection (Section 5.6). Every Nix process acquires the lock on startup, and releases it when finished. If the lock is already held, processes block until it becomes available. This approach is clearly unacceptable because of the unbounded latency it introduces for Nix operations.

A better solution is fine-grained locking, where a process acquires a mutex lock on the specific store path that it wants to add. The locking operations $lock \leftarrow$ acquirePathLock(p) and releasePathLock(p, *lock*) essentially acquire and release a lock on a temporary file p.lock, i.e., p with the extension .lock appended.

Figure 5.4 shows pseudo-code for an implementation of these operations on Unix. It uses the POSIX standard system call fcntl, which locks byte ranges in files [152]. The advantage of POSIX locks is that they are released automatically when the process holding them dies, either normally or abnormally. In contrast, the use of file *existence* as a mutex makes it harder to recover gracefully from abnormal termination and system crashes.

A mostly aesthetic complication is that we don't want the lock files to hang around indefinitely. They should be removed when they are no longer necessary, i.e., when p has been registered as being valid. Deleting the lock file after we have released the lock introduces a subtle race, however:

1. Process A creates the lock file p.lock and locks it.

2. Process B opens the lock file, attempts to acquire a lock, and blocks.

3. Process A sees that p is invalid, and begins to write to p. However, it is interrupted before it can make p valid.

4. As part of its cleanup, process A releases the lock, closes and deletes the lock file, and terminates. Note that B still has the file open; on Unix it is possible to delete open files.

5. Process B acquires the lock.

6. Process B sees that p is invalid, and begins to write to it.

7. Process C creates the lock file p.lock and locks it. This is a different file than the one currently opened by B—it has a different inode number. Thus C can acquire a lock.

8. Process C sees that p is invalid, and begins to write to it.

9. Now processes B and C are concurrently writing to p.

To prevent this race, we have to signal to process B that the lock file it has open has become "stale", and that it should restart. This is done by the function deleteAndSignal, called by the deleting process: it writes an arbitrary string (the *deletion token*) to the deleted file $\boxed{54}$. If another process subsequently acquires a lock $\boxed{51}$, but sees that the size of the file is non-zero $\boxed{52}$, then it knows that its lock is stale.

On Windows, there is a much simpler solution: we just try to delete the file, which will fail if it is currently opened by any other process. Thus, the file will be deleted by the last process to release the lock, provided no new lockers have opened the lock file.

The dual checks for path validity are an optimisation ($\boxed{46}$, $\boxed{47}$). Strictly speaking, the initial path validity check $\boxed{46}$ can be omitted. However, the extra check prevents a lock creation and acquisition in the common case where the path is already valid.

5.4. Translating Nix expressions to store derivations

Nix does most of its work on store derivations, introduced in Section 2.4. Store derivations are the result of the evaluation of high-level Nix expressions. Nix expressions can express variability, store derivations cannot—a store derivation encodes a single, specific, constant build action.

The command nix-instantiate performs precisely this translation: it takes a Nix expression and evaluates it to normal formal using the expression semantics presented in Section 4.3.4. The normal form should be a call to derivation, or a nested structure of lists and attribute sets that contain calls to derivation. In any case, these derivation Nix expressions are subsequently translated to store derivations using the method described in this section.

The resulting store derivations can then be built, as described in the next section. The command nix-store --realise does exactly that. The high-level package management command nix-env combines translation and building as a convenience to users.

The abstract syntax of store derivations is shown in Figure 5.5 in a Haskell-like [135] syntax (see Section 1.7). The store derivation example shown in Figure 2.13 is a value of this data type. The outputHash and outputHashAlgo fields implement so-called *fixed-output derivations* and are discussed in Section 5.4.1. The other fields were explained in Section 2.4.

So suppose that we have an expression e that evaluates to derivation e', and e' evaluates to an attribute set $\{as\}$. According to the DERIVATION and DERIVATION! rules (page 80), e then (essentially) evaluates to the result of calling instantiate(as). instantiate is a function that translates the argument attribute set of a call to derivation to a store derivation, writes

```
data StoreDrv = StoreDrv {
    output : Path,
    outputHash : String,
    outputHashAlgo : String,
    inputDrvs : [Path],
    inputSrcs : [Path],
    system : String,
    builder : Path,
    args : [String],
    envVars : [(String, String)]
}
```

Figure 5.5.: Abstract syntax of store derivations

as a side-effect the store derivation to the Nix store[5], and returns the original attribute set, with the following three attributes added:

- drvPath contains the path of the store derivation.
- outPath contains the output path of the store derivation, that is, d.output.
- type is set to the value "derivation". This allows instantiate to detect that attribute sets occurring in its arguments are actually subderivations. It also allows nix-instantiate and nix-env to see whether evaluation results are actually derivations.

The function instantiate is shown in Figure 5.6. Its main job is to construct a store derivation $\boxed{56}$ which is written to the Nix store using addToStore $\boxed{65}$. The hard part is in filling in the fields of the store derivation from the attributes *as*. In particular, we need to compute the following bits of information:

- We need an output path (the field output). The output path reflects all information that goes into the derivation. Therefore we initially leave output empty $\boxed{57}$, then compute a cryptographic hash over this initial derivation and use makePath to construct the actual path $\boxed{64}$. The hash is computed using the function hashDrv, which conceptually just computes a SHA-256 hash over a concrete syntax representation of the store derivation. However, the notion of fixed-output derivations complicates matters a bit. The actual operation of hashDrv is discussed below. The final output path is also placed in the out environment variable, in order to communicate the path to the builder.
- We need to compute the environment variable bindings envVars $\boxed{63}$. envVars is a set of name/value tuples. Each attribute $\langle n = e \rangle$ in $\{as\}$ is mapped to an environment variable binding. This translation is done by the function processBinding shown in Figure 5.7. It takes e and returns, among other things, a string representation of

[5] Aficionados of purely functional programming languages may be dismayed by the impurity of producing a store derivation as a side-effect. However, the side-effect is not observable in the Nix expression evaluation, and the idempotency of the operation ensures that equational reasoning still holds.

```
instantiate($as$) :
    $name$ ← eval($as$.name)
    if $name$ ends in ".drv" : abort
    if $name$ contains invalid characters : abort
    $d$ ← StoreDrv { $\boxed{56}$
        output = "" $\boxed{57}$
        outputHash = eval($as$.outputHash) if attr. exists,   "" otherwise
        outputHashAlgo = "" $\boxed{58}$
            ("r:" if eval($as$.outputHashMode) = "recursive", "" otherwise) +
            (eval($as$.outputHashAlgo) if attr. exists,   "" otherwise)
        inputDrvs = {processBinding($e$).drvs | $\langle n = e \rangle \in as$} $\boxed{59}$
        inputSrcs = {processBinding($e$).srcs | $\langle n = e \rangle \in as$} $\boxed{60}$
        system = eval($as$.system) // Must evaluate to a string.
        builder = concSp(processBinding($as$.builder).res) $\boxed{61}$
        args = map($\lambda e$ . processBinding($e$).res, $as$.args) $\boxed{62}$
        envVars = {($n$, concSp(processBinding($e$).res)) | $\langle n = e \rangle \in as$} $\boxed{63}$
          ∪ {(out, "")}
    }
    $d$.output ← makePath("output:out", hashDrv($d$), $name$) $\boxed{64}$
    $d$.$envVars$["out"] ← $d$.output
    $p$ ← addToStore(printDrv($d$), $d$.inputDrvs ∪ $d$.inputSrcs, $name$ + ".drv") $\boxed{65}$
    return {outPath = $d$.output, drvPath = $p$}
```

Figure 5.6.: instantiate: Instantiation of store derivations from a Nix expression

the normal form of e. processBinding is discussed in more detail below. However, it should be pointed out here that processBinding returns a *list* of strings, since attribute values can be lists of values. For instance, it is common to communicate a list of paths of components to a builder:

```
buildInputs = [libxml2 openssl zlib];
```

Since the value of an environment variable is a plain string, the list of strings returned by processBinding is concatenated with spaces separating the elements; e.g., "/nix/store/ngp50703j8qn...-libxml2-2.6.17⊔/nix/store/8ggmblhvzciv...-openssl-0.9.7f⊔/nix/store/kiqnkwp5sqdg...-zlib-1.2.1" where ⊔ denotes a space.

- Likewise, processBinding is used to set the command-line arguments from the args attribute $\boxed{62}$, and the builder from the builder attribute $\boxed{61}$. Since args is a list, we do not need to concatenate the strings returned by processBindings.

- The system field is initialised from the system attribute.

- The set of input derivations inputDrvs (the build-time dependencies) is the set of store paths of derivations that occur in the attributes $\boxed{59}$. These are discovered by processBinding.

```
processBinding($e$) :
    $e' \leftarrow$ eval($e$)
    if $e' =$ true :
        return $\{$drvs $= \emptyset$, srcs $= \emptyset$, res $=$ ["1"]$\}$
    if $e' =$ false :
        return $\{$drvs $= \emptyset$, srcs $= \emptyset$, res $=$ [""]$\}$
    if $e' =$ a string $s$ :
        return $\{$drvs $= \emptyset$, srcs $= \emptyset$, res $= [s]\}$
    if $e' =$ null :
        return $\{$drvs $= \emptyset$, srcs $= \emptyset$, res $=$ [""]$\}$
    if $e' =$ a path $p$ : $\boxed{66}$
        $p' \leftarrow$ addToStore(readPath($p$), $\emptyset$, baseName($p$))
        return $\{$drvs $= \emptyset$, srcs $= \{p'\}$, res $= [p']\}$
    if $e' =$ an attribute set $\{as\}$ with $as$.type = "derivation" : $\boxed{67}$
        // Note that $\{as\}$ is the result of a call to derivation.
        return $\{$drvs $= \{$eval($as$.drvPath)$\}$, srcs $= \emptyset$, res $= [$eval($as$.outPath)$]\}$
    if $e' =$ a list $[ls]$ : $\boxed{68}$
        $ps \leftarrow$ map(processBinding, $ls$)
        return
          $\{$ drvs $= \bigcup_{p \in ps} p$.drvs, srcs $= \bigcup_{p \in ps} p$.srcs
          , res $=$ concatMap($\lambda p$ . $p$.res, $ps$)$\}$
    abort
```

Figure 5.7.: processBinding: Processing derivation attributes

- The set of input sources inputSrcs is the set of store paths of sources that occur in the attributes $\boxed{60}$. These are also discovered by processBinding.

Processing the attributes Clearly, the core of the translation is the function processBinding, shown in Figure 5.7. It evaluates a Nix expression, inspects it, and returns three pieces of information in a structure:

- A list of strings that represent the normal form of the expression (res).
- The set of store paths of derivations occurring in the expression (drvs).
- The set of store paths of sources occurring in the expression (srcs).

Simple values like strings, Booleans, and null map to singleton string lists. For instance, true is translated to ["1"], and false is translated to [""]. The latter is so that we can write in shell scripts:

```
if test -n "$var"; then ... fi
```

to test whether Boolean variable $var is set.

For lists, processBinding is applied recursively to the list elements $\boxed{68}$. The resulting string lists are concatenated into a single list. Thus, nested lists are allowed; they are flattened automatically.

For paths, processBinding copies the referenced FSO to the Nix store using addToStore and returns the resulting path in srcs and res $\boxed{66}$. Thus all referenced sources end up in the Nix store. The file name of the source (returned by the function baseName) is used as the symbolic name of the store path.

For attribute sets, processBinding checks whether the type attribute of the derivation (if present) is equal to derivation. If so, the value denotes the result of a call to derivation. This is where instantiate recurses into subderivations: instantiate calls processBinding, which calls eval, which eventually calls instantiate for derivations occurring in the input attributes.

Let us look at an example. Suppose that we have a derivation

```
args = ["-e" ./builder.sh python];
```

where python is a variable that evaluates to another derivation. Since this is a list, processBinding will recurse into the list elements. This yields the following results. For "-e", we get

$$\{\mathsf{drvs} = \emptyset, \mathsf{srcs} = \emptyset, \mathsf{res} = [\text{"-e"}]\}.$$

For ./builder.sh, the file builder.sh in the current directory is added to the Nix store using addToStore. Supposing that the file has SHA-256 hash 49b96daeab53... in hexadecimal, addToStore will call makePath with the following arguments:

$$\mathsf{makePath}(\text{"source"}, \text{"49b96daeab53..."}, \text{"builder.sh"}).$$

makePath constructs the hash part by hashing and truncating the following string

```
source:sha256:49b96daeab53...:/nix/store:builder.sh
```

into 043577cf3vlb.... The store path thus becomes /nix/store/043577cf3vlb...-builder.sh, and processBinding returns

$$\begin{aligned}&\{\ \mathsf{drvs} = \emptyset, \mathsf{srcs} = \{\text{/nix/store/043577cf3vlb...-builder.sh}\}\\ &\ ,\ \mathsf{res} = [\text{"/nix/store/043577cf3vlb...-builder.sh"}]\ \}.\end{aligned}$$

For python, if we suppose that it evaluates to an attribute set with drvPath = /nix/store/-mi4p0ck4jqds...-python-2.4.1.drv and outPath = /nix/store/s3dc4m2zg9bb...-python-2.4.1, processBinding returns

$$\begin{aligned}&\{\ \mathsf{drvs} = \{\text{/nix/store/mi4p0ck4jqds...-python-2.4.1.drv}\}, \mathsf{srcs} = \emptyset\\ &\ ,\ \mathsf{res} = [\text{"/nix/store/s3dc4m2zg9bb...-python-2.4.1"}]\ \}.\end{aligned}$$

Note that the path of the store derivation is placed in drvs (and will therefore eventually end up in the inputDrvs of the referring derivation), since the derivation is needed to recursively build this dependency when we build the current derivation. On the other hand, the path of the output of the derivation is placed in res (and will end up in an environment variable or command-line argument), since that is what the builder is interested in.

```
Derive(
  [("out","/nix/store/bwacc7a5c5n3...-hello-2.1.1","","")],
  [("/nix/store/7mwh9alhscz7...-bash-3.0.drv",["out"]),
   ("/nix/store/fi8m2vldnrxq...-hello-2.1.1.tar.gz.drv",["out"]),
   ("/nix/store/khllx1q519r3...-stdenv-linux.drv",["out"]),
   ("/nix/store/mjdfbi6dcyz7...-perl-5.8.6.drv",["out"])],
  ["/nix/store/d74lr8jfsvdh...-builder.sh"],
  "i686-linux",
  "/nix/store/3nca8lmpr8gg...-bash-3.0/bin/sh",
  ["-e","/nix/store/d74lr8jfsvdh...-builder.sh"],
  [("builder","/nix/store/3nca8lmpr8gg...-bash-3.0/bin/sh"),
   ("name","hello-2.1.1"),
   ("out","/nix/store/bwacc7a5c5n3...-hello-2.1.1"),
   ("perl","/nix/store/h87pfv8klr4p...-perl-5.8.6"),
   ("src","/nix/store/h6gq0lmj9lkg...-hello-2.1.1.tar.gz"),
   ("stdenv","/nix/store/hhxbaln5n11c...-stdenv-linux"),
   ("system","i686-linux")] )
```

Figure 5.8.: ATerm representation of the Hello derivation

Finally, the results of the recursive calls are combined for the list, and we get

{ drvs = {/nix/store/mi4p0ck4jqds...-python-2.4.1.drv}
, srcs = {/nix/store/043577cf3vlb...-builder.sh}
, res = ["-e" "/nix/store/043577cf3vlb...-builder.sh"
"/nix/store/s3dc4m2zg9bb...-python-2.4."] }.

Writing the derivation When the final output path of the derivation has been computed, we write a string representation of the store derivation to the Nix store using addToStore $\boxed{65}$. The function printDrv returns a byte sequence that represents the store derivation. The contents of the byte sequence is a *textual ATerm*. ATerms [166] are a simple format for the exchange of terms. For the Hello example, the ATerm representation is shown in Figure 5.8[6].

I will not formalise the translation to ATerms here, which is straightforward. However, the reader may be wondering about the presence of the "out" strings in the ATerm encoding of the inputDrvs field, as these do not occur in the abstract syntax, and about the fact that output is a *list* here, also containing the mysterious "out" string. This is actually for future compatibility with a Nix semantics that supports multiple output paths. Every derivation would produce a set of *labelled* outputs, the default output having label out. Multiple outputs allow more fine-grained deployment. In Section 6.7, I show a semantics that supports this.

Note that in the call to addToStore, we set the references of the new store object to the union of the store derivations (inputDrvs) and sources (inputSrcs) used by the derivation. Thus, the references graph is maintained for store derivations. This means that we can query the closure of a derivation, which is the set of files necessary to do a build of the

[6] Whitespace has been added to improve legibility. Actual textual ATerms as produced by the function ATwriteToString() in the ATerm API do not contain whitespace. In other words, ATwriteToString() produces a canonical representation of an ATerm, and thus of the store derivation it represents.

derivation and all its dependencies. As we saw in Section 2.4, this is useful for source deployment.

On the other hand, the path in the output field is not a reference of the store derivation, since the output in general does not exist yet, and because we want garbage collection of derivations and outputs to be independent.

5.4.1. Fixed-output derivations

The only remaining aspect of the translation is the computation of the output path by hashDrv. As stated above, conceptually the output path is constructed from a SHA-256 hash over the ATerm representation of the initial store derivation (i.e., the one with output set to the empty string $\boxed{57}$). However, the matter is complicated by the notion of fixed-output derivations.

Fixed-output derivations are *derivations of which we know the output in advance*. More precisely, the cryptographic hash of the output path is known to the Nix expression writer.

The rationale for fixed-output derivations is derivations such as those produced by the fetchurl function. This function downloads a file from a given URL. To ensure that the downloaded file has not been modified, the caller must also specify a cryptographic hash of the file. For example,

```
fetchurl {
  url = http://ftp.gnu.org/pub/gnu/hello/hello-2.1.1.tar.gz;
  md5 = "70c9ccf9fac07f762c24f2df2290784d";
}
```

It sometimes happens that the URL of the file changes, e.g., because servers are reorganised or no longer available. We then must update the call to fetchurl, e.g.,

```
fetchurl {
  url = ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
  md5 = "70c9ccf9fac07f762c24f2df2290784d";
}
```

If a fetchurl derivation followed the normal translation scheme, the output paths of the derivation and *all derivations depending on it* would change. For instance, if we were to change the URL of the Glibc source distribution—a component on which almost all other components depend—massive rebuilds will ensue. This is unfortunate for a change which we know cannot have a real effect as it propagates upwards through the dependency graph.

Fixed-output derivations solve this problem by allowing a derivation to state to Nix that its output will hash to a specific value. When Nix builds the derivation (Section 5.5), it will hash the output and check that the hash corresponds to the declared value. If there is a hash mismatch, the build fails and the output is not registered as valid. For fixed-output derivations, the computation of the output path only depends on the declared hash and hash algorithm, *not* on any other attributes of the derivation.

Figure 5.9 shows the definition of the fetchurl function. The expression presented here is somewhat simplified from the actual expression in Nixpkgs, which accepts SHA-1 and SHA-256 hashes in addition to MD5 hashes. A fixed-derivation output is declared by defining the following attributes:

```
{stdenv, curl}: # The ‘curl’ program is used for downloading.

{url, md5}:

stdenv.mkDerivation {
  name = baseNameOf (toString url);
  builder = ./builder.sh;
  buildInputs = [curl];

  # This is a fixed-output derivation;
  # the output has to have MD5 hash ‘md5’.
  outputHashMode = "flat";
  outputHashAlgo = "md5";
  outputHash = md5;

  inherit url;
}
```

Figure 5.9.: pkgs/build-support/fetchurl/default.nix: fixed-output derivations in fetchurl

- The attribute outputHashAlgo specifies the hash algorithm: outputHashAlgo $\in$ {"md5","sha1","sha256"}.

- The attribute outputHash specifies the hash in either hexadecimal or base-32 notation, as autodetected based on the length of the string.

- The attribute outputHashMode $\in$ {"recursive","flat"} states whether the hash is computed over the serialisation of the output path p, i.e., hash(serialise(readPath(p))), or over the contents of the single non-executable regular file at p, respectively. The latter (which is the default) produces the same hashes as standard Linux commands such as md5sum and sha1sum.

  Recursive mode is useful for tools that download directory hierarchies instead of single files. For instance, Nixpkgs contains a function fetchsvn that exports revisions of directory hierarchies from Subversion repositories, e.g.,

  ```
  fetchsvn {
    url = https://svn.cs.uu.nl:12443/repos/trace/nix/trunk;
    rev = 3297;
    md5 = "2a074a3df23585c746cbcae0e93099c3";
  }
  ```

  For historical reasons, the outputHashMode attribute is encoded in the outputHashAlgo field $\boxed{58}$.

These three attributes are used in the output path computation. This means that, for instance, the url attribute in fetchurl derivation is disregarded. Thus, instantiation of the two fetchurl calls shown above produces two store derivations that have the same output field.

```
hashDrv($d$) :
  if $d$.outputHash $\neq$ ""
    return hash$_{\text{sha256}}$("fixed:out:" $+$ $d$.outputHashAlgo
      $+$":"$+$ $d$.outputHash
      $+$":"$+$ $d$.output)
  else :
    $d' \leftarrow d$ { // I.e., $d'$ is a modified copy of $d$
      inputDrvs $=$ {hashDrv(parseDrv(readPath($p$))) | $p \in d$.inputDrvs} )$\boxed{69}$
    }
    return hash$_{\text{sha256}}$(printDrv($d'$))
```

Figure 5.10.: hashDrv: Hashing derivations modulo fixed-output derivations

So how do we compute the hash part of the output path of a derivation? This is done by the function hashDrv, shown in Figure 5.10. It distinguishes between two cases. If the derivation is a fixed-output derivation, then it computes a hash over just the outputHash attributes[7].

If the derivation is *not* a fixed-output derivation, we replace each element in the derivation's inputDrvs with the result of a call to hashDrv for that element. (The derivation at each store path in inputDrvs is converted from its on-disk ATerm representation back to a StoreDrv by the function parseDrv.) In essence, hashDrv partitions store derivations into equivalence classes, and for hashing purpose it replaces each store path in a derivation graph with its equivalence class.

The recursion in Figure 5.10 is inefficient: it will call itself once for each path by which a subderivation can be reached, i.e., $O(V^k)$ times for a derivation graph with V derivations and with out-degree of at most k. In the actual implementation, memoisation is used to reduce this to $O(V+E)$ complexity for a graph with E edges.

5.5. Building store derivations

After we have translated a Nix expression to a graph of store derivations, we want to perform the build actions described by them. The command nix-store --realise does this explicitly, and nix-env and nix-build do it automatically after store derivation instantiation.

The postcondition of a successful build of a derivation is that the output path of the derivation is valid. This means that a successful derivation is performed only once (at least until the user runs the garbage collector). If the output is already valid, the build is a trivial operation. If it is not, the build described by the derivation is executed, and on successful completion the output path is registered as being valid, preventing future rebuilds.

The build algorithm is also where binary deployment using substitutes is implemented. If we want to build a derivation with output path p, p is not valid, and we have a substitute for p, then we can execute the substitute to create p (which the substitute will generally

[7] ...as well as the output field. This field is empty when hashDrv is called for "unfinished" derivations from instantiate $\boxed{64}$, but it contains an actual path when hashDrv calls itself recursively for "finished" derivations $\boxed{69}$. This ensures that the name attribute and the location of the store are taken into account.

do by downloading a pre-built binary and unpacking it to p). Substitution is discussed in Section 5.5.3; here we just assume that there exists a function substitute(p) : Bool that returns true if path p is valid or it has managed to make p valid by executing a substitute, and false otherwise.

This section shows how Nix builds store derivations. It first presents a "simple" build algorithm that recursively builds store derivations. Next, it introduces the concept of *build hooks*, which are a mechanism to support distributed multi-platform builds in a transparent and centralised manner. Finally it shows a build algorithm that can perform multiple derivations in parallel.

5.5.1. The simple build algorithm

The build algorithm is shown in Figure 5.11. The function build takes as its sole argument the store path p_{drv} of the derivation to be built. It returns the output path of the derivation if the build succeeds, and aborts if it does not. It consists of the following main steps:

$\boxed{70}$ First, we need to ensure that the derivation p_{drv} exists. If it does not, it might be possible to obtain it through a substitute. It is a fatal error if p_{drv} cannot be made valid. In any case, when p_{drv} is valid, it is read into d, a value of the type StoreDrv shown in Figure 5.5.

While the primary purpose of substitutes is to speed up the construction of derivation outputs, they may also be used to distribute the store derivations themselves. In general, what we distribute to client machines are Nix expressions, and the translation of those will cause the store derivations to be created locally. However, it is also possible to forego Nix expressions entirely and work directly on store derivations. For instance, nix-env allows one to install a store derivation directly:

```
$ nix-channel --update
$ nix-env -i /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv
```

If /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv is not valid, it is possible that a substitute for it is available (e.g., obtained from a subscribed channel). The primary advantage of skipping Nix expressions in this manner is that it makes one independent of the evolution of the Nix expression language, removing a possible source of incompatibility between the distributor and the clients. This is in fact one of the main advantages of the two-step build process.

$\boxed{71}$ We try to substitute the output path of the derivation. If it is already valid (as stated above, substitute checks for this trivial case), or it can be produced through a substitute, we're done right away.

$\boxed{72}$ Since the output is not valid and there is no substitute, we have to perform the build action. We need to build all input derivations in inputDrvs, or, more precisely, we need to ensure that the output paths of the input derivations are valid. So we recursively build the input derivations.

Note that it is *not* necessary to ensure that the input sources in inputSrcs are valid. This is because of the closure invariant: the inputSrcs are all in the references

```
build($p_{drv}$) :
    if $\neg$ substitute($p_{drv}$) : abort $\boxed{70}$
    $d \leftarrow$ parseDrv(readPath($p_{drv}$))
    if substitute($d$.output) : return $d$.output $\boxed{71}$

    $inputs \leftarrow \emptyset$
    for each $p \in d$.inputsDrvs : $\boxed{72}$
        build($p$)
        $d' \leftarrow$ parseDrv(readPath($p$))
        $inputs \overset{\cup}{\leftarrow}$ closure($d'$.output) $\boxed{73}$
    for each $p \in d$.inputsSrcs :
        $inputs \overset{\cup}{\leftarrow}$ closure($p$) $\boxed{74}$

    $lock \leftarrow$ acquirePathLock($d$.output) $\boxed{75}$
    if valid[$d$.output] $\neq \varepsilon$ :
        releasePathLock($d$.output, $lock$)
        return $d$.output
    if $d$.system $\neq$ thisSystem : abort $\boxed{76}$
    if pathExists($d$.output) : deletePath($d$.output) $\boxed{77}$
    $p_{tmp} \leftarrow$ createTempDir() $\boxed{78}$
    Run $d$.builder in an environment $d$.envVars
      and with arguments $d$.args and in directory $p_{tmp}$ $\boxed{79}$
    if the builder exits with exit code $\neq 0$ :
        deletePath($d$.output)
        abort

    canonicalisePathContents($d$.output) $\boxed{80}$
    if $d$.outputHash $\neq$ "" : $\boxed{81}$
        Extract mode $m$ and hash algorithm $t$ from d.outputHashAlgo
        if $m$ indicates flat mode :
            Check that $d$.output is a non-executable regular file
              and that hash$_t$(readFile($d$.output)) matches $d$.outputHash
        else :
            Check that hash$_t$(serialise(readPath($d$.output))) matches $d$.outputHash

    In a database transaction : $\boxed{82}$
        valid[$d$.output] $\leftarrow$ hash$_{\mathsf{sha256}}$(serialise(readPath($d$.output)))
        setReferences($d$.output, scanForReferences($d$.output, $inputs$))
        deriver[$d$.output] $\leftarrow p_{drv}$
    releasePathLock($d$.output, $lock$)
    return $d$.output
```

Figure 5.11.: build: Building store derivations

entry for p_{drv} (cf. Figure 5.6), and thus validity of p_{drv} implies validity of each $p \in$ d.inputSrcs. Likewise, we know for sure that we can load the input derivations, since these too are references of p_{drv}.

$\boxed{75}$ We acquire an exclusive lock on the output path. Once we have acquired the lock, we know that there is no other Nix process building the output path (including through substitutes, since as we will see, substitute also locks properly).

However, once we have acquired the lock, it is possible that another process got there first, i.e., saw that the output was invalid, acquire the lock, and did the build. This can be detected by checking once more whether the output path is valid. If it is, we're done.

Thanks to locking and the transactional semantics of registering valid paths (step $\boxed{82}$ described below), an interrupted operation (such as building a derivation) can always be restarted, requiring no further recovery. It also ensures that all operations are idempotent, i.e., can be repeated any number of times. Operations can also be run in parallel. In particular, it is safe to run multiple build actions in parallel. Since locking is fine-grained, as opposed to having a global lock for the entire Nix store, parallel execution can be used to speed up builds. If more than one CPU is available, this gives better resource utilisation. However, even if there is only one CPU, we may get a performance improvement if some of the builds are heavily I/O-bound [6].

$\boxed{76}$ It is only possible to build the derivation if the current machine and operating system match the system field of the derivation. A Nix installation on an i686-linux machine cannot build a derivation for a powerpc-darwin machine. The platform type of the running Nix instance is denoted by thisSystem. However, Section 5.5.2 describes an extension to the build algorithm that enables distributed multi-platform builds.

$\boxed{77}$ It is possible that the output path already exists, even though it is not valid. Typically, this is because a previous build was interrupted before the builder finished or before the output could be registered as valid. Such a residual output can interfere with the build, and therefore must be deleted first.

$\boxed{78}$ Builds are executed in a temporary directory p_{tmp} that is created here (typically, it is a fresh subdirectory of /tmp). p_{tmp} is deleted automatically after the builder finishes (not shown here). The temporary directory provides a place where the builder can write scratch data. For instance, the unpacking and compiling of the Hello sources performed by the Hello builder in Figure 2.7 is done in the temporary directory. The fact that the temporary directory is initially empty reduces the possibility of interference due to files pre-existing in the builder's initial current directory.

$\boxed{79}$ Here we actually perform the build by running the executable indicated by the builder field of the derivation, with the command-line arguments args and the environment variables envVars, and with the current directory set to p_{tmp}. Note that envVars specifies the complete environment; the builder does not inherit any environment variables from the caller of the Nix process user. This prevents interference from environment variables such as PATH.

In the real implementation some additional variables are initialised to specific values. The variable TMPDIR is set to p_{tmp} so that tools that need temporary space write to that directory, reducing the possibility of interference, left-over files, and security problems due to /tmp races. The variable HOME, which points to the user's home directory, is set to the meaningless value /homeless-shelter. Many programs look up the home directory corresponding to the current user ID in /etc/passwd or similar *if* HOME is not set. Setting HOME prevents such lookups and reduces the possibility of interference from tools that read configuration files from the home directory. Likewise, PATH is set to the value /path-not-set, since the Unix shell initialises PATH to an undesirable default such as /usr/local/bin:/bin:/usr/bin, which typically causes interference. Finally, the variable NIX_STORE is set to *storeDir*. Some tools need the location of the store for various purposes (see, e.g., Section 7.1.3).

80 If the build finishes successfully, the output is *canonicalised* by the function canonicalisePathContents. It sets all file system meta-information not represented in the canonical serialisation to fixed values. In particular, it does the following:

- The "last modified" timestamps on files are set to 0 seconds in the Unix epoch, meaning 1/1/1970 00:00:00 UTC.
- The permission on each file is set to octal 0444 or 0555, meaning that the file is readable by everyone and writable by nobody, but possibly has execute permission. Removing write permission prevents the path from being accidentally modified when it is used as an input to other derivations.
- Special permissions, such as set-user-ID and set-group-ID [152], are removed.
- A fatal error is flagged if a file is found that is not a regular file, a directory, or a symlink.

The actual implementation of canonicalisePathContents(p) is not shown here, but in its effect is equal to deserialising a canonical serialisation of the path:

$$\mathsf{writePath}(p,\ \mathsf{readPath}(p))$$

where writePath takes care to create files with canonical values for meta-information not represented in the canonical serialisation.

81 If d is a fixed-output derivation (Section 5.4.1), then we have to check that the output produced by the builder matches the cryptographic hash declared in the derivation. Recall that there are two modes of fixed-output derivation: flat mode that requires the output to be single non-executable file, and recursive mode that applies to arbitrary outputs. Flat mode hashes the plain file contents (read by the function readFile), while recursive mode hashes the serialisation of the entire FSO.

82 Since the build has finished successfully, we must register the path as valid and set its references mapping. This must be done in a database transaction to maintain the invariants. Thanks to the ACID properties of the underlying Berkeley DB database, it is always safe to interrupt any Nix operation, and Nix can always recover gracefully

```
scanForReferences($p$, $paths$) :
    $c \leftarrow$ serialise(readPath($p$))
    $refs \leftarrow \emptyset$
    for each $p' \in paths$ :
        if find($c$, hashPart($p'$)) $\neq \emptyset$ :
            $refs \overset{\cup}{\leftarrow} \{p'\}$
    return $refs$
```

Figure 5.12.: scanForReferences: Scanning for store path references in build results

from non-corrupting crashes, including power failures. In the case of operating system or hardware crashes, this requires the file system containing the Nix store to be data-journaled [142], or to otherwise guarantee that once data has been committed to disk, no rollback to previous contents will occur.

Following the stored hash invariant (page 96), the valid entry for the output path is set to the SHA-256 of the serialisation of the path contents.

The references entry is set to the set of store paths referenced from the output contents. As described in Chapter 3, the references are found by scanning for the hash parts of the input paths. The set of input paths is the union of the closures of the input sources $\boxed{74}$ and the outputs of the input derivations $\boxed{73}$. Note that it is quite possible for the output to contain a reference to a path that is not in inputSrcs or inputDrvs, but that *is* in the closure of those paths. Such a reference can be obtained by the builder indirectly by dereferencing one of its immediate inputs. Given the full set of input paths, the function scanForReferences returns the set of referenced paths.

The deriver mapping maintains a link from outputs to the derivations that built them. It is described below.

Scanning for references The function scanForReferences, shown in Figure 5.12, determines the references in a path p to other store paths.

We only need to scan for store paths that are inputs, i.e., that are in the closure of inputSrcs and the outputs of inputDrvs. It is not necessary to scan for references to arbitrary store paths, since the only way that a builder can introduce a reference to a path not in the input set, is if it were to read the Nix store to obtain arbitrary store paths. While it is easy to construct a contrived builder that does this, builders do not do so in practice. Indeed, if this were an issue, we could block such behaviour by making the Nix store unreadable but traversable (i.e., by clearing read permission but enabling execute permission on the directory on Unix systems).

Thus scanForReferences also takes a set of potentially referenced paths. It then searches for the hash part of each path in the serialisation of the contents of the FSO at p. To reiterate from Section 3.4, we only care about the hash part because that's the most easily identifiable part of a store path, and least likely to be hidden in some way. The search for the hash part is done using an ordinary string searching algorithm, denoted as find(s,t), which yields the integer offsets of the occurrences of the byte sequence t in the byte sequence s (an empty set indicating no matches). If and only if there is a match, we conclude that path p has a reference to the store path p'.

For the string search, the Boyer-Moore algorithm [18] is particularly suited because the hash parts consist of a small subset of the set of byte values, allowing the algorithm to jump through the input quickly in many cases. For instance, if we encounter any byte not in the set of base-32 characters (see digits_{32} in Section 5.1), we can immediately skip 32 characters. Especially for binary files this will happen quite often. Thus it is reasonable to expect that the search achieves Boyer-Moore's $\Theta(n/m)$ average running time in most cases (where n is the length of the serialisation in bytes, and m is the length of the hash part).

If it is desirable to support encodings of the hash part other than plain ASCII (and to date, it has not been necessary to do), scanForReferences can be adapted appropriately. For instance, to detect UTF-16 encodings of the hash part, we should just scan for the byte sequence $s[0]\ 0\ s[1]\ 0\ s[2]\ 0\ \ldots\ s[31]\ 0$, where $s = \mathsf{hashPart}(p'), p' \in paths$, and 0 denotes a byte with value 0[8].

Traceability For many applications it is important to query which derivation built a given store path (if any). Section 11.1 gives the example of *blacklisting*, which allows users to determine whether the build graph of a component contains certain "bad" sources. In Software Configuration Management, this is known as *traceability*—the ability to trace derived artifacts back to the source artifacts from which they were constructed.

To enable traceability, Nix maintains a database table $\mathsf{deriver} : \mathsf{Path} \to \mathsf{Path}$ that maps output paths built by derivations to those derivations. The following invariant maintains traceability.

Invariant 5 (Traceability invariant)**.** *The deriver of a valid path p, if defined, must be a valid store derivation with output path p.*

$$\begin{aligned}&\forall p \in \mathsf{Path} : (\mathsf{valid}[p] \neq \varepsilon \land \mathsf{deriver}[p] \neq \varepsilon) \to \\ &\quad (\mathsf{valid}[\mathsf{deriver}[p]] \neq \varepsilon \land \mathsf{parseDrv}(\mathsf{readPath}(\mathsf{deriver}[p])).\mathsf{output} = p)\end{aligned}$$

However, this invariant is actually optional in the current implementation, because not all users want traceability. The disadvantage of full traceability is that the entire derivation graph of each output path has to be kept in the store for the lifetime of those paths.

It is worth noting at this point that Nix in no way maintains a link between the build graph (store derivations and sources) in the Nix store on the one hand, and the Nix expressions from which they were instantiated on the other hand. If the latter are under version management control, there is no way to trace store paths back to the artifacts in the version management system from which they ultimately originated. This is most certainly a desirable feature, but it is not clear how such information can be maintained without coupling Nix too closely to the version management system (as is the case in Vesta and ClearCase, which integrate version management and build management, as discussed in Section 7.6).

[8] This assumes a little-endian encoding of UTF-16 (i.e., *UTF-16LE*) [34]. For the big-endian variant (*UTF-16BE*), the 0s should prefix the hash characters.

```
let {
  pkgsLinux = ...;
  pkgsDarwin = ...;

  fooC = derivation {
    name = "fooC";
    system = "i686-linux";
    src = ./foo.tar.gz;
    builder = ...;
    inherit (pkgsLinux) stdenv ... tiger;
  };

  fooBin = derivation {
    name = "foo";
    system = "powerpc-darwin";
    src = fooC;
    builder = ...;
    inherit (pkgsDarwin) stdenv ...;
  };

  body = fooBin;
}
```

Figure 5.13.: Example of a distributed, two-platform build

5.5.2. Distributed builds

The build algorithm in Figure 5.11 builds derivations only for the current platform (thisSystem). Through a relatively simple extension called *build hooks*, Nix supports distributed multi-platform builds.

Figure 5.13 shows a hypothetical example of a Nix expression that requires a distributed multi-platform capable build system. The scenario is that we want to compile a package foo, written in a hypothetical language called Tiger, for Mac OS X (powerpc-darwin). Unfortunately, the Tiger compiler does not run on that platform, but it does run on Linux (i686-linux) and can compile to C code. Thus, we have two derivations. The first derivation compiles the foo package to C code on i686-linux, and produces an output consisting of C source files. The second derivation compiles the C sources produced by the first derivation to native PowerPC code on powerpc-darwin. With the build hook mechanism, we can just build the second derivation on any machine normally, e.g.,

```
$ nix-env -f foo.nix -i foo
```

and any derivation that cannot be executed on the local machine is forwarded to a machine of the appropriate type. In this way, build hooks allow us to abstract over multi-platform builds. It is not necessary for the user to interact explicitly with the various machines involved in the build. In Section 9.4 we will see a powerful application of this: building distributed multi-platform services.

Build hooks are useful not just for multi-platform builds, but also to accelerate large single-platform builds. For instance, Nixpkgs contains hundreds of components that need to be built for i686-linux, and many of those derivations can be scheduled in parallel since they do not depend on each other. The build hook mechanism allows as many derivations as possible to run in parallel on the available hardware of that type.

A build hook is a user-defined program (typically written in a scripting language like Perl) that is called by Nix to perform the build on behalf of Nix. The build hook mechanism is policy-free, that is, it can do anything it wants, but a build hook typically does the following things:

- It determines whether it knows a remote machine of the appropriate platform type that can handle the derivation. If not, it *refuses* the job, and the calling Nix process executes the job normally, which is to say that it runs the builder if thisSystem matches d.system, and aborts otherwise.

- If an appropriate machine is known, the hook can either *accept* the job or tell the calling Nix process to *postpone* the job. The latter is appropriate if the load on the selected machine is too high: it is typical to allow no more jobs to run concurrently on a machine than the number of CPUs. If the hook returns a “postpone” reply, the calling Nix process will wait for a while, or continue with some other derivation (this is not supported by the algorithm in Figure 5.11, but is supported by the parallel algorithm in Section 5.5.4).

- If the hook has accepted the job, the calling Nix process will create a temporary directory containing all information that the hook needs to perform the build, such as the store paths of the inputs and output.

- The hook copies the FSOs denoted by the input paths (the *inputs* set computed in Figure 5.11) to the Nix store on the target machine, and registers them as valid with the appropriate reference graph mappings. It is important that the Nix stores on both machines are in the same location in the file system. The derivation itself (at p_{drv}) is also copied.

- The hook builds the derivation on the remote machine, e.g., by using the Secure Shell (ssh) protocol [61] to run nix-store --realise remotely.

- If the remote build succeeds, the output at d.output in the Nix store of the remote machine is copied back to the local Nix store.

When the hook finishes successfully, Nix proceeds normally as if it had executed the builder itself, i.e., by scanning for references in the output and registering path validity.

As stated above, this is a policy-free mechanism: it doesn’t care how the build hook does its work. In general, the build hook will have some configuration file listing remote machines, authentication information to automatically log in to those machine, and so on. An example of a concrete build hook is given in the context of build farms in Section 8.3.

Hook protocol A build hook is enabled by the user by pointing the environment variable NIX_BUILD_HOOK at the hook. When Nix needs to perform a build, it first executes the hook to discover whether it is able and willing to perform the build. Thus the build algorithm in Figure 5.11 is augmented as shown in Figure 5.14. Nix calls the hook $\boxed{84}$ with all information that the hook needs to determine whether it can handle the job, namely:

```
build($p_{drv}$) :
    (...elided...)
    if the environment variable NIX_BUILD_HOOK is defined :
        $p_{tmp} \leftarrow$ createTempDir()
        Write $inputs \cup$ closure($p_{drv}$) to file $p_{tmp}$/inputs
        Write $d$.output to file $p_{tmp}$/outputs
        Write the references of $inputs$ to file $p_{tmp}$/references
        $(fd_{read}, fd_{write}) \leftarrow$ pipe() $\boxed{83}$
        Asynchronously start NIX_BUILD_HOOK in directory $p_{tmp}$
            with $fd_{write}$ attached to file descriptor 3
            and arguments $[0, \text{thisSystem}, d.\text{system}, p_{drv}]$ $\boxed{84}$
        $reply \leftarrow$ read a line from $fd_{read}$
        if $reply$ = "accept" then :
            Wait until the hook finishes
            if the hook exited with exit code $\neq 0$ : abort
            Scan and register the path as valid, as in Figure 5.11
            return
        else if $reply$ = "postpone" then :
            Sleep a bit, then retry; see Section 5.5.4 for a better implementation.
        else if $reply \neq$ "decline" then : abort
        Fall through to normal handling...
    if $d$.system $\neq$ thisSystem : abort
    (...elided...)
```

Figure 5.14.: Augmentation of build in Figure 5.11 to add support for build hooks

- A Boolean value signifying whether the calling Nix process can execute more build jobs right now. This is applicable to the parallel build algorithm shown in Section 5.5.4 below, where Nix can execute a user-defined number of jobs in parallel.

- The value of thisSystem.

- The value of d.system. The hook needs this to determine whether it knows a remote machine of the appropriate type.

- The store path of the derivation (p_{drv}).

In addition, Nix creates a temporary directory containing the following files:

- inputs contains the store paths of all inputs, along with the closure of the store derivation itself.

- outputs contains the store path of the output. It is output*s*, plural, to support multiple output paths as described in Section 6.7.

- references contains a textual representation of the references mapping of the inputs. After copying the inputs to the target Nix store, the command nix-store --register-validity is used to register the validity of the copied paths on the target machine. This command accepts a description of the references graph in exactly this format.

A Unix pipe is created to communicate with the hook $\boxed{83}$. Its write side is attached to file descriptor 3 in the hook child process. The hook should print one the following strings on file descriptor 3 (followed by a linefeed character):

- postpone indicates that the caller should wait, typically until another job has terminated, hopefully decreasing the load on the intended target machine(s).
- decline means that the caller should try to build the derivation itself.
- accept means that the hook has accepted the build job.

If the hook accepts the job, it is responsible for producing output in d.output. If it finishes successfully, the output path is registered as valid, with the references determined by scanForReferences, just as in Figure 5.11.

5.5.3. Substitutes

Substitutes are the mechanism by which Nix enables its transparent source/binary deployment paradigm, as we saw in Section 2.6. A substitute is an invocation of some arbitrary program that creates a store object through some other means than normal building. It is primarily used to obtain the output of a derivation by downloading a pre-built FSO from a server rather than building the derivation from source.

The database mapping $\mathsf{substitutes} : \mathsf{Path} \rightarrow \{(\mathsf{Path}, [\mathsf{String}], \mathsf{Path})\}$ stores the substitutes that have been registered by user commands. The left-hand side is the store path to which the substitutes apply. The right-hand side is a set of substitutes. Each substitute is a three-tuple $(\mathit{substituter}, \mathit{args}, \mathit{deriver})$, where *substituter* is the path of the program to be executed to perform the substitution, *args* are the command-line arguments to be passed to that program, and *deriver* is the store path of the derivation that produced the FSO on the originating machine. The latter is necessary to maintain the deriver link for traceability (on installations where this is enabled). An example entry in the substitutes mapping is:

```
substitutes["/nix/store/bwacc7a5c5n3...-hello-2.1.1"] = {
    (   "download-url.sh"
    ,   ["http://example.org/1pc7y48cs3qn....nar.bz2"]
    ,   "/nix/store/1ja1w63wbk5q...-hello-2.1.1.drv"
    )
}
```

which establishes a single substitute: if we want to make path /nix/store/bwacc7a5c5n3...-hello-2.1.1 valid, we can do so by executing the program download-url.sh with argument http://example.org/1pc7y48cs3qn....nar.bz2. Here download-url.sh is presumably a program that downloads a file from the given URL, and uncompresses and unpacks it to store path /nix/store/bwacc7a5c5n3...-hello-2.1.1.

This is another example of a policy-free mechanism: Nix does not care how the substitute produces the store object. It can download from a server, interactively prompt the user to insert an installation CD, and so on. Indeed, there is no guarantee that a substitute produces *correct* output, which is a problem for secure deployment. The intensional model discussed in Chapter 6 addresses this issue. Section 7.3 discusses the implementation of a concrete binary deployment scheme based on the substitute mechanism.

The FSOs that we obtain through substitutes typically have references, of course. For instance, the Hello binary in /nix/store/bwacc7a5c5n3...-hello-2.1.1 has a reference to /nix/store/72by2iw5wd8i...-glibc-2.3.5. To maintain the closure invariant, we must ensure that the latter path is valid before we download the former. Thus, the references mapping for a path must be known beforehand. This means that when a substitute is registered, the references for the path in question must also be registered. Formally, this gives us the following invariant.

Invariant 6 (Reference invariant)**.** *A path that is valid or substitutable is called a* usable path*. For all usable paths, the set of references must be known:*

$$\forall p \in \mathsf{Path} : (\mathsf{valid}[p] \neq \varepsilon \vee \mathsf{substitutes}[p] \neq \varepsilon) \rightarrow \mathsf{references}[p] \neq \varepsilon$$

Note that the references entry for a valid or substitutable path can be the empty set ($\emptyset$), it just cannot be undefined (ε).

The command nix-store --register-substitutes registers a set of substitutes, along with the references of the paths to which the substitutes apply. This command is used by concrete deployment policies such as nix-pull (Section 2.6). For example:

```
$ nix-store --register-substitutes
/nix/store/bwacc7a5c5n3...-hello-2.1.1
/nix/store/1ja1w63wbk5q...-hello-2.1.1.drv
download-url.sh
1
http://example.org/1pc7y48cs3qn....nar.bz2
3
/nix/store/72by2iw5wd8i...-glibc-2.3.5
/nix/store/bwacc7a5c5n3...-hello-2.1.1
/nix/store/q2pw1vn87327...-gcc-3.4.4
```

Here a substitute is registered for the path /nix/store/bwacc7a5c5n3...-hello-2.1.1 that takes one argument and has three references. The substitute specification on standard input lists the store path being substituted, its deriver (if any), the substituter program, the command-line arguments, and the references. The latter two are listed one per line, each list preceded by a line specifying its length.

The function substitute(p) that tries to make a usable path p valid through substitutes is shown in Figure 5.15. It returns a Boolean indicating whether it succeeded in making p valid. If the path is already valid, we are done right away $\boxed{85}$. Likewise, if the path is not valid but there are no substitutes, we are also done, in a negative sense $\boxed{86}$.

Otherwise, the path is not valid but we do have substitutes, so we can hopefully build the path using one of them. However, to maintain the closure invariant, we must first ensure that all the paths that will be referenced by p are also valid. By the reference invariant, we already know the references of p. Thus, we call substitute recursively on all references $\boxed{87}$.

To prevent interference with potential other Nix processes while running the substitutes, a lock is first acquired on the output path $\boxed{88}$. As in the build algorithm in Figure 5.11, it

```
Bool substitute($p$) :
    if valid$[p] \neq \varepsilon$ : return true $\boxed{85}$
    if substitutes$[p] = \varepsilon$ : return false $\boxed{86}$
    for each $p' \in$ references$[p]$ :
        if $\neg$ substitute$(p')$ : return false $\boxed{87}$

    $lock \leftarrow$ acquirePathLock$(p)$ $\boxed{88}$
    if valid$[p] \neq \varepsilon$ : return true
    for each $(program, args, deriver) \in$ substitutes$[p]$ : $\boxed{89}$
        if pathExists$(p)$ : deletePath$(p)$
        if execution of program $program$ with
            arguments $[p] + args$ exits with exit code 0 : $\boxed{90}$
            assert(pathExists$(p)$)
            canonicalisePathContents$(d$.output$)$ $\boxed{91}$
            valid$[p] \leftarrow$ hash$_{\text{sha256}}$(serialise(readPath$(p)$)) $\boxed{92}$
            deriver$[p] \leftarrow deriver$
            releasePathLock$(p, lock)$
            return true
    releasePathLock$(p, lock)$
    if fall-back is disabled : abort $\boxed{93}$
    return false
```

Figure 5.15.: substitute: Substitution of store paths

is possible that the output path has become valid between the initial validity check and the acquisition of the lock. Thus, a final validity check is done.

Next, we try each substitute until one succeeds $\boxed{89}$. The substitute program is called with as command-line arguments the target path p, and the arguments registered for the substitute $\boxed{90}$. The substitute program is supposed to produce an FSO at path p and return exit code 0. If so, the path's metadata is canonicalised $\boxed{91}$, the path is registered as valid, and the deriver is set $\boxed{92}$. If not, the next substitute is tried until no more substitutes remain.

Generally, it is a fatal error if all substitutes for a path fail. When substitute is called from a derivation build action, Nix does not automatically "fall back" to building from source $\boxed{93}$. The reason is that we presumably registered the substitute for a reason, namely, to avoid the overhead of a source build. If the substitute fails for a transient reason—such as a network outage—the user in general does not want a source build instead, but prefers to wait or repair the cause of the problem. However, Nix commands do accept a flag to enable automatic fall-back:

```
$ nix-env -i hello --fall-back
...
downloading '/nix/store/bwacc7a5c5n3...-hello-2.1.1'
  from http://example.org/foo
curl: 'example.org': no route to host
falling back...
building '/nix/store/1ja1w63wbk5q...-hello-2.1.1.drv'
```

...

Relation to build hooks So what is the difference between a substitute and a build hook, as they both short-circuit the build process? A build hook is used to “out-source” the building of a derivation, while a substitute is used to reproduce an already existing build result. That is, build hooks operate at build time, while substitutes operate at deployment time. Specifically, the substitution of the output of a derivation only needs the references of the output (i.e., runtime dependencies) to be valid, while a build hook requires that all inputs of the derivation (i.e., build-time dependencies) are valid.

For example, the GNU Hello derivation has build-time dependencies on GCC and several other packages, but these are not referenced from the output. Thus, if we use a substitute to download a Hello binary, dependencies such as GCC will not be installed. But if we used build hooks, Nix *would* first install them. So while the substitutes can in principle be subsumed by build hooks, substitutes are much more efficient for deployment purposes (apart from the fact that making the build-time dependencies available may simply be unacceptable in many cases, such as closed-source deployment).

Also, substitutes allow us to forego store derivations altogether. For instance, nix-env can install output paths directly:

```
$ nix-env -i /nix/store/bwacc7a5c5n3...-hello-2.1.1
```

This command will make /nix/store/bwacc7a5c5n3...-hello-2.1.1 valid using a substitute (if it wasn’t already valid). The difference with installing from a store derivation is that this mode of installation cannot fall back to a build from source if the substitutes fail.

5.5.4. The parallel build algorithm

The build algorithm described above does not always make optimal use of the available hardware. When multiple CPUs are available, it is a good idea to execute multiple derivations in parallel. When substitutes are used, it may well be the case that each substitute only uses a fraction of the available download bandwidth, i.e., executing multiple downloads in parallel might speed things up. More importantly, when build hooks are used to distribute build jobs to remote machines, we certainly shouldn’t wait for one job to finish when there are idle machines.

This section briefly describes a *parallel* build algorithm that can execute multiple builds and substitutions simultaneously. The parallel algorithm, rather than the one described above, is what is implemented in the current Nix implementation.

The parallel build algorithm works on sets of *goals*. There are two types of goals:

- A *derivation goal* attempts to make the output of a derivation valid.
- A *substitution goal* attempts to make a path valid through substitutes.

Each type of goal is implemented by a finite state machine. The nodes of the finite state machine represent states. Each state has associated with it an *action* that is to be performed when the state is reached. Typically, the action is to start other goals or processes. The goal is then “put to sleep” until the condition associated with one of the out-going edges of the

```
run(topGoals) :
    awake ← topGoals
    while true :
        for each goal ∈ awake :
            Remove goal from awake
            Execute the action associated with the current state of goal
            if goal has reached a final state :
                Add goals waiting on goal to awake
                Remove goal from topGoals
        if topGoals = ∅ : return
        Wait for termination of child processes, add corresponding goals to awake
```

Figure 5.16.: run: Goal execution (sketch)

state becomes true. At this point, the goal is woken up, the transition to the corresponding new state is made, and the action associated with the new state is performed. This continues until the goal reaches a final state.

There is a top-level function run that executes a set of initial top-level goals (all being in their initial state). It is shown in Figure 5.16. It maintains a set of "awake" goals, e.g., those goals that have made a transition (meaning that some event has occured that is relevant to the goal). Initially, the top-level goals are all awake, so their initial actions can be executed. If a top-level goal finishes, it is removed from the set of top-level goals, and the runner continues until there are no more top-level goals.

The "events" that can wake up a goal are the termination of a child process (i.e., a builder, a substituter, or a build hook), or the termination of a *subgoal*. A subgoal of a goal g is a goal that g is waiting on. Subgoals can be shared. Suppose that we start the parallel building of top-level derivations Hello and Firefox, which both depend on the same Glibc. One top-level derivation will create a subgoal for Glibc and its other input derivations. It then goes to sleep, waiting for the subgoals to finish. When the Firefox goal is executed, it will see that a subgoal for Glibc already exists, and simply add itself to the list of "waiters". When the Glibc goal finishes, it will wake up both the Hello and Firefox goals, allowing them to make progress (as soon as their other subgoals have finished too, that is).

Figure 5.17 shows the state machine for derivation goals. When a goal starts, it creates a substitution subgoal for its derivation store path (which corresponds to the call substitute(p_{drv}) in the simple build algorithm in Figure 5.11). If the subgoal succeeds, it starts a substitution subgoal for the output path (corresponding to the call substitute(d.output)). If that subgoal does *not* succeed, we have to perform the build. So first the input derivations must be built, and thus derivation subgoals are created for each of those (corresponding to the recursive calls to build). If each is successful, we can run the build hook, or if that yields a decline response, the builder. We limit however the number of child processes that can run in parallel[9]. If this limit is reached, no more child processes can be started until one finishes (the available "build slots" are said to be exhausted), so the goal goes to sleep. When the hook or builder finishes successfully, the usual postprocess-

[9] This limit can be specified using the --max-jobs N flag.

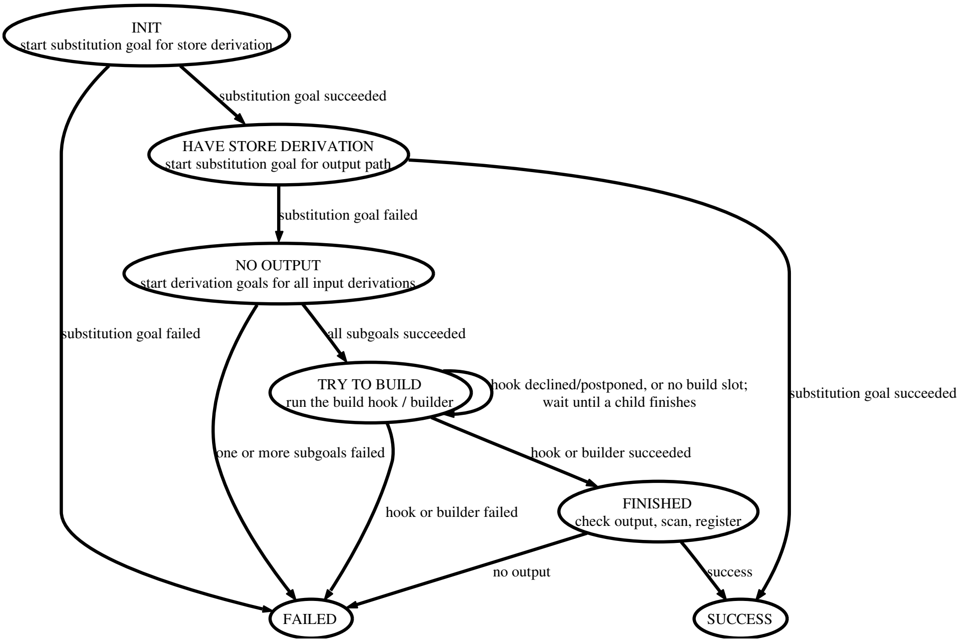

INIT
start substitution goal for store derivation
substitution goal succeeded
HAVE STORE DERIVATION
start substitution goal for output path
substitution goal failed
NO OUTPUT
start derivation goals for all input derivations
substitution goal failed
all subgoals succeeded
TRY TO BUILD
run the build hook / builder
hook declined/postponed, or no build slot;
wait until a child finishes
substitution goal succeeded
one or more subgoals failed
hook or builder succeeded
hook or builder failed
FINISHED
check output, scan, register
no output
success
FAILED
SUCCESS

Figure 5.17.: State machine for derivation goals

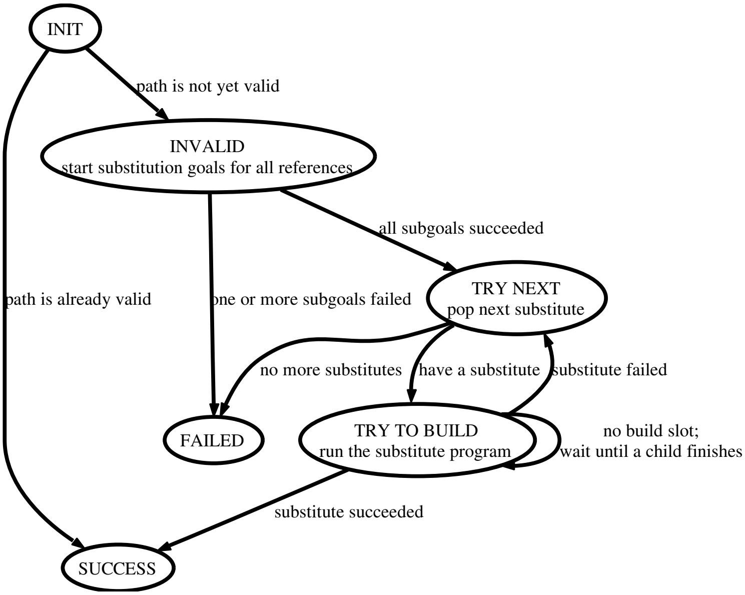


Figure 5.18.: State machine for substitution goals

ing is performed: the references are determined, the content hash of the path is computed, and the path is registered as valid. (The transitions from state FINISHED are not actual events, but simply the outcomes of the action associated with that state's action.)

Figure 5.18 shows the state machine for substitution goals. A substitution goal first checks that the path is not already valid (again, the edges from INIT are not actual events). If it is invalid, subgoals are started to make the references of the path valid. Then, each substitute is tried until one succeeds.

Normally, when a subgoal fails, its waitees (the goals waiting on it) are immediately woken up and fail as well (except when failure is normal, e.g., in the substitution of an output path). These waitees may have other subgoals that are still busy. Each goal has a reference count, and when the waitees kill themselves, they decrement the reference counts of all remaining subgoals. If the reference count of a goal reaches 0, it is also killed. Associated child processes are terminated. Thus, failure of a single subgoal propagates upwards, killing all other goals along the way.

This is usually the desired behaviour, but not always. For instance, in the Nix build farm described in Chapter 8, the job for Nixpkgs contains hundreds of more-or-less independent derivations. The failure of one should not terminate the others, since the build results of the successful derivations can be reused in subsequent build jobs. For this reason, there is a flag --keep-going that has the following effect: when a subgoal fails, the waitees are not woken up until all their other subgoals have finished as well. At that point, the waitees notice the subgoal failure, and make the transition to the FAILED state as well[10].

5.6. Garbage collection

Nix never deletes components from the Nix store on uninstallation or upgrade actions. Rather, to reclaim disk space, it is necessary to routinely run a *garbage collector*.

Garbage collection in Nix is pretty much like garbage collection in programming languages (except for one unique property discussed below). There is a pointer graph defined by the references relation, and there is a set of *roots*. The closure of the set of roots under the references relation is the set of *live store objects*—these are FSOs that are potentially in use, and therefore must not be deleted. All store objects that are not live are *dead*—these can be deleted safely.

The garbage collector therefore has the following phases:

- Determine the set of *root paths*.
- Compute the *live paths* as the closure of the set of root paths.
- Delete the *dead paths*, which are those that exist in the Nix store but are not in the set of live paths.

This section first discusses how root paths are registered. Then it shows two garbage collection algorithms: a plain "stop-the-world" collector that blocks all other Nix actions while it is running, and a concurrent collector that can run in parallel with other Nix actions.

[10] This behaviour is similar to GNU Make's [68] -k flag.

5.6.1. Garbage collection roots

The set of roots of the garbage collector are defined as symlinks to store paths in the directory /nix/var/nix/gcroots. Conceptually, if we for example have a store path /nix/store/-bwacc7a5c5n3...-hello-2.1.1 that we want to be preserved, we can register it as a root as follows:

```
$ nix-store -r $(nix-instantiate foo.nix)
/nix/store/bwacc7a5c5n3...-hello-2.1.1

$ ln -s /nix/store/bwacc7a5c5n3...-hello-2.1.1 \
    /nix/var/nix/gcroots/my-root
```

It is also possible to register store derivations as roots, e.g.,

```
$ nix-instantiate foo.nix
/nix/store/1ja1w63wbk5q...-hello-2.1.1.drv

$ ln -s /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv \
    /nix/var/nix/gcroots/my-root
```

This causes the store derivations and its dependencies—other store derivations and sources—to be preserved by the collector.

However, there is a race condition in this method of root registration. If the garbage collector is ran just between the call to nix-store or nix-instantiate on the one hand, and the creation of the symlink on the other hand, the store object and some or all of its dependencies may be garbage collected. It is therefore possible to let the Nix tool in question create the root:

```
$ nix-instantiate --add-root /nix/var/nix/gcroots/my-root foo.nix
/nix/var/nix/gcroots/my-root
```

Through locking, Nix ensures that there is no race condition.

Having to store links in /nix/var/nix/gcroots is often inconvenient. Consider the Nix utility nix-build that instantiates and builds a Nix expression, and leaves a symlink result pointing to the output path of the derivation in the current directory:

```
$ nix-build ./foo.nix

$ ls -l result
lrwxr-xr-x ... result -> /nix/store/bwacc7a5c5n3...-hello-2.1.1

$ ./result/bin/hello
Hello, world!
```

We want to implement nix-build so that its result is registered as a root. One way to do this is to have nix-build generate a fresh symlink in /nix/var/nix/gcroots, and have ./result be a symlink to *that*. I.e., we solve the problem through an extra indirection. However, there is a problem: if we delete ./result, the root persists. So the user would always need to make sure to delete the target of the result symlink when deleting result.

A better solution is the notion of *indirect roots*, which is in a sense the opposite solution. We let ./result be a direct symlink to the store path, and create a symlink to result in the designated directory /nix/var/nix/gcroots/auto. For example,

```
$ nix-store -r --add-root ./my-root --indirect \
    $(nix-instantiate foo.nix)
./my-root

$ ls -l my-root
lrwxr-xr-x ... result -> /nix/store/bwacc7a5c5n3...-hello-2.1.1

$ ls -l /nix/var/nix/gcroots/auto
lrwxr-xr-x ... 096hqdxcf6i2...  -> /home/eelco/test/my-root

$ ./my-root/bin/hello
Hello, world!
```

The name of the symlink in auto is a cryptographic hash of the root location (e.g., /home/eelco/test/my-root). This ensures that subsequent root registrations with the same location will overwrite each other (which is good, since it minimises the number of redundant symlinks in auto).

When the garbage collector runs, it chases the symlinks defined in /nix/var/nix/gcroots/auto. If it encounters a dangling symlink (i.e., pointing to a non-existent path), the root pointer is automatically deleted. Thus, when result is removed, the root in the auto directory will be removed as well on the next run of the garbage collector.

Figure 5.19 shows the algorithm for finding the garbage collector roots. It returns a set of store paths. The closure of this set under the references relation is the set of live paths.

By default there is a symlink in /nix/var/nix/gcroots to the directory /nix/var/nix/profiles/ that contains the user environment generation links. Thus all user environment generations are roots of the garbage collector.

Discussion Why don't we scan the entire file system outside of the Nix store for roots? This is certainly possible, but I have opted for a more disciplined root registration protocol instead for performance reasons—scanning the entire file system might take too long. Just scanning for symlink targets is not overly expensive, since it is easy to recognise symlinks that point to the Nix store. However, to be fully general we should also scan the contents of regular files, just as we do with the output of a build. Since we do not know anything about the internal structure of those files, we have to apply scanForReferences to the entire file system with *paths* set to all current store paths. Since there can be many store paths—millions in large installations—this becomes prohibitively expensive.

A possible extension to root discovery is to automatically use open store files as roots. In the scheme described above, it is possible that a currently executing program is garbage collected. For instance, the command sequence

```
$ nix-env -i firefox
$ firefox &  # start Firefox in the background
$ nix-env -e firefox
$ nix-env --remove-generations old
$ nix-store --gc
```

```
findRoots() :
    return findRoots′("/nix/var/nix/gcroots", false)

findRoots′($p$, $recurseSymlinks$) :
    if $p$ is a directory :
        $roots \leftarrow \emptyset$
        for each directory entry $n$ in $p$ :
            $roots \overset{\cup}{\leftarrow}$ findRoots′($p$ + "/" + $n$, $recurseSymlinks$)
        return $roots$
    else if $p$ is a symlink :
        $target \leftarrow$ targetOf($p$)
        if $target$ is a store path :
            return $\{p\}$
        else : // target is not a store path
            if $recurseSymlinks$ :
                if pathExists($target$) :
                    return findRoots′($target$, false)
                else :
                    deletePath($p$)
```

Figure 5.19.: findRoots: Root discovery

will cause Firefox to be (probably) garbage collected, even though it is still running.

There is no portable method to discover the set of open files, but on most operating systems there are methods to do so. For instance, on Linux the open files are the targets of the symlinks /proc/*/fd/*[11].

A more extreme approach is to scan for roots in memory. On Linux, the device /dev/mem provides access to the system's physical memory. This has the same efficiency problem as scanning in regular files: since we do not know anything about the internal structure of the memory contents, we have to scan for hash parts. Furthermore, since /dev/mem maps the physical address space, hashes may be fragmented in memory if they cross page boundaries; i.e., if character n of a hash is stored at address m, then character $n+1$ need not necessarily be at address $m+1$. The scanner may fail to detect references if this happens. So memory scanning only works well for memory devices that map virtual address spaces, e.g., /proc/*/mem.

5.6.2. Live paths

The live store paths are those that are reachable from the roots through the references relation. However, it is useful to extend the notion of reachability a bit. For instance, for some users it is desirable to keep the build-time dependencies of a component: if a user (e.g., a developer) often builds components from source, it is nice if a build-time

[11] On Linux, the /proc directory contains subdirectories for each live process containing information about them, e.g., /proc/7231, each having a subdirectory fd that holds symlinks to the open files of the process.

dependency like the compiler is retained by the garbage collector.

As another example, consider the traceability afforded by the deriver mapping, which tells us which derivation built a path. Some techniques, such as the blacklisting approach described in Section 11.1 to enforce upgrades of insecure or otherwise "bad" components, rely (in part) on knowledge of the build-time dependency graph.

For this reason, the Nix garbage collector has two options that can be set by the administrator of a Nix installation. They both extend the notion of liveness. The options are defined in the /nix/etc/nix/nix.conf.

- gc-keep-outputs: If true, the garbage collector will keep the outputs of non-garbage derivations. That is, if derivation p is live, then its output path parseDrv(readPath(p_{drv})).output is also live (if it is valid). If false (which is the default), derivation outputs are not followed, so they will be deleted unless they are roots themselves (or reachable from other roots).

  Note that this is stronger than registering the output of a derivation as a separate root, since it is transitive. That is, it keeps not just the output of the top-level derivation, but also the outputs of all its subderivations. For instance, if we registered the Hello derivation /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv as a root, then not only would the Hello output /nix/store/bwacc7a5c5n3...-hello-2.1.1 be kept (if valid), but also the source distribution downloaded by fetchurl, the C compiler, and so on.

- gc-keep-derivations: If true (the default), the garbage collector will keep the derivations from which non-garbage store paths were built. That is, if store path p is live, then its deriver deriver[p] is also live (if defined and valid).

Thus, the first option causes liveness to flow from derivations to outputs, while the second causes liveness to flow from outputs to derivations. Typically, developers turn both options on to ensure that all build-time dependencies are kept, even when only the outputs are registered as roots.

Figure 5.20 shows the computation of the set of live paths from the set of roots. It performs a simple traversal through the graph of store paths with edges defined by the union of the references relation, and the deriver and output links.

5.6.3. Stop-the-world garbage collection

After we have computed the live paths, we can garbage-collect by deleting all paths that are not live. However, life is not quite so simple:

- Some non-live paths are actually about to be "born", i.e., they are currently being built by some derivation.

- Paths that are garbage at the beginning of the collection might be "resurrected". This is a *big* difference with garbage collection in general [179]. In programming language implementations, garbage memory objects can never become live again because there are no pointers to them, and pointers are only created through allocation from the free space. Here, pointers (paths) are computed deterministically,

```
livePaths($roots$) :
    $todo \leftarrow roots$
    $done \leftarrow \emptyset$
    while $todo \neq \emptyset$
        $p \leftarrow$ remove one path from $todo$
        if $p \notin done$ :
            $done \overset{\cup}{\leftarrow} \{p\}$
            $todo \overset{\cup}{\leftarrow}$ closure($p$)
            if gc-keep-outputs = true $\wedge$ $p$ ends in ".drv" :
                $d \leftarrow$ parseDrv(readPath($p$))
                if valid[$d$.output] $\neq \varepsilon$ :
                    $todo \overset{\cup}{\leftarrow} \{d$.output$\}$
            if gc-keep-derivations = true $\wedge$ deriver[$p$] $\neq \varepsilon$ $\wedge$ valid[deriver[$p$]] $\neq \varepsilon$ :
                $todo \overset{\cup}{\leftarrow} \{$deriver[$p$]$\}$
    return $done$
```

Figure 5.20.: livePaths: Computing the set of live paths

so they can be recreated. For example, suppose that the Hello output /nix/store/-bwacc7a5c5n3...-hello-2.1.1 is dead at the start of the collection run. During the collection, a user may run nix-env -i hello and so make it reachable and live again.

The simplest solution by far is to use "stop-the-world" garbage collection, which means that there are no other Nix operations while the collector is running. A stop-the-world collector is shown in pseudo-code in Figure 5.21.

Stop-the-world semantics is implemented by using a global lock. All normal Nix processes acquire a read (shared) lock when they start, and release it when they terminate[12]. The garbage collector, on the other hand, acquires a write (exclusive) lock on startup $\boxed{94}$. Thus the collector can only proceed when there are no other processes modifying the store or the database. While the collector is running, new processes block until the collector finishes.

The collector then finds the roots and computes the set of live paths. Next, it finds the set of dead paths by reading the entries in the Nix store and tossing out those that are in the live set $\boxed{95}$. The dead paths should then be deleted. However, they cannot be deleted in arbitrary order, since that violates the closure invariant. This does not matter if the collector runs to completion, since at that time the closure invariant holds again. But if the collector is interrupted prematurely, we can have a store that violates the closure invariant. Consider a store with the Hello output and one of its dependencies, Glibc. Suppose that both are dead and about to be deleted. The collector deletes Glibc but is interrupted before it can delete Hello. Then Hello is still valid, and so if the user were to run nix-env -i hello, an incomplete Hello is installed into the user environment. (Note that the build algorithm

[12] This is not shown here, but such a lock is acquired in POSIX using the system call fcntl(fd, F_SETLKW, ...) and passing a lock type F_RDLCK. In contrast, the exclusive lock acquired by acquirePathLock in Figure 5.4 uses F_WRLCK.

```
gc() :
    $lock \leftarrow$ acquirePathLock("/nix/var/nix/global") $\boxed{94}$
    $roots \leftarrow$ findRoots()
    $live \leftarrow$ livePaths($roots$)
    $dead \leftarrow \emptyset$
    for each directory entry $n$ in the Nix store ($storeDir$) :
        $p \leftarrow storeDir + "/" + n$
        if $p \notin live$ : $\boxed{95}$
            $dead \overset{\cup}{\leftarrow} \{p\}$
    $dead' \leftarrow$ topological sort of $dead$ under the references relation
      such that $p < q$ if $q \in$ references$[p]$ $\boxed{96}$
    for each $p \in dead'$ (in topologically sorted order) :
        invalidatePath($p$)
        deletePath($p$)
    releasePathLock("/nix/var/nix/global", $lock$)

invalidatePath($p$) :
    In a database transaction: $\boxed{97}$
        valid$[p] \leftarrow \varepsilon$
        for each $p' \in$ references$[p]$ :
            referers$[p'] \leftarrow$ referers$[p'] - \{p\}$
        references$[p] \leftarrow \varepsilon$
        deriver$[p] \leftarrow \varepsilon$
```

Figure 5.21.: Stop-the-world garbage collector

through substitute in Figure 5.15 only checks the validity of the top-level path, and not its references, since the closure invariant makes such a check redundant. So it is actually difficult to recover from an inconsistent store; we cannot just rebuild all derivations.)

To prevent this, we need to make sure that a path is deleted before its references are deleted. That is, if p refers to q, then p must be deleted before q. This defines a partial ordering on the set of dead paths. A correct ordering on the set of dead paths that preserves the closure invariant can therefore be found by topologically sorting the dead paths under the relation $<$, where $p < q$ if $q \in$ references$[p]$ $\boxed{96}$.

Before each store object is deleted, its metadata in the database should be cleared, which is done by the function invalidatePath $\boxed{97}$. The cleanup must be done in a transaction to maintain database integrity. It must also be done before path deletion to maintain the validity invariant (Invariant 1).

It is easy to see that the collector in Figure 5.21 maintains the closure invariant.

Lemma 3 (No cycles in references graph)**.** *The references graph contains no non-trivial cycles. A cycle is trivial if it is a self-reference, i.e., $p \in$ references$[p]$.*

Proof. This property follows trivially from the fact that a path's references must exist prior to the creation of the path (since the references, determined by scanForReferences, are a subset of the *inputs* in Figure 5.11). A cycle means that there must exist a path p_1 that references a path p_2 that did not exist at the time p_1 was created. □

Theorem 4 (GC correctness)**.** *The garbage collector in Figure 5.21 maintains the closure invariant at all times.*

Proof. Assume that the invariant holds at the start of the garbage collector run. By contradiction, assume that the invariant is then violated at some point. This means that there exists a valid path p and an invalid path q, such that $q \in \mathsf{references}[p]$. However, then $p < q$ under the ordering defined above, and since by Lemma 3 there are no cycles in the references graph, $p < q$ implies $\neg(q < p)$. This means that p must have been deleted and made invalid before q. Thus p is both valid and invalid, a contradiction. □

5.6.4. Concurrent garbage collection

Stop-the-world garbage collection, while simple to implement, has very bad *latency*. While the collector runs, no other Nix processes can start (that is, they block until the collector finishes). This is unacceptable, since the collector might run for a very long time. For instance, in our Nix-based build farm (Chapter 8), where we run the collector once every few months, it can take a few hours to delete some 100 gigabytes of garbage.

Worse, there is a *liveness* issue: the collector might never start. If there always is a process holding a read lock, then the collector cannot acquire a write lock. Of course, any such process should terminate eventually, but by that time another process may have acquired a read lock as well. In a build farm or a Nix store with a similarly high utilisation, there might never be a point in time in which there is no process holding a read lock. If locks are strongly fair [3], this is not a problem: the kernel will just block further attempts to acquire a read lock while the garbage collector is waiting for a write lock. However, this is not the case in general[13].

Therefore we need a *concurrent* garbage collector—one that can run even as other Nix operations are executing. This section shows first a *partially* concurrent collector that addresses the liveness issue, i.e., that can run in parallel with already existing Nix processes. In particular, builds that are in progress (and that might take a long time to finish) do not prevent the collector from starting. We can then improve the latency problem by making this collector run incrementally.

The main issues to consider are the following:

- Some paths may be in use but may not *yet* be reachable from a root, e.g., during a build done by nix-env.
- New roots may be created while the collector is running. Paths reachable from those roots should not be deleted.

The solution to the first problem is to have running processes write a log of all store paths that they are about to access or create to a log file, which is read by the collector. Such a log file is per-process, e.g., /nix/var/nix/temproots/15726, where 15726 is the Unix process ID. The store paths in those files are called *temporary roots*. Function addTempRoot, shown in

[13] For instance, strong fairness is not specified for POSIX's fcntl operations.

```
addTempRoot($p$) :
    if this is the first call to addTempRoot in this process :
        do :
            Acquire a read lock on "/nix/var/nix/gc" $\boxed{98}$
            $fd \leftarrow$ open("/nix/var/nix/temproots/" $+ pid$,
              O_WRONLY|O_CREAT|O_TRUNC, 0666) $\boxed{99}$
            Release the read lock on "/nix/var/nix/gc"
            Acquire a read lock on $fd$
        while file $fd$ is not empty $\boxed{100}$
    Upgrade the read lock on $fd$ to a write lock $\boxed{101}$
    Write $p$ to $fd$
    Downgrade the write lock on $fd$ to a read lock
```

Figure 5.22.: addTempRoot: Registering a temporary garbage collection root

Figure 5.22, shows how a process registers a store path p as a temporary root. The idea is that to register a root, the process must acquire a write lock on its per-process log file $\boxed{101}$. The garbage collector, shown below, will acquire a read lock on each per-process log file. Thus, no process can proceed after the garbage collector has acquired a read lock on its log file.

There is a mechanism in place to clean up stale log files in /nix/var/nix/temproots. When the log file is first created $\boxed{99}$, a read lock is acquired to enable the collector to detect whether the log file is stale. If the collector can acquire a write lock on a log file, then apparently the corresponding process has died without removing its own log file, and thus the collector can remove the log file. However, to prevent a race with file creation, i.e., the collector removing a newly created log file before the process can acquire a lock on it, we use deleteSignal (page 98) to signal and detect this condition $\boxed{100}$.

There are several instances where addTempRoot must be called:

- In addToStore (Figure 5.3), after the call to makePath and before the first validity check. This ensures that the results of Nix expression translation are kept until process termination.

- In substitute (Figure 5.15), before the validity check. This ensures that the results of builds are kept until process termination.

On the other hand, we don't register the derivation path p_{drv} in build (Figure 5.11) as a temporary root, even though p_{drv} is going to be accessed. The reasoning is that registering p_{drv} is someone else's responsibility. For instance, in an execution of nix-env, any derivation path will have been registered in the same process by addToStore in the Nix expression translation. Likewise, in a call to nix-store --realise, the user is responsible for registering the argument as a root (e.g., by using nix-instantiate --add-root).

The second problem—new roots being created while the collector is running—is addressed by blocking the creation of new roots while the collector is running. To register a permanent root, a process must first acquire a read lock on the global lock file "/nix/var/nix/gc". The collector on the other hand acquires a write lock and holds it during

```
gc() :
    Acquire a write lock on "/nix/var/nix/gc" $\boxed{102}$
    $roots \leftarrow$ findRoots() $\boxed{103}$
    $live \leftarrow$ livePaths($roots$)
    $temp \leftarrow \emptyset$
    for each log file $l$ in "/nix/var/nix/temproots/" : $\boxed{104}$
        $fd \leftarrow$ open log file $l$
        if we can acquire a write lock on $fd$ :
            deleteAndSignal($l, fd$)
        else :
            Acquire a read lock on $fd$
            $temp \overset{\cup}{\leftarrow}$ the set of paths read from $fd$
    $live \overset{\cup}{\leftarrow} \bigcup_{p \in temp}$ closure($p$)
    $dead' \leftarrow$ topological sort of $dead$ under the references relation
    for each $p \in dead'$ (in topologically sorted order) :
        if $p$ ends in ".lock" : $\boxed{105}$
            $fd_{lock} \leftarrow$ open lock $p$
            if we cannot acquire a write lock on $fd_{lock}$ : skip
        invalidatePath($p$)
        deletePath($p$)
        if $p$ ends in ".lock" : deleteAndSignal($p, fd_{lock}$)
    Release the write lock on "/nix/var/nix/gc"
```

Figure 5.23.: Concurrent garbage collector

its execution. Since adding a lock takes a small, bounded amount of time, the collector will quickly acquire the write lock[14].

Figure 5.23 shows the concurrent garbage collector. It first acquires the global lock "/nix/var/nix/gc" $\boxed{102}$. This prevents new permanent roots from being created. Thus, the set of permanent roots determined in $\boxed{103}$ cannot increase while the collector runs. It also blocks new Nix processes since these acquire a read lock $\boxed{98}$.

It then reads the set of temporary roots from the per-process files in /nix/var/nix/-temproots/ $\boxed{104}$. It acquires a read lock on each file, thus blocking the creation of new temporary roots by the corresponding processes. Stale log files are removed here, as described above.

After this, no new roots—permanent or temporary—can be created. The garbage collector can therefore proceed normally. There are some boring details $\boxed{105}$ regarding the deletion of lock files, which we only want to delete if they are stale (left over from an interrupted operation). Staleness is discovered by attempting to obtain a write lock (without blocking). If this succeeds, then the lock is either stale, or a process just created it but has not acquired a lock on it yet. The latter case is handled by writing a deletion token to it

[14] Strictly speaking, this is not strongly fair: it is in principle possible that there is always at least one process holding a read lock. However, given the typical frequency of root creation and the time it takes to do so, this is extremely unlikely to occur in practice.

using deleteAndSignal. This token is detected by addTempRoot, which will restart upon seeing it.

The correctness of the algorithm follows from the observation that no Nix process will access any store path that is not in the *live* set, since all temporary roots are known to the collector through the log files, and the creation of new temporary or permanent roots is blocked.

The collector described here is still not perfectly concurrent: though it can run in parallel with previously executing processes (in particular, with builders), it blocks new processes and new temporary roots. Thus collector execution is strongly fair but still has bad latency. However, this is easily fixed by making the collector *incremental*. The collector can simply periodically restart until no more garbage exists (indeed, the *user* is free to interrupt the collector if desirable). Restarting will release the locks, allowing waiting processes to proceed. An even more concurrent scheme is one where processes inform the collector of new temporary roots by writing them to a socket and waiting synchronously for an acknowledgement from the collector.

5.7. Extensionality

So why do we call this model *extensional*? The reason is that we make an assumption of *extensional equality*. Build operations in general are not *pure*: the contents that a builder stores in the output path can depend on impure factors such as the system time. For instance, linkers often store a timestamp inside the binary contents of libraries. Thus, each build of a derivation can produce a subtly different output.

However, the *model* is that such impure influences, if they exist, do not matter in any significant way. For instance, timestamps stored in files generally do not affect their operation. Hence extensional equality: two mathematical objects (such as software components) are considered extensionally equal if they behave the same way for any operation on them. That is, we do not care about their internal structure.

Thus, while any derivation can have a potentially infinite set of output path contents that can be produced by an execution of its builder, we view the elements of that set as interchangeable.

But this is a model—an assumption about builders. If a builder yields a completely different component when invoked at a certain time of day, it is beyond the scope of the model. In reality, also, we can *always* observe inequality by observing the internal encoding of the component, since that is a permissible operation. But the model is that any such observation does not take place or does not have an observable effect.

6. The Intensional Model

The previous chapter discussed the *extensional* Nix store model. This model works very well, but it has one substantial limitation (as well as a few smaller ones): it does not allow *sharing* of a Nix store between arbitrary users. If we allow just anyone with an account on the system to run arbitrary Nix operations, we run into serious security problems:

- A user could modify the Nix store directly and inject a *Trojan horse* (a malicious component masquerading as legitimate software) into a component used by other users, allowing their accounts to be compromised when they execute the component.
- Even if direct access to the store were restricted, i.e., if all builds were performed by a system process on behalf of other users, it is still possible for users to "crack" the Nix store by starting a build that interferes with the builds of other users.
- And even if *that* were not possible, the substitute mechanism is dangerous in that we must *trust* that an FSO obtained through a substitute is actually a real build result of the derivation from which it is claimed to have been built.

This chapter shows a different model for the Nix store, called the *intensional model*, that allows these problems to be solved. The basic idea is to make the entire Nix store *content-addressable*, meaning that each store path has a name that corresponds to the hash of the contents of that path. This invariant implies that if a store path is trusted, then its contents are trusted as well, an improvement over the extension model in the previous chapter. Of course, whether to trust a store path is a different problem, but that assessment can be made by each individual user instead of being made globally for all users. If different users obtain substitutes from different sources, with different levels of trustworthiness, they do not interfere with each other.

The intensional model is a more recent development than the extensional model, and it has only been implemented as a rough prototype. Thus some of the results in this chapter are somewhat more tentative in nature. However, the crucial assumption of the intensional model—that the technique of *hash rewriting* works on common components—has been validated on the Nix Packages collection.

6.1. Sharing

This section motivates why sharing of a Nix store between multiple users is a desirable property. Sharing here means that multiple users have "write access" to the Nix store, i.e., can build and install software, create user environments, and so on. This immediately raises issues of trust: if not all users mutually trust each other, how can they share a store?

It should be noted that many deployment systems do not support sharing in this sense. For instance, almost all Unix deployment systems allow only the administrator (`root`) to install software. (An exception is Zero Install [112], which allows arbitrary users to install software, with duplicate installations of the same component shared between users, i.e., downloaded and stored on disk only once.) So lack of sharing is not necessarily a terrible limitation for a deployment system, as in many environments it is perfectly fine if only the administrator can make installation decisions. Such environments include single-user workstations (where the user is the administrator), small multi-user systems where all users trust each other, and typical multi-user environments where it is in fact a *feature* if the software environment can be determined exclusively by the system administrators.

Why do we want to share Nix stores? But many environments are somewhere in between the extremes of a single user on the one hand, and multiple users with a single deployment authority on the other hand. Imagine a university shell server that allows any student with an account to log in. It is clearly terribly insecure to give "root" access to any student in order to install software. But on the other hand, it is also desirable to allow students to install software. Other examples are computational grids and hosting environments. If the facilities of the deployment system are only available to the administrator, the users have to "manually" install software, e.g., by compiling from source, or by downloading and unpacking pre-built binaries. But in either case, the advantages of using a deployment system—such as dependency tracking—are gone and unavailable to the end-users. This can lead to correctness problems. For instance, if a user compiles a component that has a runtime dependency on a "globally" installed component, that is, one managed by the deployment system, then the deployment system will have no knowledge of this dependency. Thus, the deployment system may consider it safe to uninstall the dependency.

Also, if many users need a piece of software that has not been globally provided by the administrator, and each of them installs it separately (e.g., in his or her account), this will cause a considerable amount of resource wastage in terms of disk space, memory, CPU time lost due to redundant dynamic link resolution, and so on.

Another scenario where sharing is extremely valuable is in build management, i.e., in the domain of tools such as Make and Ant. A typical development model in a multi-developer project is that developers each have a working copy of the source code, obtained from a version management repository. Each developer then builds the project in his or her own working copy. However, with large projects, the overhead of building can be quite substantial, sometimes in the order of hours or days. Thus it would be great if the artifacts created by one user could be automatically "recycled" by other users; i.e., if a user has to build something that has already been built previously by another user, the previous build result is re-used automatically. Some build systems have this feature, notably Vesta's evaluator and ClearCase's ClearMake (discussed in Section 10.3). However, these tools depend on non-portable techniques to implement dependency identification, as discussed in Chapter 10.

Also, as we will see in Chapter 9, Nix is very useful for *service deployment*, e.g., a concrete web server with all its software components, static configuration and static data files, management scripts, and so on. It is easy to imagine a shared system, such as a hosting environment, where many different users instantiate Nix services. Many of those services will overlap, e.g., because they partially use the same software components (say, Apache). This common subset should be stored only once.

Finally, sharing is especially important in Nix because derivation outputs are typically not relocatable, since most outputs contain references to their own output path or to other paths in the Nix store. Any user can have a private Nix store, e.g., /home/alice/nix/store. However, pre-built binaries assume a particular location for the Nix store; almost always, this is /nix/store. Since the location of the Nix store is often stored in outputs, pre-built binaries are simply not applicable to a Nix store in another location in the file system. In fact, this is precisely why the location of the Nix store is part of the store path computation (page 94): different hashes are computed for the same store derivations in different Nix stores to prevent any possible mix-up through the substitute mechanism.

So for these reasons it is important for a Nix store—the "heap" or address space of installed components in the system—to be shareable among users. This means that users have not only "read" access to the components in the store, but can also run package management operations themselves, i.e., build, install, uninstall and upgrade components, both from source and from binaries through the substitute mechanism.

The trust model Of course, sharing is not hard—secure sharing is. The difficulty lies in providing certain trust properties. Given a derivation d, how can a user trust that the contents of d.output are actually the build result of this derivation, and not, say, a Trojan horse placed there by another user?

The basic trust problem is as follows. A user Alice[1] has obtained a set of Nix expressions from a *trusted* source, Dave (the **D**istributor). We do not define how this trust relation is established. The idea is that Alice trusts the producer of the Nix expressions to create expressions that build software that is useful, does not contain Trojans or *spyware*, does not destroy data, and so on. This type of trust cannot be established formally; the determination can only be made by a combination of knowledge of the producer's history, reputation, and other social factors, and possibly code review (although this is beyond most users). For instance, a Nix user will typically trust the Nix Packages collection.

We also assume that the set of expressions has been received unmodified by Alice, meaning that they have not been modified by a "man in the middle". This can be ensured by using appropriate cryptographic techniques, e.g., verifying a digital signature on the set of expressions [145], or fetching them from Dave through a secure channel (e.g., HTTPS with a valid certificate).

Given this trusted set of Nix expressions, the trust property that we seek to ensure is that if Alice installs a derivation from this Nix expression, the resulting component is one that has been produced by that derivation. I.e., if foo.nix is trusted, and Alice runs

```
$ nix-env -f foo.nix -i bar
```

to install a derivation named bar, then the derivation output that is added to Alice's user environment is an actual derivation output of that derivation. It should not be possible for other users to replace the derivation output with something else, or to interfere with the build of the derivation. To put it precisely, the property that we want is:

> The *trust property*: in a shared Nix store, if a user A installs or has installed a trusted Nix expression, then the closures resulting from building the store

[1] We follow the convention of the security and cryptographic literature to denote the several roles in a protocol or exchange as Alice (**A**), Bob (**B**), Carol (**C**), Dave (**D**), and so on.

> derivations instantiated from that expression are equal to what may have been produced if user *A* had exclusive write permission to the Nix store.

Thus, for each user, a shared Nix store must produce the same results as if the store had been a single-user store owned by that user. Note a subtlety in the phrasing: I say that the result of building must match what *may have produced* in a single-user store, not what they *would have been*. This is due to indeterminism; a single derivation may produce different results (e.g., due to time stamps). In essence, each derivation has a possibly infinite set of possible build results, its *output equivalence class*. But we do not have a direct way to test set membership in this class, i.e., to verify that a particular output was produced by a derivation. Testing this set membership is undecidable in general.

Informally, the trust property can be stated as follows:

> If the source of a component is trusted, then the binary that is installed in the user's user environment is also trusted.

Breaking the trust property So under what circumstances can the trust property be violated? The principal scenarios were sketched at the beginning of this chapter. Clearly, if users can modify the contents of the Nix store directly, then there is no security whatsoever. So let's assume that all operations on the Nix store are executed by a Nix server process (*daemon*) that performs Nix operations, such as instantiation, building and adding garbage collector roots, on behalf of users; and that only this server process and its children have write access to the store and the Nix database.

Even if direct access to the store is prevented in this way, malicious users can still "hack" the store by creating and building a derivation whose builder modifies other paths than its output path, e.g., overwrites an already existing store object with a Trojan horse. This violates the trust property, as the store object no longer corresponds to an actual output of its deriver.

Then there is the substitute mechanism. Recall from Section 5.5.3 that a substitute is an arbitrary program that builds a store path, typically by downloading its contents from somewhere. How can we trust that a substitute for a derivation output path actually produces contents that correspond to an actual output of the derivation? So the substitute mechanism can be used to trivially violate the trust property.

Sharing in the extensional model In the extensional model of the previous chapter, sharing a Nix store is only possible if all users of the Nix store trust each other not to violate the trust property. For instance, suppose that user Alice has obtained a Nix expression E from a trusted source, and pulls substitutes from machine X, where X is a malicious machine that provides Trojaned binaries for the output paths of the derivation produced by E. For instance, it provides a manifest with the following substitute definition:

```
{ StorePath: /nix/store/bwacc7a5c5n3...-hello-2.1.1 106
  NarURL: http://haxxor.org/dist/f168bcvn27c9...-hello.nar.bz2
  Size: 35182 }
```

where the NAR archive f168bcvn27c9...-hello.nar.bz2 unpacks to a version of Hello 2.1.1 containing a Trojan horse that installs a back door on Alice's machine. (Manifests were briefly discussed in Section 2.6, and will be covered in detail in Section 7.3.) When Alice executes the Hello binary, her account may become compromised.

Suppose that subsequently Bob wants to install the same expression E. He first pulls from a different, trusted machine Y, obtaining a bona fide substitute:

```
{ StorePath: /nix/store/bwacc7a5c5n3...-hello-2.1.1 107
  NarURL: http://nix.org/dist/1hnv0817f168...-hello.nar.bz2
  Size: 34747 }
```

However, this correct substitute will not avail him. When he installs Hello from the trusted Nix expression (nix-env -i hello), Alice's Trojaned binary will be added to his user environment. After all, the Hello derivation's output path, /nix/store/bwacc7a5c5n3...-hello-2.1.1, is already valid, and by the substitute mechanism (Figure 5.15) that is sufficient to finish the build of the derivation.

The basic problem is that both substitutes want to occupy the same location in the file system (cf. $\boxed{106}$ and $\boxed{107}$); but this is of course not possible. Nix makes an assumption of extensional equality (Section 5.7): any substitute for the same path is behaviourally equivalent. But it has no way to verify that this assumption is valid.

Thus, source X may claim that its substitute is an output of some derivation d; but this may be a lie, leading to a violating of the trust property. We can therefore only register substitutes from source X if all users that share the store trust source X.

The remainder of this chapter first shows that the trust property actually *can* be provided in the extensional model *for locally built derivations*. Thus, users need not have mutual trust relations. However, secure sharing becomes impossible once substitutes are used. This leads to the development of the *intensional model* that enables sharing of a Nix store even in the presence of substitutes.

6.2. Local sharing

As we have seen above, sharing between machines is only possible in the extensional model if all users have the same remote trust relations. For locally-built derivations on the other hand (i.e., when not using substitutes), we *can* allow mutually untrusted users. The trick is in preventing a user from influencing the build for some derivation d in such a way that the result is no longer a legitimate output of d.

For instance, if Alice has direct write access to the Nix store, she can start a build of derivation d, then overwrite the output path with a Trojan horse. Similarly, even if builds are done through a server process that executes builds on behalf of users but running under a different user ID (uid), Alice can interfere with the build of d by starting a build of a specially crafted derivation d', the builder of which writes a Trojan horse to the output path of d.

To ensure that the trust property holds for locally-built derivations, we therefore need to ensure that the following specialised property holds.

> The execution of the builder of a derivation d (d.builder) modifies no path in the Nix store other than d.output.

We can ensure this property as follows. First, users no longer have direct write access to the Nix store. As suggested above, this is the absolute minimum measure necessary to obtain secure sharing. All builds are performed by a Nix server process on behalf of users. The server runs builders under user IDs that are distinct from those of ordinary user or system processes (e.g., nix-build-$\{1,2,\ldots\}$). Also, no two concurrent builds can have the same uid. This prevents one builder from interfering with the output of another, as illustrated above. Thus, the server maintains a "pool" of free and in-use uids that can be used for building. How a builder can create its intended output path is discussed on page 157.

When a build finishes, prior to marking the output path as valid, we do the following:

- Ensure that there are no processes left running under the uid selected for the builder. On modern Unix systems this can be done by performing a kill(-1, SIGKILL) operation while executing under that uid, which has the effect of sending the KILL signal to all processes executing under uid [99]. Older versions of the POSIX standard however did not mandate this behaviour [98]: on such systems, killing all processes running under a certain uid is tricky as it is fraught with race conditions.

- Change ownership of the output to the global Nix user.

- Remove write permission and any set-user-ID and set-group-ID bits (which are special permission bits on files that cause them to be executed with the rights of a different user or group—a possible security risk).

Note that the latter two steps have a subtle race condition. For instance, if we change ownership first, we have the risk of inadvertently creating a setuid binary owned by the global Nix user[2]. If however we remove write and setuid permission first, a left-over process spawned by the builder could restore those permissions *before* the ownership is changed. This is why the first step is important. Also, on Unix, if a left-over process opened a file before the ownership changes, it can still write to it after the change, since permissions are only checked when a file is opened.

In conclusion, in the extensional model we can do secure source-based deployment *with sharing*. Thus, the trust property is guaranteed in the absence of substitutes. Of course, this is not enough: we also want transparent binary deployment through the substitute mechanism.

6.3. A content-addressable store

As we saw in Section 6.2, we can have secure sharing of locally built derivation outputs, but not of remotely built outputs obtained through the substitute mechanism. All users have to trust that the contents produced on another machine purportedly from some derivation d is indeed from derivation d. As stated above, such a trust relation must be global for a Nix installation. In this section we develop a substantially more powerful model in which this is not the case. We do this by moving to a *fully content-addressable Nix store* for all store objects.

The property of content-addressability means that the address of an object is determined by its contents. Equivalently, if we know the contents of an object, we know its address. In the context of Nix, addresses are of course store paths. We have seen in Section 5.3

[2] Exactly to prevent this race, Linux removes setuid permission when changing ownership. However, this is not standard behaviour [99].

(page 97) that some FSOs in the extensional model have this property. In particular, store derivations and sources added to the store by instantiate have store paths that are computed from their contents. The function addToStore (Figure 5.3) computes the store path as follows:

$$p \leftarrow \mathsf{makePath}(\texttt{"source"}, \mathsf{printHash}_{16}(h), \mathit{name})$$

Thus the store path depends entirely on the cryptographic hash h of the contents of the FSO being added, and its symbolic name. (So strictly speaking, these paths can only be considered "content-addressing" if the symbolic name is considered part of the content.)

Since we assume that collisions of the cryptographic hash functions used in Nix do not occur in practice (since they are supposedly infeasible to produce), we have a one-to-one correspondence between paths and contents. This gives us some concrete properties. Given a particular FSO *fso*, there is at most one path whose contents are equal to *fso*. Also, assuming infeasibility of finding hash collisions and given two Nix stores both containing a valid path p, the FSOs stored at p must be equal (for if they are not, we have found a hash collision).

Content-addressability is an extremely useful property. Recall the example on page 138, where Alice and Bob had different trust relations, so there were substitutes with different contents for a single store path. This problem does not exist in a content-addressable store, since in such a system path equality (between different substitutes) implies content equality.

But not all store objects in the extensional model are stored in a content-addressed way. Derivation outputs are not; the output path of a derivation is computed from the input derivation (Figure 5.6), *not* from the actual contents produced by the builder. Below we will show how we can achieve content-addressability even for derivation outputs. As we shall see, this is not trivial due to self-referential components.

Once we have the property of content-addressability, different users can have different trust relations: *for each user* we can have a *different* derivation to output path mapping. This is the *intensional model*—equality is defined by internal contents, not by observable behaviour. This model is much stronger than the extensional model, since it doesn't make the simplifying but unenforcible assumption of builder purity. Rather, intensionality is an inherent property of the system.

6.3.1. The hash invariant

The crucial property of the intensional model is that the Nix store is now content-addressable: if we know the contents of a store object, we know its store path.

Ideally, this would mean that if an FSO has hash h, then it is stored in the Nix store at $\mathit{nixStore}/h-\mathit{name}$, where *nixStore* is the location of the Nix store and *name* is the symbolic name for the FSO (e.g. hello-2.1.1). For instance, if a Hello FSO has truncated SHA-256 hash lm61ss8nz7hl..., then its store path would be /nix/store/lm61ss8nz7hl...-hello-2.1.1.

However, this is not sufficient, since the symbolic name must be taken into account in the hash part of the store path as well. If the same FSO were added multiple times, but with different symbolic names, the resulting store paths must not have the same hash part, since our scanning approach to finding references uses only the hash part to distinguish dependencies. Therefore, in the intensional model, the function makePath that computes store paths from a content hash h and a symbolic name *name* is defined as follows:

$$\begin{aligned}&\mathsf{makePath}(h,\ \mathit{name}) = \\ &\quad \mathit{nixStore} + \text{"/"} + \mathsf{printHash}_{32}(\mathsf{truncate}(20,\ \mathsf{hash}_{\mathsf{sha256}}(s))) + \text{"-"} + \mathit{name}\end{aligned}$$

where

$$s = \text{"sha256:"} + \mathsf{printHash}_{16}(h) + \text{":"} + \mathit{name}$$

Note that contrary to the definition of makePath in the extensional model (page 94), this function produces a store path that depends only on a cryptographic hash h of the contents of the FSO to be stored, and its symbolic name *name*.

The hash h is the hash of the contents of the FSO being added. It is almost, *but not quite*, the SHA-256 hash over the serialisation of the FSO, i.e., $\mathsf{hash}_{\mathsf{sha256}}(\mathsf{serialise}(\mathit{fso}))$. As we shall see below, hashing is a bit more complicated due to self-referential components.

But let's ignore that for now. Suppose that we want to add a store object to the store that has SHA-256 hash 73b7e0fc5265... and symbolic name hello-2.1.1. Then,

$$\begin{aligned}s &= \text{"sha256:"} + \text{"73b7e0fc5265..."} + \text{":"} + \text{"hello-2.1.1"} \\ &= \text{"sha256:73b7e0fc5265...:hello-2.1.1"}\end{aligned}$$

and since

$$\begin{aligned}&\mathsf{printHash}_{32}(\mathsf{truncate}(20, \mathsf{hash}_{\mathsf{sha256}}(s))) = \\ &\quad \text{"wf9la39rq7sx5hj0jry0mn5v48w8cmwi"},\end{aligned}$$

we obtain the store path

/nix/store/wf9la39rq7sx5hj0jry0mn5v48w8cmwi-hello-2.1.1

The hash part wf9la39rq7sx... is determined entirely by the contents and the symbolic name. There can never be another object in the store with the same contents and symbolic name, but with a different store path. So there is a correspondence between a store path's hash part, and the contents at that store path. This is called the *hash invariant*, and it is what sets the intensional model apart from the extensional model.

Invariant 7 (Hash invariant)**.** *The hash part of a valid store path p is determined by the cryptographic hash of p:*

$$\forall p \in \mathsf{Path} : \mathsf{valid}[p] \neq \varepsilon \rightarrow p = \mathsf{makePath}(\mathsf{valid}[p], \mathsf{namePart}(p))$$

Recall that $\mathsf{namePart}(p)$ returns the symbolic name component of path p, e.g. for /nix/store/bwacc7a5c5n3...-hello-2.1.1 it returns hello-2.1.1. Also recall that $\mathsf{valid}[p]$ contains the cryptographic hash of the serialisation of the FSO at p, stored when p was made valid. Due to the stored hash invariant (page 96), this hash must match the actual contents at p. Also note that while we compare the entirety of p to $\mathsf{makePath}(\mathsf{valid}[p], \mathsf{namePart}(p))$, it is only the hash part of p that actually matters. The equality of the name part is tautological, since we feed namePart(p) into the call to makePath.

So how does content-addressability help us to achieve secure sharing in multi-user Nix stores? The answer is that users can now independently install software, i.e., build derivations. If the results of those independent builds are the same, they get sharing; if results differ due to impurity, they do not. This applies not just to local builds but more significantly to substitutes.

In the example on page 138, when Alice installs a derivation for Hello using a Trojaned substitute from a malicious machine, the result will end up in some path, say /nix/store/-5n5drxv693s3...-hello-2.1.1. If Bob installs the same derivation but using a legitimate substitute, the contents will differ and thus the result will necessarily be in a different store path, e.g. /nix/store/4d6hb6vxh388...-hello-2.1.1. This path will be added to his user environment. Thus, he is insulated from Alice's bad remote trust relation.

6.3.2. Hash rewriting

The property of content-addressability is easily stated but not so easily achieved. This is because we do not know the content hash of a component until after we have built it, but we need to supply an output path to the builder beforehand so that it knows where to store the component. In order to achieve content-addressability for derivation output, we must *rename* them after they have been built to the appropriate content-addressed name in the store. Roughly speaking, we first build the derivation in a temporary location, compute the hash over the result, and rename the temporary path to the content-addressed final location.

So the first step is to perform a build in a store path p with a *randomly generated hash part*, i.e.,

$$p = \text{makePath}(\text{randomHash}(), \textit{name})$$

The function randomHash() produces a sufficiently long pseudo-random base-32 number of the same length as a normal hash part, i.e., 32 characters. For instance, if we are building Hello, we might produce the temporary path

$$\begin{aligned} p &= \text{makePath}(\text{"2c8d367ae0c4..."}, \text{"hello-2.1.1"}) \\ &= \text{/nix/store/2jxsizriq3al...-hello-2.1.1} \end{aligned}$$

Thus, the builder is executed with the environment variable out set to the store path /nix/store/2jxsizriq3al...-hello-2.1.1.

If the build finishes successfully, we compute the SHA-256 hash h over the serialisation of the FSO at p. We then *rename* path p to p', computed as follows:

$$p' = \text{makePath}(h, \textit{name})$$

Thus, the temporary path p, which did not obey the hash invariant, is changed to one that does obey the hash invariant. Suppose that after the build of Hello we compute content hash 5c769ad74cac.... Then the final path p' becomes:

$$\begin{aligned} p &= \text{makePath}(\text{"5c769ad74cac..."}, \text{"hello-2.1.1"}) \\ &= \text{/nix/store/jj8d7j9j6scc...-hello-2.1.1} \end{aligned}$$

So the temporary build result /nix/store/2jxsizriq3al...-hello-2.1.1 is renamed to /nix/store/-jj8d7j9j6scc...-hello-2.1.1.

There is a snag, however: simply renaming the temporary path doesn't work in case of self-references, that is, if the binary image of a file refers to its own store path. This is quite common. For instance, the RPATH of a Unix executable (page 23) frequently points to its own directory so that related library components can be found. If we rename the temporary path p to p' in such a case, the references to p will become *dangling references*, and the component probably will not work anymore.

We solve this problem through the technique of *hash rewriting*. The basic idea is that we copy p to p', but at the same time replace all occurrences of the string $\mathsf{hashPart}(p)$ in the FSO with $\mathsf{hashPart}(p')$. (Recall that hashPart(p) yields the hash part of path p.)

However, this changes the contents of the component, thereby invalidating the hash! So we have a circular problem: we want to change the contents of an FSO to match its hash, but that changes its hash, so the contents must be changed again, and so on. With cryptographic hashes it is not feasible to compute a “fixed point”: a string that contains a representation of the hash to which the string hashes. I.e., it is not computationally feasible to compute a string s such that

$$s = s_1 + \mathsf{printHash}(\mathsf{hash}_t(\mathsf{s})) + s_2$$

where s_1 and s_2 are an arbitrary prefix and suffix to the occurrence of the hash representation.

We solve this problem by computing hashes *modulo self-references*. Essentially, this means that we ignore self-references when computing the hash. So the hash h is not computed as the SHA-256 hash of the serialisation of the FSO:

$$h = \mathsf{hash}_{\mathsf{sha256}}(\mathsf{serialise}(\mathsf{readPath}(p)))$$

Instead, the hash is computed over the serialisation of the FSO, with all occurrences of $\mathsf{hashPart}(p)$ *zeroed out*. That is,

$$h = \mathsf{hashModulo}(\mathsf{serialise}(\mathsf{readPath}(p)),\ \mathsf{hashPart}(p))$$

where

$$\mathsf{hashModulo}(c,\ hp) = \mathsf{hash}_{\mathsf{sha256}}(\sum_{i \in \mathsf{find}(c,hp)} (i + \texttt{":"}) + \texttt{":"} + \mathsf{replace}(c,\ \{hp \leadsto \mathbf{0}\}))$$

The function $\mathsf{find}(c,h)$ (first seen in Section 5.5) yields the offsets of the occurrences of the substring h in the string s. The constant $\mathbf{0}$ denotes a string consisting of binary 0s of the same length as the hash part hp, 32 characters. The function $\mathsf{replace}(c,r)$ applies a set of textual substitutions r to the string s. Textual substitutions in r are denoted as $x \leadsto y$, i.e., occurrences of the string x are replaced with the string y.

The function hashModulo computes a hash not just over c with the hash part hp cleared out, but also takes the offsets of hp into account. It is necessary to encode the offsets of the occurrences of hp into the hash to prevent hash collisions for strings that are equal except for having either hp or 0-strings at the same location. The colons simply act as disambiguators, separating the offsets and the contents.

For example, suppose we have two FSOs A and B that are equal except that at one point, A has a self-reference while B has zeroes, e.g.,

$$\begin{aligned}A &= \text{"xxxxHxxxHxxHxxx"}\\ B &= \text{"xxxxHxxx0xxHxxx"}\end{aligned}$$

where H is a self-reference and 0 is a string of 32 zeroes. If we do not take self-reference locations into account, A and B will hash to the same value.

The hash h computed using hashModulo is used to produce the new store path p' as described above, i.e., using makePath. Then, we copy p to p', rewriting all occurrences of $\mathsf{hashPart}(p)$ in the contents of p with $\mathsf{hashPart}(p')$:

$$\mathsf{writePath}(p', \mathsf{deserialise}(c'))$$

where

$$c' = \mathsf{replace}(\mathsf{serialise}(\mathsf{readPath}(p)), \mathsf{hashPart}(p) \rightsquigarrow \mathsf{hashPart}(p')).$$

Note that even though in case of self-references in general

$$\mathsf{hash}_{\mathsf{sha256}}(\mathsf{serialise}(\mathsf{readPath}(p))) \neq \mathsf{hash}_{\mathsf{sha256}}(\mathsf{serialise}(\mathsf{readPath}(p'))),$$

we do have

$$\begin{aligned}&\mathsf{hashModulo}(\mathsf{serialise}(\mathsf{readPath}(p)), \mathsf{hashPart}(p))\\ =\ &\mathsf{hashModulo}(\mathsf{serialise}(\mathsf{readPath}(p')), \mathsf{hashPart}(p')).\end{aligned}$$

That is, the hash modulo the randomly generated hash part does not change after hash rewriting.

6.4. Semantics

We can now formalise the intensional model. The main difference with the extensional model is that output paths are no longer known *a priori*. But because of this, we cannot prevent re-building a derivation by checking (as was done in substitute in Figure 5.15) whether its output path is already valid. The same applies to checking for substitutes, which are also keyed on output paths.

Also, due to impurity, a single derivation can now result in several distinct components residing at different store paths, if the derivation is built multiple times (e.g., by different users). That is, a derivation actually defines an *equivalence class* of store paths within the Nix store, the members of such classes all having been produced by the same derivation.

Therefore, we make the following change to the abstract syntax of store derivations defined in Figure 5.5 (page 101). The field output is removed; we don't know the output path in advance. But we add a field eqClass : EqClass that identifies the equivalence class of the output.

So what is an equivalence class (i.e., what is the type EqClass)? In fact, the equivalence class eqClass is *exactly the same* as the original output field! It is computed in the same way, namely, by hashing the derivation with its eqClass field set to the empty string (callout $\boxed{64}$ on page 102). Note that the hash parts produced there cannot conflict with those computed by makePath in the previous section, assuming that there are no hash collisions (since the former hash preimages start with sha256:output:out:, while the latter hash preimages start with sha256: followed by a hash). So for instance, when instantiating our running example, the Hello derivation, we obtain d.eqClass = "/nix/store/bwacc7a5c5n3...-hello-2.1.1". In the extensional model, we had d.output = "/nix/store/bwacc7a5c5n3...-hello-2.1.1". Since eqClass is just a (fake) store path, the type EqClass is just an alias for the type Path introduced for legibility.

So we have really just renamed the field output to eqClass. However, there is an important difference in the *meaning* of eqClass. The equivalence class path is “virtual”: it is never built. For example, in the intensional Nix store, we will never actually *build* a path /nix/store/bwacc7a5c5n3...-hello-2.1.1.

The reason for using store paths for equivalence classes is that it gives us an easy way to refer to the output of a derivation from other derivations. For instance, the envVars field of a derivation that depends on Firefox must in some way refer to the path of the Firefox component, even though this path is not known in advance anymore. When we build a derivation d depending on derivation d', we simply *rewrite* in d.envVars all occurrences of hashPart(d'.eqClass) to a trusted member of the equivalence class denoted by d'.eqClass.

Since we must remember for each derivation what output paths were produced by it and who built or substituted them, we define a database mapping eqClassMembers : EqClass → {(UserId, Path)}. The type UserId denotes a user identity. Here we will simply take them to denote Unix user names, e.g., alice. More elaborate notions of identity are also possible, e.g., using groups or classes of users.

Through the table eqClassMembers, an equivalence class maps to a set of concrete store paths along with the names of the users that built or substituted them. For instance, if both Alice and Bob have obtained Hello from their respective substitutes, the entry for /nix/store/bwacc7a5c5n3...-hello-2.1.1 might be as follows:

$$\begin{array}{l}\mathsf{eqClassMembers}[\texttt{/nix/store/bwacc7a5c5n3...-hello-2.1.1}] = \\ \quad \{\ (\texttt{"alice"},\ \texttt{/nix/store/bpv6czrwrmhr......-hello-2.1.1}) \\ \quad ,\ (\texttt{"bob"},\ \texttt{/nix/store/mqgi8xail9vp......-hello-2.1.1}) \\ \quad \} \end{array}$$

Of course, a store path can occur multiple times for different users. This happens if multiple builds have yielded exactly the same result (the ideal!), or if users have installed from the same substitute.

The members of an equivalence class (i.e., the right-hand side of an eqClassMembers entry) must be usable paths; that is, they must either be valid or have at least one substitute.

We can now define the set of *trusted paths* in the equivalence class of a derivation output as the set of valid or substitutable paths for some user:

$$\mathsf{trustedPaths}(eqClass,\ user) = \{p \mid (user, p) \in \mathsf{eqClassMembers}[eqClass]\}$$

Equivalence classes and closures A path can be a member of multiple equivalence classes. This is easy to see: we can conceive of any number of different derivations that produce the output "Hello World" in various ways. So we cannot unambiguously say to what equivalence class a path belongs. However, as we shall see below, there are times when we need to know this. In particular, when we compute the closure of a path p, we want to know to what equivalence class each reference belongs. Such a query is only meaningful given a certain *context*, i.e., when we compute the closure of a path p in an equivalence class e.

To this end, a new database mapping $\mathsf{refClasses} : (\mathsf{Path}, \mathsf{Path}) \to \{(\mathsf{EqClass}, \mathsf{EqClass})\}$ is needed. This table allows us to determine equivalence classes when we are following the references graph. It has the following meaning: if $(e, e') \in \mathsf{refClasses}[(p, p')]$, then there is an edge in the references graph from path p in equivalence class e to path p' in equivalence class e'.

The function $\mathsf{closure}'(p,\ e)$ computes the closure of a path p in equivalence class e. It returns a set of *pairs* $(p, e) : (\mathsf{Path}, \mathsf{EqClass})$. Its definition is as follows:

$$\mathsf{closure}'(p,\ e) = \{(p, e)\} \cup \bigcup_{p' \in \mathsf{references}[p]} \mathsf{followRef}(p, e, p')$$

The auxiliary function $\mathsf{followRef}(p, e, p')$ yields the closure of p' coming from path p in equivalence class e:

$$\mathsf{followRef}(p, e, p') = \begin{cases} \mathsf{closure}'(p',\ \varepsilon) & \text{if } es' = \emptyset \\ \bigcup_{e' \in es'} \mathsf{closure}'(p',\ e') & \text{otherwise} \end{cases}$$

where

$$es' = \{e' \mid (e, e') \in \mathsf{refClasses}[(p, p')]\}$$

The case where $es' = \emptyset$ is to handle paths that are not in any equivalence class. This is quite normal: paths that are not outputs of derivations (such as sources) need not be in equivalence classes. For such paths, the special value ε is used to denote the absence of an actual equivalence class. It should be noted that the left-hand sides of the pairs (the paths) in the set returned by $\mathsf{closure}'(p,\ e)$ are the same for all e given a certain p. Only the right-hand sides (the equivalence classes) can differ.

6.4.1. Equivalence class collisions

The fact that a derivation can resolve to any number of output paths due to impurity, leads to the problem that we might end up with a closure that contains more than one element from the output equivalence class of a derivation.

Figure 6.1 shows an example of this phenomenon. (As in Figure 2.3, an edge from node A to B denotes that A is a build-time input of B, i.e., the output of B can have a reference to the output of A.) Suppose that Alice has built gtk and pkgconfig locally. These both depend on glibc. She has also registered Bob's remotely built libIDL as a substitute. It also depends on glibc. However, though Bob's glibc was built from the same derivation, due to impurities the build result is different. Thus, the equivalence class for the Glibc derivation has at least two members. This is not a problem in itself. However, suppose that Alice next tries to build firefox, which depends on gtk, pkgconfig, and libIDL. We then end up with a Firefox

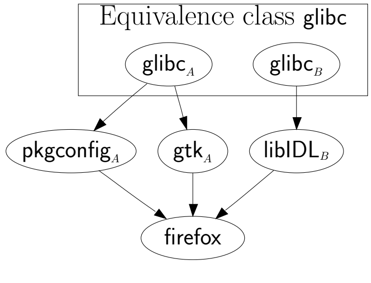


Figure 6.1.: Equivalence class collision

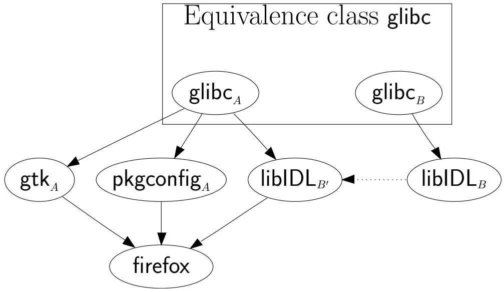


Figure 6.2.: Resolution of the equivalence class collision

binary that links against two glibcs. This might work, or it might not—depending on the exact semantics of dynamic linking. In any case, it is an observable effect—it influences whether a build succeeds and whether the build result works properly.

Thus, we need to prevent that any closure ever contains more than one path from an equivalence class. This is expressed by the following invariant.

Invariant 8 (Equivalence class unicity invariant)**.** *No two paths in a closure can be in the same equivalence class.*

$$\begin{array}{l}\forall p \in \mathsf{Path} : \mathsf{valid}[p] \neq \varepsilon \rightarrow \\ \quad \forall e \in \mathsf{EqClass} : \\ \quad\quad \forall (p_1, e_1) \in \mathsf{closure}'(p,\ e) : \forall (p_2, e_2) \in \mathsf{closure}'(p,\ e) : \\ \quad\quad\quad (e_1 \neq \varepsilon \wedge e_1 = e_2) \rightarrow p_1 = p_2\end{array}$$

That is, for any two elements (p_1, e_1) and (p_2, e_2) in any closure of a valid path p, if the equivalence classes are the same ($e_1 = e_2$), then the paths must also be the same ($p_1 = p_2$). The condition $e_1 = \varepsilon$ is to handle paths that are not in any equivalence class (such as sources).

So when we build a derivation, we have to select from the paths in the union of input closures *a single element from each equivalence class*. However, we must still maintain the closure invariant (page 96). For instance, in Figure 6.1, we cannot simply select the set $\{\mathsf{glibc}_A, \mathsf{gtk}_A, \mathsf{pkgconfig}_A, \mathsf{libIDL}_B\}$, since libIDL_B depends on glibc_B which is not in this set. (In this running example, for clarity I do not use store paths but symbolic constants, e.g., glibc_A for an output and glibc_{eq} for its equivalence class. The subscripts have no significance; they just denote different store paths.)

Once again, hash rewriting comes to the rescue. We can *rewrite* libIDL_B so that it refers to glibc_A instead of glibc_B. That is, we compute a new path libIDL'_B with contents

$$\mathsf{replace}(\mathsf{serialise}(\mathsf{readPath}(\mathsf{libIDL}_B)),\ \{\mathsf{hashPart}(\mathsf{glibc}_B) \rightsquigarrow \mathsf{hashPart}(\mathsf{glibc}_A)\})$$

(Of course, self-references in libIDL_B must also be rewritten as described in Section 6.3.2.) This is shown in Figure 6.2. The dotted edge denotes a copy-with-rewriting action.

Path selection An interesting problem is *which* paths to select from each equivalence class such that the number of rewrites is minimised. For instance, if we select glibc_A, then we have to rewrite one path (namely libIDL_B), while if we select glibc_B, we have to rewrite two paths (gtk_A and $\mathsf{pkgconfig}_A$). I do not currently know whether there exists an efficient algorithm to find an optimal solution. A heuristic that works fine in practice is to do a bottom-up traversal of the equivalence class dependency graph, picking from each class the path that induces the least number of rewrites.

However, picking an optimal solution with respect to the current derivation is not particularly useful in any case, since it ignores both the state of the Nix store as a whole, and future derivations. In our example Alice might in the future install many additional components from Bob's remote repository (e.g., because Bob is a primary distribution site). Thus, globally there are many more paths referring to glibc_B than to glibc_A. In this case it is better to select glibc_B and rewrite Alice's locally built components. Possible heuristics include:

- Select the path that has the largest *referrer closure* in the Nix store. Recall from Section 5.2.3 that the referrer relation is the transpose of the references relation. Thus, the referrer closure of p is the set of all paths that directly or indirectly reference p.
- Select the path that is also trusted by the system administrator (e.g., a special user named root).
- Select the path that was obtained from a substitute referring to a distribution site with some special status.

Example In the description of the resolution algorithm below, we will use the example from Figure 6.1. Thus, we have the following paths in the combined input closures of Firefox, paired with the equivalence classes to which they belong:

$$\begin{aligned}&\{\ (\mathsf{glibc}_A, \mathsf{glibc}_{eq}), (\mathsf{glibc}_B, \mathsf{glibc}_{eq}), (\mathsf{gtk}_A, \mathsf{gtk}_{eq})\\ &,\ (\mathsf{pkgconfig}_A, \mathsf{pkgconfig}_{eq}), (\mathsf{libIDL}_B, \mathsf{libIDL}_{eq})\ \}\end{aligned}$$

Note that glibc_A and glibc_B share the same equivalence class glibc_{eq}, thus

$$\mathsf{eqClassMembers}[\mathsf{glibc}_{eq}] = \{(\texttt{"alice"}, \mathsf{glibc}_A),\ (\texttt{"alice"}, \mathsf{glibc}_B)\}.$$

We have the following references:

$$\begin{aligned}\mathsf{references}[\mathsf{gtk}_A] &= \{\mathsf{glibc}_A\}\\ \mathsf{references}[\mathsf{pkgconfig}_A] &= \{\mathsf{glibc}_A\}\\ \mathsf{references}[\mathsf{libIDL}_B] &= \{\mathsf{glibc}_B\}\end{aligned}$$

Also, the refClasses table tells us what equivalence classes correspond to those references, e.g.,

$$\mathsf{refClasses}[(\mathsf{gtk}_A, \mathsf{glibc}_A)] = \{(\mathsf{gtk}_{eq}, \mathsf{glibc}_{eq})\}$$

and so on.

```
resolve$(paths)$ :
    // Find the conflicts.
    $sources \leftarrow \emptyset$
    for each $(p,e) \in paths$ :
        if $e = \varepsilon : sources \overset{\cup}{\leftarrow} \{p\}$ $\boxed{108}$
        else : $conflicts[e] \overset{\cup}{\leftarrow} \{p\}$ $\boxed{109}$

    // Select one path for each equivalence class.
    $selected \leftarrow$ selectPaths$(conflicts)$ $\boxed{110}$

    $paths' \leftarrow \emptyset$
    for each $(p,e) \in selected$ : $\boxed{111}$
        $paths' \overset{\cup}{\leftarrow} \{$maybeRewrite$(p,\ e,\ selected)\}$
    return $paths' \cup \{(p,\varepsilon) \mid p \in sources\}$ $\boxed{112}$
```

Figure 6.3.: resolve: Collision resolution algorithm

The resolution algorithm Figure 6.3 shows the resolution algorithm. The function resolve accepts a set $paths : \{(\mathsf{Path}, \mathsf{EqClass})\}$ of pairs (p,e) denoting a path p in equivalence class e[3]. The elements in *paths* must be closed under the references relation but may violate the equivalence class unicity invariant; the idea is that *paths* is the union of a number of calls to closure′, defined above. The function resolve yields a closed set of paths that does obey the invariant.

The resolution algorithm works as follows. First, the set of *conflicts* is determined $\boxed{109}$. The mapping $conflicts : \mathsf{EqClass} \to \{\mathsf{Path}\}$ maps the equivalence classes in *paths* to the corresponding paths in *paths*. Note that if any equivalence class in *conflicts* has more than one member in *paths*, there is a conflict. For our example, the conflicts set is as follows:

$$\begin{array}{rr} conflicts[\mathsf{glibc}_{eq}] = \{\mathsf{glibc}_A, \mathsf{glibc}_B\} & conflicts[\mathsf{gtk}_{eq}] = \{\mathsf{gtk}_A\} \\ conflicts[\mathsf{pkgconfig}_{eq}] = \{\mathsf{pkgconfig}_A\} & conflicts[\mathsf{libIDL}_{eq}] = \{\mathsf{libIDL}_B\} \end{array}$$

Thus, there is a conflict in the glibc_{eq} equivalence class. (Some paths are in the "fake" equivalence class ε $\boxed{108}$, returned by closure′ to indicate paths that are not in an equivalence class, such as sources. These are not subject to rewriting, but they are added to the result returned by resolve to ensure the closure property.)

Second, the function selectPaths returns a mapping *selected* from equivalence classes to paths, i.e., $\mathsf{EqClass} \to \mathsf{Path}$. That is, it chooses an element from the members of each equivalence class in *conflicts* $\boxed{110}$. By definition, this defines a set that is free of equivalence class collisions. We do not specify here *how* selectPaths chooses an element from each class. As discussed above, it must implement some policy of selecting a path from each

[3] Again, it is necessary to specify the intended equivalence class because a path may be in multiple equivalence classes. In a previous implementation, a table $\mathsf{eqClasses} : \mathsf{Path} \to \{EqClass\}$ was used to map paths to the equivalence classes to which they belong. This led to a subtly flawed semantics: since we didn't know which particular equivalence class for a path p was intended, we had to use all of them. The result was that a path could be replaced with a path in an entirely different equivalence class.

```
maybeRewrite($p$, $e$, $selected$) :
    $newRefs \leftarrow \emptyset$
    $eqRefs \leftarrow \emptyset$
    $rewrites \leftarrow \emptyset$

    for each $p_{ref} \in \mathsf{references}[p]$ : $\boxed{113}$
        if $p = p_{ref} \lor \neg(\exists e_{ref} : (e, e_{ref}) \in \mathsf{refClasses}[(p, p_{ref})])$ : $\boxed{114}$
            $newRefs \overset{\cup}{\leftarrow} \{p\}$
        else :
            Set $e_{ref}$ such that $(e, e_{ref}) \in \mathsf{refClasses}[(p, p_{ref})]$
            $p_{repl} \leftarrow selected[e_{ref}]$ $\boxed{115}$
            $(p'_{repl}, e'_{repl}) \leftarrow \mathsf{maybeRewrite}(p_{repl},\ e_{ref},\ selected)$ $\boxed{116}$
            // Note that $e_{ref} = e'_{repl}$
            $newRefs \overset{\cup}{\leftarrow} \{p'_{repl}\}$
            $eqRefs \overset{\cup}{\leftarrow} \{(p_{ref}, e, e_{ref})\}$
            $rewrites \overset{\cup}{\leftarrow} \{\mathsf{hashPart}(p_{ref}) \leadsto \mathsf{hashPart}(p'_{repl})\}$ $\boxed{117}$

    if $newRefs = \mathsf{references}[p]$ : return $(p, e)$ $\boxed{118}$

    in a database transaction :
        $p_{new} \leftarrow \mathsf{addToStore}'(\mathsf{readPath}(p),$
           $\mathsf{references}[p],\ eqRefs,\ \mathsf{namePart}(p),\ rewrites,\ \mathsf{hashPart}(p))$ $\boxed{119}$
        $\mathsf{eqClassMembers}[e] \overset{\cup}{\leftarrow} \{(\mathsf{curUser}, p_{new})\}$
    return $(p_{new}, e)$
```

Figure 6.4.: maybeRewrite: Collision resolution algorithm (cont'd)

equivalence class. For our example, there are only two possibilities, namely

$$\begin{array}{rr} selected[\mathsf{glibc}_{eq}] = \mathsf{glibc}_A & selected[\mathsf{gtk}_{eq}] = \mathsf{gtk}_A \\ selected[\mathsf{pkgconfig}_{eq}] = \mathsf{pkgconfig}_A & selected[\mathsf{libIDL}_{eq}] = \mathsf{libIDL}_B \end{array}$$

and

$$\begin{array}{rr} selected[\mathsf{glibc}_{eq}] = \mathsf{glibc}_B & selected[\mathsf{gtk}_{eq}] = \mathsf{gtk}_A \\ selected[\mathsf{pkgconfig}_{eq}] = \mathsf{pkgconfig}_A & selected[\mathsf{libIDL}_{eq}] = \mathsf{libIDL}_B \end{array}$$

Note that the paths of neither set are closed under the references relation. Thus, they are not a valid set of paths that can be used as an input to a build. Indeed, the builder might traverse each path to arrive at its references; if we pick the first set, glibc_B might still be reached by following libIDL_B.

Therefore, the paths in *selected* must be *rewritten* to a new set of paths that *is* closed under the references relation, and still obeys the equivalence class unicity invariant. This is done by the helper function maybeRewrite shown in Figure 6.4, which resolve calls for each selected path $\boxed{111}$. maybeRewrite takes a path p in equivalence class e, and returns a new store path p_{new} in the same equivalence class e.

The basic idea is that the path p_{new} is a copy of p produced by rewriting all references of p that are *not* in *selected*, to equivalent paths that *are* in *selected*. Thus, we walk over all references $\boxed{113}$, and for each reference p_{ref} in equivalence class e_{ref} that is not trivial (i.e., a self-reference to p) or a source (there is no e_{ref}) $\boxed{114}$, the equivalent path $p_{repl} = selected[e_{ref}]$ is chosen $\boxed{115}$. That is, the reference is replaced with the path that was selected for its equivalence class. All these p_{ref}s together form the set of new references *newRefs*. If the new set of references is not equal to the old $\boxed{118}$, a set of rewrites is applied to the contents of p that replaces the hash part of each p_{ref} with the hash part of the corresponding p_{repl} $\boxed{117}$. This rewriting is done by the function $\mathsf{addToStore}'$ (explained below), and yields a new store path p_{new} $\boxed{119}$.

There is a final complication, though; the selected replacement reference p_{repl} may itself not be closed with respect to *selected*, and therefore in need of rewriting. Thus, p_{repl} *itself* is rewritten using maybeRewrite $\boxed{116}$, and it is the resulting path p'_{repl} that is actually used in the new set of references and for the hash rewrites.

The pseudo-code of maybeRewrite shown in Figure 6.4 will typically perform many redundant recursive calls. This is safe, since the implementation of $\mathsf{addToStore}'$ is idempotent. In the actual implementation, simple memoisation is used to remember for each path p in equivalence class e the corresponding rewritten path p_{new}.

In essence, maybeRewrite performs a bottom-up rewrite on the graph of selected paths. Let us illustrate this using our example. Suppose that we have selected the set $\{(\mathsf{glibc}_A, \mathsf{glibc}_{eq}), (\mathsf{gtk}_A, \mathsf{gtk}_{eq}), (\mathsf{pkgconfig}_A, \mathsf{pkgconfig}_{eq}), (\mathsf{libIDL}_B, \mathsf{libIDL}_{eq})\}$. We first call maybeRewrite on $(\mathsf{glibc}_A, \mathsf{glibc}_{eq})$. This path has no references, so $(\mathsf{glibc}_A, \mathsf{glibc}_{eq})$ is returned.

Next, we call maybeRewrite on $(\mathsf{gtk}_A, \mathsf{gtk}_{eq})$. This path *does* have a reference, namely, glibc_A in the equivalence class glibc_{eq}. We replace this reference with the path in *selected* in equivalence class glibc_{eq}, which happens to be glibc_A itself. Recursively applying maybeRewrite to $(\mathsf{glibc}_A, \mathsf{glibc}_{eq})$ yields $(\mathsf{glibc}_A, \mathsf{glibc}_{eq})$. Thus, nothing changes in the references, and $(\mathsf{gtk}_A, \mathsf{gtk}_{eq})$ is returned. The same happens with $(\mathsf{pkgconfig}_A, \mathsf{pkgconfig}_{eq})$.

However, when we call maybeRewrite on $(\mathsf{libIDL}_B, \mathsf{libIDL}_{eq})$, things get more interesting since we *do* need a rewrite. This is because its sole reference, glibc_B is replaced with glibc_A. Therefore, a copy is made of libIDL_B, in which the occurrences of $\mathsf{hashPart}(\mathsf{glibc}_B)$ are replaced with $\mathsf{hashPart}(\mathsf{glibc}_A)$. This yields a new path, libIDL'_B, which has a reference to glibc_A instead of glibc_B. In summary,

$$\begin{aligned}
\mathsf{maybeRewrite}(\mathsf{glibc}_A,\ \mathsf{glibc}_{eq},\ selected) &= (\mathsf{glibc}_A, \mathsf{glibc}_{eq}) \\
\mathsf{maybeRewrite}(\mathsf{gtk}_A,\ \mathsf{gtk}_{eq},\ selected) &= (\mathsf{gtk}_A, \mathsf{gtk}_{eq}) \\
\mathsf{maybeRewrite}(\mathsf{pkgconfig}_A,\ \mathsf{pkgconfig}_{eq},\ selected) &= (\mathsf{pkgconfig}_A, \mathsf{pkgconfig}_{eq}) \\
\mathsf{maybeRewrite}(\mathsf{libIDL}_B,\ \mathsf{libIDL}_{eq},\ selected) &= (\mathsf{libIDL}'_B, \mathsf{libIDL}_{eq})
\end{aligned}$$

Thus the set returned by resolve for our running example is $\{(\mathsf{glibc}_A, \mathsf{glibc}_{eq}), (\mathsf{gtk}_A, \mathsf{gtk}_{eq}), (\mathsf{pkgconfig}_A, \mathsf{pkgconfig}_{eq}), (\mathsf{libIDL}'_B, \mathsf{libIDL}_{eq})\}$. Note that this set is closed under the references relation since libIDL'_B references glibc_A, not glibc_B.

It may not be obvious at first glance that resolve is correct, i.e., that it produces a set of paths that obeys the equivalence class unicity invariant, and is closed. The first property is trivial, since selectPaths by definition yields a set *selected* that obeys the invariant, and we call maybeRewrite once for each $(p,e) \in \mathit{selected}$, yielding a path in the same equivalence class. The second property is more subtle, and is proved as follows.

Theorem 5 (Closure of resolved set)**.** *The paths in the set paths′ returned by the function* resolve *in Figure 6.3 are closed under the* references *relation.*

Proof. The proof is by structural induction. Note that maybeRewrite is called for each (p,e) in *selected*. The induction hypothesis is that the closure of every path returned by a call to maybeRewrite for a path p in equivalence class e is in paths'.

There is one base case: if a path $(p,e) \in \mathit{selected}$ has only trivial or source references, maybeRewrite returns (p,e) unmodified, so $(p,e) \in \mathit{paths}'$. Any non-trivial references are source references and explicitly included in paths' at $\boxed{112}$. Thus the hypothesis holds.

The inductive step is as follows. If a path $(p,e) \in \mathit{selected}$ has non-trivial references, then it is rewritten to a new path (p_{new},e). Each non-trivial, non-source reference p_{ref} of p in equivalence class e_{ref} is rewritten by a recursive call to maybeRewrite on $p_{repl} = \mathit{selected}[e_{ref}]$, i.e., the selected replacement for (p_{ref},e_{ref}). Note that $(p_{repl},e_{ref}) \in \mathit{selected}$, which by the induction hypothesis means that the closure of p'_{repl} is in paths'.

Since this is the case for all p'_{repl}, the closures of all references of the path p_{new} produced by the call to addToStore are in paths', and since p_{new} is returned and added to paths', the closure of p_{new} is also in paths'. □

6.4.2. Adding store objects

Figure 6.5 shows the intensional version of the function addToStore that adds an atom (an FSO) to the Nix store. It is merely a wrapper around a more powerful function, addToStore′, that writes an FSO to the Nix store at its proper content-addressed location. addToStore′ is the only primitive operation that adds valid paths to the store in the intensional model. This is in contrast to the extensional model, where the functions build and substitute also produce valid paths. It takes six arguments:

- *fso* is the FSO to be written to the store.
- *refs* is the set of referenced store paths of the FSO.
- *eqRefs* is a set of tuples (p_{ref},e,e_{ref}) that describe elements to be added to the refClasses mapping. Each tuple signifies that, given a resulting path p, the entry (e,e_{ref}) should be added to $\mathsf{refClasses}[(p,p_{ref})]$.
- *name* is the symbolic name to be used in the construction of the store path.
- *rewrites* is a set of hash rewrites (i.e., $h_1 \leadsto h_2$) to be applied to the FSO *and* to the references *refs* and *eqRefs*.
- *selfHash* is the hash part of the path from which the FSO is being copied. As discussed in Section 6.3.2, self-references must be zeroed out when computing the cryptographic hash of the FSO, and must be replaced with the hash part of the new store path of the FSO. For atoms (such as sources), which originate from outside of

```
$\mathsf{addToStore}(fso,\ refs,\ name)$ :
    return $\mathsf{addToStore}'(fso,\ refs,\ \emptyset,\ name,\ \emptyset,\ \varepsilon)$

$\mathsf{addToStore}'(fso,\ refs,\ eqRefs,\ name,\ rewrites,\ selfHash)$ :
    $c \leftarrow \mathsf{replace}(\mathsf{serialise}(fso),\ rewrites)$ $\boxed{120}$
    if $selfHash \neq \varepsilon$ :
        $h \leftarrow \mathsf{hashModulo}(c,\ selfHash)$ $\boxed{121}$
    else :
        $h \leftarrow \mathsf{hash}_{\mathsf{sha256}}(":" + c)$ $\boxed{122}$
    $p \leftarrow \mathsf{makePath}(h,\ name)$ $\boxed{123}$
    if $\mathsf{valid}[p] = \varepsilon$ :
        $lock \leftarrow \mathsf{acquirePathLock}(p)$
        if $\mathsf{valid}[p] = \varepsilon$ :
            if $selfHash \neq \varepsilon$ :
                $c \leftarrow \mathsf{replace}(c,\ \{selfHash \rightsquigarrow \mathsf{hashPart}(p)\})$ $\boxed{124}$
                $rewrites \overset{\cup}{\leftarrow} \{selfHash \rightsquigarrow \mathsf{hashPart}(p)\}$
            $\mathsf{writePath}(p,\ \mathsf{deserialise}(c))$
            in a database transaction :
                $\mathsf{valid}[p] \leftarrow h$
                $refs \leftarrow \{\mathsf{replace}(p,\ rewrites) \mid p \in refs\}$ $\boxed{125}$
                $\mathsf{setReferences}(p,\ refs)$
        $\mathsf{releasePathLock}(p,\ lock)$
    for each $(p_{ref}, e, e_{ref}) \in eqRefs$ :
        $\mathsf{refClasses}[(p, \mathsf{replace}(p_{ref},\ rewrites))] \overset{\cup}{\leftarrow} \{(e, e_{ref})\}$ $\boxed{126}$
    return $p$
```

Figure 6.5.: addToStore: Adding store objects to a content-addressable store

the Nix store, no such self-reference is present in the FSO; the dummy value ε must then be provided for *selfHash*. But when *copying* store paths, such as in the collision resolution algorithm (Figure 6.3) or in the build algorithm (Figure 6.6), *selfHash* is set to the hash part of the source path.

The function addToStore′ works as follows. First, the hash rewrites are applied to the serialisation of the FSO $\boxed{120}$. Next, the hash modulo self-references is computed using the function hashModulo (page 144) $\boxed{121}$. However, this is only applicable when copying inside the store, that is, when *selfHash* is not ε. If it is ε, the hash is computed normally over the serialisation, but with a colon prepended $\boxed{122}$. This exactly matches a hashModulo computation over contents that contain no self-references, since in that case we have

$$\begin{aligned}
\mathsf{hashModulo}(c,\ hp) &= \mathsf{hash}_{\mathsf{sha256}}\Big(\sum_{i \in \mathsf{find}(c,hp)} (i + ":") + ":" + \mathsf{replace}(c,\ \{hp \rightsquigarrow \mathbf{0}\})\Big) \\
&= \mathsf{hash}_{\mathsf{sha256}}\Big(\sum_{i \in \emptyset} (i + ":") + ":" + \mathsf{replace}(c,\ \{hp \rightsquigarrow \mathbf{0}\})\Big) \\
&= \mathsf{hash}_{\mathsf{sha256}}(":" + c)
\end{aligned}$$

and so *selfHash* is not needed.

The hash and symbolic name are used to construct a store path p $\boxed{123}$. Then, just as in the extensional version of addToStore, we check if p is already valid, and if not, acquire an exclusive lock on p and recheck for validity. If the path is still not valid, *selfHash* is rewritten to hashPart(p) (if applicable) $\boxed{124}$, and the FSO is written to p. The path is then registered as valid. But note that the hash rewrites that have been applied to the contents c are also applied to the set of references $\boxed{125}$. This ensures, for instance, that the old references passed in maybeRewrite $\boxed{119}$ are changed to the new references. Likewise, appropriate refClasses entries are added as described above $\boxed{126}$.

6.4.3. Building store derivations

Figure 6.6 shows the build algorithm for the intensional model. As in Section 6.2, we assume that all operations on the Nix store are done by a privileged user on behalf of the actual users, who do not have write access themselves.

The most important differences with the build algorithm for the extensional model (Figure 5.11) are as follows. Since there is no longer a single output that can be substituted or checked for validity, we have to consider *all* paths in the output equivalence class. Thus, we query the set of *trusted* paths in the equivalence class d.eqClass $\boxed{127}$. We then check whether any of those paths are valid; if so, we're done $\boxed{128}$. If not, we try to create one through substitutes $\boxed{129}$.

If all of the substitutes fail, we have to perform the build. We first recursively build all input derivations. Just as in the extensional build algorithm, the set *inputs* of input paths is computed. However, this set may contain equivalence class collisions. Thus, those conflicts must be resolved $\boxed{130}$. The resulting set *inputs* is a closure that obeys the equivalence class unicity invariant.

Recall that the envVars, args and builder fields of the derivation cannot refer to the output paths of the actual input derivations, because those are not yet known at instantiation time. Rather, these fields refer to the output equivalence classes of the input derivations. Now that we have actual paths for the inputs, we can rewrite the hash parts of the equivalence classes to the hash parts of the actual paths $\boxed{131}$.

The build is performed with the environment variable out set to a temporary path in the Nix store $\boxed{132}$. The temporary output path is computed by replacing the hash part of the output equivalence class with a random hash part of the same length, as described in Section 6.3.2.

After the build, we copy and rewrite the temporary path to its final, content-addressable location in the Nix store $\boxed{133}$. The temporary path can now be deleted. The new store path is registered as being inside the equivalence class d.eqClass $\boxed{134}$.

It is worth noting that the intensional build algorithm does not perform locking. It does not have to. The random temporary path can be assumed to be unique if the random-number generator is sufficiently cryptographically strong, or we can simply check whether it already exists, and if so, pick another path. Since the temporary path is unique, there can be no interference from other Nix processes building the same derivation. The absence of locking has the advantage that it simplifies the implementation and prevents a malevolent user from causing a livelock (e.g., by registering a substitute for a path that does not terminate). On the other hand, it may also create work duplication if multiple processes perform

```
build($p_{drv}$) :
    if $\neg$ substitute($p_{drv}$) : abort
    $d \leftarrow$ parseDrv(readPath($p_{drv}$))

    // Note: curUser is the invoking user.
    $trusted \leftarrow$ trustedPaths($d$.eqClass, curUser) $\boxed{127}$
    for each $p \in trusted$ : if valid[$p$] $\neq \varepsilon$ : return $p$ $\boxed{128}$
    for each $p \in trusted$ : if substitute($p$) : return $p$ $\boxed{129}$

    // Gather all trusted input closures, then resolve.
    $inputs \leftarrow \emptyset$
    for each $p \in d$.inputsDrvs :
        $d' \leftarrow$ parseDrv(readPath($p$))
        build($p$)
        for each $p' \in$ trustedPaths($d'$.eqClass, curUser) :
            $inputs \overset{\cup}{\leftarrow}$ closure'($p'$, $d'$.eqClass)
    for each $p \in d$.inputsSrcs :
        $inputs \overset{\cup}{\leftarrow}$ closure'($p$, $\varepsilon$)
    $inputs \leftarrow$ resolve($inputs$) $\boxed{130}$

    // Rewrite equivalence classes to real paths.
    $mapping \leftarrow \emptyset$
    for each $(p, e) \in inputs$ :
        $mapping \overset{\cup}{\leftarrow} \{$hashPart($e$) $\rightsquigarrow$ hashPart($p$)$\}$
    Apply the rewrites $mapping$ to $d$.envVars, $d$.args and $d$.builder $\boxed{131}$

    // Build in a temporary path.
    $output \leftarrow$ replace($d$.eqClass, $\{$hashPart($d$.eqClass) $\rightsquigarrow$ randomHash()$\}$)
    $d$.envVars["out"] $\leftarrow output$ $\boxed{132}$
    Run $d$.builder in an environment $d$.envVars
       and with arguments $d$.args and in a temporary directory $p_{tmp}$
    Canonicalise $output$ & perform fixed-output derivation tests as in Figure 5.11

    // Copy to content-addressed location and register as valid.
    $refs \leftarrow$ scanForReferences($output$, $\{p \mid (p, e) \in inputs\}$)
    in a database transaction :
        $output' \leftarrow$ addToStore'(readPath($output$), $refs$,
           $\{(p_{ref}, d$.eqClass$, e_{ref}) \mid (p_{ref}, e_{ref}) \in inputs \wedge p_{ref} \in refs\}$,
           namePart($d$.eqClass), $\emptyset$, hashPart($output$)) $\boxed{133}$
        eqClassMembers[$d$.eqClass] $\overset{\cup}{\leftarrow} \{$(curUser, $output'$)$\}$ $\boxed{134}$
        return $output'$
```

Figure 6.6.: build: Building store derivations in the intensional model

exactly the same build. This is safe, however: even if multiple builds produce exactly the same outputs, the idempotency and transactional semantics of $\mathsf{addToStore}'$ ensure that the resulting path is not written to twice.

Builder write permission A trivial but annoying problem is how we can allow builders to create their output path, i.e., the path specified in the environment variable out. After all, the builder is not supposed to have write permission to the Nix store! Thus a builder should not be allowed to perform:

```
mkdir $out
```

since that implies that it can also create *other* paths in the store. The prototype implementation of the intensional model does not yet solve this problem (builders still have arbitrary write permission to the store). There are a number of possible solutions. On Unix, the Nix store directory could be given the *sticky permission bit* (S_ISVTX) [152], which prevents users from deleting or renaming directories not owned by them. Under the secure local build scheme from Section 6.2, the builder runs under a unique uid that differs from the uid that owns the valid FSOs, and from the uids of other concurrently running builders. Thus, the builder can create other paths than its expected output, but these can be deleted afterwards, and it cannot modify valid paths.

Another solution is to have the calling Nix process create the output path on behalf of the builder. The builder then does not need write permission to the store directory itself. However, Nix must know the *type* of the output path (i.e., regular file, directory, or symlink) ahead of time. Finally, the builder could *ask* the calling Nix process to create the output path of the appropriate type, e.g., by talking to it through a socket.

6.4.4. Substitutes

Just as the table eqClassMembers ties equivalence class membership to users, we extend the substitutes mapping (first defined in Section 5.5.3) with a UserId for each substitute to indicate which user registered it, i.e., $\mathsf{substitutes} : \mathsf{Path} \rightarrow \{(\mathsf{UserId}, \mathsf{Path}, [\mathsf{String}], \mathsf{Path})\}$. This means that a substitute is a 4-tuple consisting of the ID of the user who registered the substitute, the path of the substitute program, its command-line arguments, and the deriver (page 114). For example,

$$\begin{array}{l}
\mathsf{substitutes}[\text{/nix/store/bpv6czrwrmhr......-hello-2.1.1}] = \\
\quad \{\ (\ \text{"alice"},\ \text{"download-url.sh"} \\
\qquad ,\ [\text{"http://example.org/0n4laxf6kq44....nar.bz2"}] \\
\qquad ,\ \text{"/nix/store/1ja1w63wbk5q...-hello-2.1.1.drv"}\ ) \\
\quad ,\ (\ \text{"bob"},\text{"download-url.sh"} \\
\qquad ,\ [\text{"http://example.org/r35w1y8xll12....nar.bz2"}] \\
\qquad ,\ \text{"/nix/store/1ja1w63wbk5q...-hello-2.1.1.drv"}\ ) \\
\quad \}
\end{array}$$

Adding a UserId is not strictly necessary to ensure the trust property. After all, due to content-addressability, if a substitute for path p fails to produce a path with a content hash

```
Bool substitute($(p)$) :
    if valid$[p] \neq \varepsilon$ : return true
    $subs \leftarrow$ trustedSubs$(p,$ curUser$)$ $\boxed{135}$
    if $subs = \emptyset$ : return false
    for each $p' \in$ references$[p]$ :
        if $\neg$ substitute$(p')$ : return false

    for each $(user, program, args, deriver) \in$ substitutes$[p]$ :
        $p_{tmp} \leftarrow$ replace$(p,$ {hashPart$(p) \rightsquigarrow$ randomHash()}$)$ $\boxed{136}$
        if execution of $program$ with arguments $[p_{tmp}] + args$ exits with exit code 0 :
            assert(pathExists$(p_{tmp})$)
            $p' \leftarrow$ addToStore$'$(readPath$(p_{tmp})$, references$[p]$, $\emptyset$
                namePart$(p_{tmp})$, $\emptyset$, hashPart$(p_{tmp})$) $\boxed{137}$
            if $p = p'$ :  return true $\boxed{138}$

    if fall-back is disabled : abort
    return false
```

Figure 6.7.: substitute: Substitution of store paths in the intensional model

that corresponds with hashPart(p), the substitute can simply be considered to have failed, and its output in p can be deleted. However, we do not want users to trigger other users' substitutes, as that can lead to privacy violations. For instance, Alice could register a substitute that as a side effect informs her when other users try to install certain paths.

Another necessary change is that when a substitute for path p is registered, refClasses entries for the references of p must also be registered. This means that the command nix-store --register-substitutes (Section 5.5.3) must be extended to accept a list of (p_{ref}, e, e_{ref}) entries that can be added to refClasses.

Figure 6.7 shows the substitution function substitute(p). The main difference with the extensional version (Figure 5.15) is that the substitute program must produce its output in a temporary path p_{tmp} $\boxed{136}$ which is then copied (using addToStore$'$) to its content-addressed location $\boxed{137}$. Note that it can simply pass $\emptyset$ for the *eqRefs* argument of addToStore$'$, since the refClasses entries for p have already been set when the substitute was registered.

If the resulting path p' does not match with the path p that the substitute is supposed to produce, the substitute is ignored and the next substitute is tried $\boxed{138}$. Also, only substitutes registered by the current user are considered $\boxed{135}$. Analogously to trustedPaths, the function trustedSubs$(p,$ *user*$)$ yields from substitutes$[p]$ those substitutes register by *user*.

Security of references When a substitute is registered, the caller must also provide a list of references and refClasses entries for the path. However, that information may be wrong—the user could supply garbage. For instance, the references supplied by the user could be incomplete. This could cause incomplete deployment: a path p realised through a substitute could have dangling pointers. However, this is only a problem if the path is used by other users. Those users presumably first register their own substitutes before they build a derivation that produces p. As part of the substitute registration, bona fide users must supply a full set of references. A simple solution is therefore as follows: when users register additional references for a path that is already valid, the additional references must be realised through substitutes immediately, then added to the references of p.

Likewise, the registered references can contain bogus paths, i.e., paths that are not actually references. This can lead to an unsubstitutable path, e.g., where a user cannot substitute path p because another user registered a bogus (unsubstitutable) reference p_{ref}. A simple solution is to substitute p *before* we substitute its references (but not before making it valid, of course), then scan p to check what declared references actually occur. The bogus references can then be tossed out.

6.5. Trust relations

The intensional model described in the previous section gives us a Nix store that can be shared by mutually untrusted users, or users who have different remote trust relations. Due to content-addressability, we get sharing between multiple builds of a derivation if each build produces exactly the same binary result, that is, if there is no impurity in the build. If there *is* impurity, then each build result will end up under a different store path.

Between untrusted users, this is exactly what we want. If Alice obtains substitutes from a malicious machine, it does not affect Bob. Note that Alice and Bob do automatically get sharing if they happen to get their substitutes from the same remote machine.

However, we wish to re-enable sharing in common scenarios. For instance, users generally trust components installed by the administrator. Thus, if Alice is an administrator, then Bob should be able to use the output paths already installed by Alice. In general, users should be able to specify *trust relations* between each other.

We can achieve this through a simple extension of the intensional model called the *mixed model*. For each user, we maintain a mapping $\mathsf{trustedUsers} : \mathsf{UserId} \rightarrow \{\mathsf{UserId}\}$ that specifies for each user a set of trusted users. E.g., $\mathsf{trustedUsers}[\mathsf{"bob"}] = \{\mathsf{"alice"}, \mathsf{"bob"}\}$. (The mapping should be reflexive, i.e., $u \in \mathsf{trustedUsers}[u]$.) We then augment the function trustedPaths:

$$\begin{array}{l}\mathsf{trustedPaths}(eqClass, user) = \\ \quad \{p \mid \exists u \in \mathsf{trustedUsers}[user] : (u,p) \in \mathsf{eqClassMembers}[eqClass]\}\end{array}$$

Otherwise, this is exactly the intensional model. Of course, sharing between users increases the possibility of equivalence class collisions, but that is handled through the resolution algorithm in Section 6.4.1. The name “mixed model” does not imply that we back away from intensionality—the store is still fully content-addressed. We just gain back the ability to have sharing between users. The crucial difference with the extensional model is that sharing is now selective and fine-grained.

6.6. Related work

Nix’s transparent source/binary model is a unique feature for a deployment system. Relative to binary-only or source-only deployment models, it adds the complication that we do not only need to authenticate binaries but also the fact that they are a bona fide result of certain sources. However, caching and sharing between users of build results is a feature of some SCM systems such as Vesta [92].

As claimed in the introduction, deployment systems tend to have monolithic trust models. Typical Unix package management systems such as RPM [62], Debian APT, or Gentoo Linux [77] allow installation of software by the administrator only; software installed by individual users is not managed by those tools. On the other hand, Mac OS X application bundles may be installed and moved around by any user, but the system does not track dependencies in any way.

Security aspects of deployment have typically focused on ensuring integrity of components in transit (e.g., by using signatures), and on assessing or constraining the impact of a component on the system (e.g., [173]). We have not addressed the issue of ensuring that remote substitutes have not been tampered with (e.g., by a man-in-the-middle). Obviously, such problems can be solved by cryptographically signing substitutes—or rather the manifests, since the fact that substitutes themselves have not been tampered with is easily verified by comparing their cryptographic hashes to their names.

Microsoft's .NET has a Global Assembly Cache that permits the sharing of components [154]. It is however not intended for storage of components private to an application. Thus, if multiple users install an application having such private components, duplication can occur. Also, .NET has a purely binary deployment model, thus bypassing source/binary correspondence trust issues.

In [80] a scenario is described in which components impersonate other components. This is not possible in a content-addressable file system with static component composition (e.g., Unix dynamic libraries with RPATHs pointing to the full paths of components to link against, as happens in the Nix Packages collection).

Content-addressability is a common property of the distributed hash schemes used in peer-to-peer file-sharing and caching applications (e.g., Pastry [144]). It is also used in the storage layer of version management tools such as Monotone [95].

6.7. Multiple outputs

Apart from secure sharing, a major advantage of the intensional model is that it allows derivations to have *multiple outputs*. Derivations as we have defined them up till now produce a single output, i.e., just one store path in the Nix store. However, in many circumstances it is very useful to allow a derivation to produce multiple outputs, i.e., multiple store paths. This allows finer-grained deployment, reducing the size in bytes of the closures that need to be deployed to and stored on client machines.

For instance, the GNU C Library component, Glibc, contains parts that are not actually needed at runtime by the components that depend on it. Among these parts is its subdirectory /include that contains 2.8 MiB of C header files and is only needed at build time. If we perform a binary deployment of a dependent component to clients, it is quite wasteful that these header files are copied also. In fact, the Glibc header files contain a symlink to a different component, the Linux Kernel headers, which take up 7.6 MiB. So the unnecessary Glibc header files drag along another entirely unnecessary component! Incidentally, this is why the Hello example in Chapter 2 had a retained dependency on linux-headers (page 41).

Multiple outputs are not yet implemented in Nix, so there is no concrete syntax for specifying them in Nix expressions. A possible syntax is as follows. A derivation with multiple outputs specifies a list of symbolically named outputs in the attribute outputs. The Glibc derivation would specify the following:

```
derivation { ...
  outputs = ["out" "dev" "bin"]
}
```

This declares three outputs. The output out contains the bulk of the component, i.e., the C library. The output dev contains development-only parts, such as the header files mentioned above. The output bin contains utility programs shipped with the C library, which like the header files are not necessary for most users.

The effect of the outputs attribute is that the resulting store derivation specifies not one output equivalence class, but rather several, identified by their symbolic names, e.g.,

$$\begin{aligned} d_{glibc}.\mathsf{eqClasses} = \{ & (\text{"out"}, \text{"/nix/store/jzhnqh7ffq72...-glibc-2.3.5"}) \\ , & (\text{"dev"}, \text{"/nix/store/34z5jyqqs1sf...-glibc-2.3.5-dev"}) \\ , & (\text{"bin"}, \text{"/nix/store/76zsvfw4zizs...-glibc-2.3.5-bin"})\} \end{aligned}$$

Conversely, a way is needed for dependent components to refer to specific outputs. A component foo can specify that it needs the out and dev outputs of Glibc, but (by omission) not bin:

```
{glibc}: derivation { ...
  glibc = glibc.out; # can be shortened to just "glibc"
  glibcHeaders = glibc.dev;
}
```

To represent this in store derivations, the elements of the inputDrvs field become tuples that specify *which* output of an input derivation is needed for a build, e.g.,

$$d_{foo}.\mathsf{inputDrvs} = \{\ (p_{d,glibc}, \{\text{"out"}, \text{"drv"}\}),\ ...\}$$

where $p_{d,glibc}$ is the store path of the derivation containing d_{glibc}. In fact, the on-disk ATerm representation of store derivations already supports this extension (see Figure 5.8).

So why are multiple outputs impossible in the extensional model? Imagine that we install Hello from source, and add it to a user environment. Hello has a reference to the out output of Glibc, but not the bin and dev outputs. Thus, when we run the garbage collector, those outputs will be deleted (if they were present to begin with). Suppose that we subsequently build a derivation that has a dependency on (say) the bin output of Glibc. Unless we have a substitute that builds that path, we now have an unsolvable problem. The only way that we can obtain the path without a substitute is to build Glibc. But the Glibc build would overwrite its own out output, which is already valid, and thus should never be modified. (If the overwriting is atomic or otherwise unobservable, there is no problem, but that is not the case in general.) So we need to delete the out output first. But to delete it violates the closure invariant, as that path is reachable from a user environment. So the Nix store is "jammed": we have reached a state where a build simply cannot be done because it is blocked by the validity of some of its outputs.

In the intensional model, this problem does not exist. A build can always be done, because the outputs go to temporary paths that are constructed on the fly. We can simply perform a build to obtain all of its outputs in temporary paths, copy the missing outputs to content-addressed locations, and discard the outputs that we already have. In the example above, we perform a full build of Glibc. The outputs end up in temporary paths p_{out}, p_{dev}, and p_{bin}. We finalise the previously missing outputs p_{dev} and p_{bin} using addToStore′, and discard p_{out} as we already have it. The FSOs in p_{dev} and p_{bin} might have references to p_{out}, but those can be rewritten to the path of the out output that we already had.

The outputs of a derivation can have references to each other, and in fact this is quite common. For instance, it can be expected that the programs in the bin output of Glibc depend on the libraries in the out output. This means that the out output is in the closure of the bin output, but not *vice versa*. But what happens when there are mutually recursive references, e.g., when out also refers to bin? These must be forbidden, since the hash rewriting scheme from Section 6.3.2 cannot handle them. For instance, when we copy out and bin to their content-addressable locations, we must rewrite in both FSOs the hashes of both paths. The function hashModulo only handles direct self-references, and it can do so because the hashes to be disregarded in the hash computation are encoded into the file name.

Fortunately, banning mutually recursive outputs is not a very onerous restriction, since they are pointless. After all, mutual recursion between output paths requires them to be deployed and garbage collected as a unit, negating the granularity advantages that multiple outputs are intended to achieve.

6.8. Assumptions about components

It is useful to make explicit at this point what assumptions Nix makes about components, i.e., the requirements imposed on components in order to allow them to be deployed through Nix. These assumptions are:

- The component can be built with and installed at an arbitrary prefix. *Counterexample of a component that violates this assumption*: some Linux system components have Makefiles that contain "hard-coded" paths such as /lib. (Such Makefiles are easily patched, though.)

- Likewise, the component does not expect that its dependencies are installed at certain fixed locations, and the locations of the dependencies can instead be specified at build or runtime. *Counterexample*: the GNU C Library, Glibc, has a hard-coded dependency on the POSIX shell in /bin/sh.

- The component can be built automatically, i.e., with no user interaction whatsoever. *Counterexample*: the Unreal Tournament 2004 Demo has an interactive installer that among other things asks the user to approve a license agreement. (Nevertheless it is in Nixpkgs, since we can bypass the installer by unpacking the TAR archive embedded in the installer's executable.)

- The build process of the component is essentially pure (i.e., deterministic), meaning that the output of the build process is fully determined by the declared inputs. *Counterexample*: any builder that depends on a mutable network resource. This is why fetchurl checks the downloaded file against a pre-specified cryptographic hash. *Another counterexample*: the archiving tool for object files (ar) on Mac OS X stores timestamps in archives, and these must not be newer than the timestamp on the archive itself; otherwise, the linker will refuse to use the archive. (Since Nix canonicalises timestamps to 1970 through canonicalisePathContents (page 112), i.e., back in time, this is not a problem.)

- The component does not need to be modified after it has been built, e.g., at runtime. *Counterexample*: Firefox 1.0 needs to be run interactively with write permission to its installation directories so that it can create some additional files. (However, there is an obscure, poorly documented solution for this that allows these files to be generated non-interactively.)

- The component has no environment dependencies. For instance, the component should not require global registry settings. It is however fine to use automatically created per-user configuration files or state. In other words, it should not be necessary to run a program to realise the required environment; the closure in the Nix store should be the only dependency. *Counterexample*: system daemons (server processes) that require the existence of certain user accounts.

- Retained dependencies can be found in the component through scanning by scanForReferences. There is no pointer hiding.

The intensional model makes one additional assumption:

- Hash rewriting does not change the semantics of the component. For instance, the component has no internal checksums that are invalidated by hash rewriting.

If these assumptions hold for a component, then it can be deployed through Nix. Of course, this raises the obvious question of whether these assumptions are likely to hold for components in practice. In the next chapter, we will see that this is indeed the case.

Part III.

Applications

7. Software Deployment

The purpose of the Nix system is to support software deployment. In Part II we have seen the low-level *mechanisms* necessary to support deployment. This chapter deals with the higher-level mechanisms and actual policies that enable support for a wide range of deployment scenarios. It also serves as *validation* for the Nix approach, since as we have just seen in Section 6.8, Nix makes a number of assumptions about components that may not always hold in practice. To discover to what extent these assumptions are valid, we have applied Nix to the deployment of a large set of pre-existing software components, the *Nix Packages collection*.

This chapter first discusses the Nix Packages collection and the design principles that went into it, as these are instructive for the use of Nix in general.

Second, it shows concrete "high-level" deployment mechanisms. The nix-pull mechanism is built on top of the low-level substitute mechanism to enable a variety of deployment policies, such as channels and one-click installations. User environments are "pseudo-components" that are constructed automatically by nix-env to activate components.

Third, Nix's purely functional model creates some unique challenges in supporting the evolution of software installations over time. For instance, users will want to periodically upgrade software components. In a binary deployment policy, it is undesirable if all changed components must be re-downloaded in their entirety, particularly because in the purely functional model all dependent components change as well. Section 7.5 shows that this problem can be ameliorated using *binary patching*.

Finally, Section 7.6 compares Nix to other deployment systems, and to techniques that might be used to implement deployment.

7.1. The Nix Packages collection

As we have seen in Chapter 2, the Nix Packages collection (Nixpkgs) is a set of Nix expressions that evaluate to derivations for a large set of pre-existing, third-party software components. Figure 7.1 shows 278 components for which Nixpkgs contains expressions[1]. For some of the components, Nixpkgs provides multiple versions. In general, we endeavour to standardise on a single version of a component within a given release of Nixpkgs, but this is not always possible. For instance, several versions of the GNU C Compiler are necessary because there is no single version of the compiler that can build all components. Also, some library components have major revisions that are not backwards compatible (e.g., versions 1 and 2 of the GUI library GTK). Of course, the great thing about the Nix expression language is that it makes it so easy to deal with this kind of variability.

The components in Nixpkgs are a reasonable cross section of the universe of Unix software components, though with a bias towards open source components. This is because

[1] This set is based on nixpkgs-0.9pre3415.

a52dec acrobat-reader alsa-lib ant-blackdown ant-j2sdk apache-httpd aterm aterm-dynamic aterm-java atk audiofile autoconf automake bash bibtex-tools binutils bison bittorrent blackdown boehm-gc bsdiff bzip2 cdparanoia chmlib cil cksfv clisp coreutils cua-mode curl db4 diffutils docbook-xml docbook-xml-ebnf docbook-xsl e2fsprogs eclipse-sdk ed emacs enscript esound exif expat file findutils firefox flac flashplayer flex fontconfig freetype f-spot gail gawk gcc gcc-static GConf gdk-pixbuf generator getopt gettext ghc ghostscript glib glibc gnet gnome-desktop gnome-icon-theme gnome-keyring gnome-mime-data gnome-panel gnome-vfs gnugrep gnum4 gnumake gnupatch gnuplot gnused gnutar gperf gqview graphviz grub gtk+ gtkhtml gtkmozembed-sharp gtk-sharp gtksourceview gtksourceview-sharp guile gwydion-dylan gzip happy harp haskell-mode helium hello help2man hevea intltool iputils j2sdk jakarta-bcel jakarta-regexp jakarta-tomcat jclasslib jdk jetty jikes jing-tools jjtraveler jre kaffe lame lcms lcov less libart_lgpl libbonobo libbonoboui libcdaudio libdvdcss libdvdplay libdvdread libexif libglade libgnome libgnomecanvas libgnomeprint libgnomeprintui libgnomeui libgphoto2 libgtkhtml libICE libIDL libjpeg libmad libogg libpng libsigsegv libSM libtheora libtiff libtool libvorbis libwnck libX11 libXau libXaw libXext libXft libXi libXinerama libxml2 libXmu libXp libXpm libXrender libxslt libXt libXtrans libXv libXxf86vm lynx mesa mingetty mjpegtools mktemp modutils mono monodevelop mono-dll-fixer monodoc mpeg2dec MPlayer mplayerplug-in mysql mythtv nano nasm ncurses net-tools nix nmap nxml-mode ocaml octave openssh openssl ORBit2 pan pango panoramixext par2cmdline patchelf pcre perl perl-BerkeleyDB perl-DateManip perl-HTML-Parser perl-HTML-Tagset perl-HTML-Tree perl-libwww-perl perl-LocaleGettext perl-TermReadKey perl-URI perl-XML-LibXML perl-XML-LibXML-Common perl-XML-NamespaceSupport perl-XML-Parser perl-XML-SAX perl-XML-Simple perl-xmltv perl-XML-Twig perl-XML-Writer pkgconfig popt postgresql postgresql-jdbc procps pygtk python qt quake3demo rcs readline RealPlayer renderext rte ruby saxon saxonb screen scrollkeeper sdf2-bundle SDL shadow shared-objects sqlite stdenv-linux strace strategoxt strategoxt-utils subversion swig sylpheed sysvinit tetex texinfo thunderbird uml uml-utilities unzip ut2004demo util-linux valgrind vim vlc wget which wxGTK wxPython xchm xextensions xf86vmext xfree86 xine-lib xine-ui xlib xpf xproto xsel zapping zdelta zip zlib zoom zvbi

Figure 7.1.: Components in the Nix Packages collection

these components are most readily available, and allow third-party use and confirmation of our results. However, some binary-only components are also included, such as Adobe Reader, Real Player, Sun's Java 2 SDK, and the Unreal Tournament 2004 Demo. How we support such components is discussed in Section 7.1.4.

In terms of application areas, the set of components spans a wide spectrum:

- Fundamental tools and libraries (Glibc, GCC, Binutils, Bash, Coreutils, Make, ...).
- Other compilers and interpreters (Perl, Python, Mono, Sun's Java 2 JDK, ...).
- GUI libraries (X11 client libraries, GTK, Qt, ...).
- Other libraries (Berkeley DB, libxml, SDL, and *many* more).
- Developer tools (Eclipse, Emacs, Valgrind, Subversion, Autoconf, Bison, ...).
- Non-developer tools (teTeX, NMap, ...)
- Servers (Apache httpd, PostgreSQL, MySQL, Jetty, ...).

- End-user applications (Firefox, Xine, Gaim, UT2004, ...).

Likewise, several programming languages are represented, including C, C++, Java, C#, Haskell, Perl, Python, and OCaml.

Nixpkgs does not contain any of these components itself; it only contains Nix expressions and builders. The source or binary distribution files for the components are obtained using fetchurl calls. This is a policy decision; it is perfectly possible to include these files in the Nixpkgs distribution itself—but that would make a Nixpkgs download several hundreds of MiBs large (not to mention that it might violate several licenses).

Nixpkgs is severely biased towards Linux-based systems. Many of the components in Nixpkgs also build on other Unix-based operating systems, such as FreeBSD and Mac OS X, but as we shall see in Section 7.1.2, the standard environment on Linux is far superior to those on other platforms as it produces components that do not require any external (non-Nix) components to build or run. For instance, it uses its own copy of the GNU C Library (Glibc). In addition, to build the standard environment on Linux, no external components are needed, i.e., it is fully *self-contained*. Significantly, this means that Nixpkgs on Linux is almost perfectly insulated from the "host" operating system. A component that builds and runs on, say, Red Hat Enterprise Linux, will also build and run on SuSE Linux.

This bias towards Linux is not the result of any fundamental aspect of Nix, but rather is triggered by two important considerations:

- The fundamental components of Linux systems[2] are all open source. This makes them much more useful for experimentation and adaptation than closed-source components. In contrast, in Mac OS X, many fundamental components (such as the GUI libraries) are closed source, which means that we cannot build them in the Nix store.

- Linux is highly component-based, much more than any other system, in the sense that the components that make up a Linux system are all separately maintained, built, and distributed. This is probably bad for users, as it creates a lack of vision and coordination between the various parts of the system, but it is very good for us. Consider by contrast FreeBSD, which is also open source, but only available as a unified whole. That is, components such as the kernel, C library, compiler, basic Unix tools, system administration tools, documentation, etc., are all part of a single distribution. This means that we cannot easily take out, say, the C library, and use it in our standard environment. Also, since Linux components must support a wide variability in the environments in which they are used, they are typically quite configurable. For instance, Glibc can be installed at a non-standard prefix. So Linux is a truly component-based system in the sense of components in Section 3.1.

But again, other Unix systems are also supported, if in a somewhat less perfect state. The open source nature of Linux enables us to explore an "ideal", uncompromising environment on Linux, where *everything* is deployed through Nix. On other systems, we have to allow a certain measure of impurity. For instance, components must use system libraries that are not built or deployed through Nix.

[2] Of course, we cannot really speak of Linux systems in general, as Linux is just a kernel. There is no single component other than the kernel that is common to all Linux-based systems (not even Glibc!). However, there is a fuzzy set of components that make up typical Linux environments, e.g., Glibc, GCC, Bash, GTK, Qt, Gnome, KDE, and so on.

So what about non-Unix systems, in particular Microsoft Windows? I claimed in the introduction that Nix does not require any operating system specific extensions, such as virtual file systems (as used by, e.g., Zero Install [112]). But Nix does make one Unix-centric assumption: that *paths abstract over storage devices*. For binary deployment to work, it is important that the store is in the same path between systems. That's why we more-or-less insist on using /nix/store. On Unix, this fixed path is not a problem, because the directory /nix can be mounted on any physical device. But on Windows, if we use a path such as C:\nix\store, we encode the device on which the store resides (e.g., C:) into the store location. If another user uses D:\nix\store, e.g., because her D: has sufficient free space, she cannot pull substitutes from a cache that assumes C:\nix\store. Thus we run afoul of Windows's CP/M heritage. (This does not apply to Cygwin [88] or similar environments, which provide a Unix-like file system view on Windows.) However, it is a little-known feature that Windows also provides mount points[3], so this is in fact not a fatal problem.

7.1.1. Principles

There are a number of design principles guiding the construction of the components in Nixpkgs. The most important of these is the following:

⇒ "Static compositions are good."

A *static composition* is one that is established at component build time. A *dynamic composition* is one that is established at runtime, i.e., after the components have been built and deployed.

Suppose that we have a component that needs to call a program foo, provided by another component. Dynamic composition, or *late binding*, means that we expect that at runtime, we can find foo somewhere in the PATH environment variable, and so the composition mechanism in C might be:

```
execlp("foo",args);
```

In a static composition, on the other hand, the path of foo is already known at build time:

```
execl("/nix/store/4sqiwbf0kj22...-foo-1.3.1/bin/foo",args);
```

Of course, the path /nix/store/4sqiwbf0kj22...-foo-1.3.1 would not be hard-coded into the source. Rather, the foo component should be given as an input to the derivation, and the path should be filled in by the preprocessor:

```
execl(FOO_PATH "/bin/foo",args);
```

So why is the latter better? The answer is that it gives a more exact closure and more predictable runtime behaviour. The component will always be able to call foo, as it is contained in the component's closure and so is always present. With dynamic composition, foo might be missing, or it might be a wrong version.

An important example of the preference of static composition over dynamic composition is the style of linking used in Nixpkgs. Of course, the ultimate form of static composition is static libraries (which are linked into the images of the executables that use them), but those are inefficient. They waste disk space, memory, disk cache space, and memory cache lines.

[3] In fact, MS-DOS 3.1 already provided a mount-like command called JOIN.

So are we doomed to the dynamic composition risks of the alternative: dynamic linking? In fact, as we have seen previously, this is not the case with Unix (ELF) style dynamic libraries, as these *also* support static compositions through the RPATH, which specifies a search path for the dynamic linker.

For instance, the Hello component in Chapter 2 contains in its executable image a direct reference to the path of Glibc. Likewise, the Subversion component contains the paths of its OpenSSL, Expat, Berkeley DB, Zlib, and Glibc dependencies. Thus, the Subversion program *can never fail to find its dependencies*, and it can never accidentally link to the wrong libraries at runtime. The latter is actually a substantial risk; it happens quite often that users (when they manually compile software) have different versions of libraries in different places (e.g., /usr/lib and /usr/local/lib) and that the library used at runtime does not match with what was detected at build time.

Of course, static composition is at odds with the desire to change compositions later on. An advantage of dynamic linking over static linking is that dynamic libraries can be *replaced* with "better" versions, e.g., to fix bugs; and all dependent applications will then automatically use the new version. However, this provides two conflicting features:

- The ability to fix dependent applications all at once.
- The ability to break dependent applications all at once.

Clearly there is some tension between these two.

In the purely functional model, destructive upgrading is impossible by definition, and so this "all at once" semantics is simply not available. To deploy a new version of some shared component, it is necessary to redeploy Nix expressions of all installed components that depend on it. This is expensive, but I feel that it is a worthwhile tradeoff against correctness. Section 7.5 will show how this kind of upgrading can be done with reasonable efficiency.

⇒ "Static compositions are good. Late static compositions are better."

A *late static composition* is when composition is delayed to allow components to be reused between compositions. Consider for instance the Mozilla Firefox component, a web browser. Firefox can be extended through several means. For instance, *plugins* are dynamically linked libraries that allow Firefox to handle additional MIME types, e.g., Macromedia Flash files through the Flash Player plugin. We can establish the composition between Firefox and its plugins when we build Firefox. So its Nix expression will look like this:

```
{stdenv, fetchurl, ..., plugins ? []}:
stdenv.mkDerivation { ...;
  inherit plugins;
}
```

and its builder must somehow embed the paths of the plugins into the Firefox executable. Thus the call

```
import .../firefox.nix { ...
  plugins = [flashplayer realplayer MPlayer java];
}
```

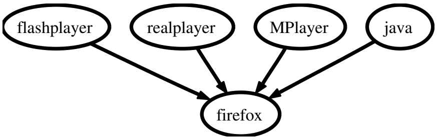


Figure 7.2.: Static Firefox composition

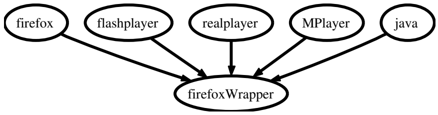


Figure 7.3.: Late static Firefox composition

yields the composition shown in Figure 7.2 (an arrow $A \rightarrow B$ denotes that A is an input of B). Now if we want to change the composition, e.g., add a new plugin, the entire Firefox component must be rebuilt, which takes an hour or so on current hardware.

Figure 7.3 shows a better way through late static binding: the plugins are not composed with the Firefox component directly, but rather the composition is delayed and established in a Firefox *wrapper component*. A wrapper component is typically just a shell script that sets some environment variables, then calls the original (wrapped) program. Thus, the firefoxWrapper component consists entirely of a shell script generated by its builder[4]:

```
#! .../sh
export MOZ_PLUGIN_PATH=realplayer-path/lib/plugins:...
exec original-firefox-path/bin/firefox "$@"
```

Since the wrapper component can be generated very quickly, changing the composition is very cheap.

Note that *dynamic binding* of the Firefox plugins requires the user to set the environment variable MOZ_PLUGIN_PATH manually prior to invoking the Firefox program. In the late static binding example, the variable is set by the wrapper script, which resides in the Nix store and is therefore part of the closure installed in the user environment.

The terms (late) static binding and dynamic binding are orthogonal to the usual notions of early binding and dynamic or late binding in component-based development [154]. They are not about binding time relative to the deployment process, for either can be done at the deployer side and on the client side. They are about whether the composition is *controlled* (i.e., described by a self-contained store derivation that results in a self-contained closure) or *uncontrolled* (i.e., performed by means outside of the scope of Nix).

⇒ “User environments are not a composition mechanism.”

User environments *can* be used as a composition mechanism, since they bring applications together in the PATH of the user, where they can find each other through dynamic composition. But that is not how they should be used in general. Before we know it, users will be given installation instructions like this: “Application foo needs application bar, so you should run nix-env -i foo bar.” This is an abuse of user environments. If there is a dependency, it should be expressed in a Nix expression that for instance builds a wrapper around foo that allows it to find bar.

There are pragmatic exceptions. We do not currently have a good handle on *crosscutting*, configurable dependencies. An example is the EDITOR environment variable, honoured by many Unix programs, that specifies the editor preferred by the user. (For instance,

[4] Nixpkgs contains code to generate wrappers automatically.

Subversion calls the program specified by this variable to allow the user to write commit messages.) Wrapper scripts are not currently a feasible solution, since if the user wishes to change his editor, he must rebuild all wrappers. Another example is *fonts* for X11 applications, which we want to configure globally; we don't want to have a set of fonts as a dependency of each X11 application.

The following is not so much a current principle as a hope for a better future:

> ⇒ "Scoped composition mechanisms are good."

A mechanism is scoped if it limits visibility of symbols. For instance, if *A* imports *B*, and *B* imports *C*, then the interface of *C* should not be visible to *A* (unless *A* explicitly imports *C*). For very many composition mechanisms this is not the case. For instance, consider the Subversion example again (Figure 2.9). Its Nix expression is a function that takes many optional dependencies:

```
{ ..., bdbSupport ? false, httpServer ? false, ...
, openssl ? null, httpd ? null, db4 ? null, expat, zlib ? null
}:
```

Suppose that we call this function with httpServer = true and bdbSupport = false. Then Subversion should not be built with Berkeley DB support. Unfortunately, there is a good chance that it will be, even though the db4 derivation attribute is empty. The reason is that Apache may be built with Berkeley DB support, and Subversion's build process will sneakily find Berkeley DB *through Apache*. For instance, Subversion's configure script asks the Apache subcomponent apr-util what compiler and linker flag it needs to properly use apr-util, and gets the following (approximate) results:

```
$ apu-config --includes
-I/nix/store/h7yw7a257m1i...-db4-4.3.28/include
$ apu-config --ldflags
-L/nix/store/h7yw7a257m1i...-db4-4.3.28/lib
```

If Subversion's configure script then dutifully adds these flags to its own list of compiler and linker flags, it will find Berkeley DB despite it not having been passed in as an explicit dependency.

This does not violate Nix's property of preventing undeclared dependencies in a *technical* sense. After all, Berkeley DB is in the input closure of the Subversion build process, since it is in the closure of the Apache output. But in a *logical* (or "moral") sense, it is an undeclared dependency.

Unscoped compositions mechanisms are not a Nix-specific problem; far from it. For instance, every conscientious C programmer who cares about header file hygiene has experienced that the C header file mechanism is unscoped: if we have a C file that includes header file *X*, and *X* in turn includes header file *Y*, then our file gets all the definitions in *Y*. This makes it virtually impossible to validate whether the C file will compile under all circumstances. Consider the case where *X* is included conditionally, but we unconditionally need the definitions from *Y*. (A result is that programmers tend to include more dependencies than perhaps required, leading to increased build times [41].) Likewise, the only way that we can validate whether the derivations involving Unix libraries produced by a Nix function will build for all function parameters, is to build all possible derivations.

A build can prove the presence of a dependency problem, not its absence, to paraphrase Dijkstra [44].

⇒ "Fine-grained components are better than large-grained components."

For the purpose of software deployment, fine-grained components are preferable to large-grained components since the latter tend to lead to unnecessary dependencies and thus to unnecessarily large closures. This was already observed in [43]. Large-grained components also reduce software reuse [42], although that is not a central concern here.

Consider for example the Python interpreter package. This package contains not just an interpreter for the Python language, but also many Python *modules*. Some of these are optional, as they can only be built if certain dependencies are present, e.g., wrapper modules for database libraries, the NCurses and Readline libraries, and so on. Thus, if we want to build a Python instance that supports all possible uses, we have to make it dependent on all these external libraries, even if most components that use Python don't actually need most or all of the optional functionality. Thus, the closure size of all components that use Python increases.

In contrast, the Perl interpreter package has far fewer optional dependencies, and Perl wrapper modules for the aforementioned modules are distributed as separate components. For instance, support for the Berkeley DB 4 database is provided in a separate package. Thus, a Perl component that does not require DB 4 support will not end up with the Berkeley DB 4 component and the Perl wrapper module in its closure.

In Nixpkgs, a strong preference is therefore given to small-grained components. For instance, Nixpkgs initially used the "XFree86" package to provide X11 client libraries for GUI applications. However, XFree86 is a large, monolithic distribution that contains not only the X11 client libraries, but also the X11 server, drivers, and utilities. Furthermore, many client libraries are not actually needed by most X clients. So it was a distinct improvement when we were able to replace XFree86 with the *X.org modular X libraries* [182], which provide each individual library as a separately deployed entity. Thus, each X application in Nixpkgs has in its closure only the libraries that it actually needs.

Of course, small-grained components also have a downside: they increase composition effort. So one certainly should not feel compelled to make components as small as possible, just to improve reuse. Rather, from the deployment perspective, the presence of external dependencies should guide decomposition of large-grained components. Also, the quality of small components is not necessarily better than that of large components [89].

7.1.2. The standard environment

Of all the components in Nixpkgs, the *standard environment* (stdenv) has a special significance as it is used as an input by almost all other components. It is in essence an aggregate of a number of components that are required by the build processes of almost all other components. It is very inconvenient to specify them separately for each derivation that uses them, and so they are combined into stdenv. The included components are:

- GCC, the GNU C Compiler, configured with C and C++ support. It has been patched to help ensure purity, as described below in Section 7.1.3

- GNU coreutils, which contains a few dozen standard Unix commands.
- GNU findutils, which provides find.
- GNU diffutils, which provides diff.
- The text manupulation tools GNU sed, GNU grep, and GNU awk.
- GNU tar, necessary for unpacking sources in TAR format.
- The compression utilities gzip and bzip2, necessary for uncompressing sources.
- GNU Make, an implementation of the make command.
- Bash, an implementation of the POSIX shell. It provides many extensions that the standard environment does depend on.
- Patch, a tool for applying deltas produced by diff.

In addition, the standard environment provides a shell script called setup that initialises the environment so that all these tools are in the builder's path. Thus, the line

```
source $stdenv/setup
```

at the top of a builder suffices to bring them into scope.

Furthermore, the inputs specified in the attribute buildInputs are brought into scope, where "scope" depends on the build tools being used. By default, the bin subdirectory of components is added to PATH, the include subdirectory is added to the C compiler's search path, and the lib subdirectory is added to the linker's search path. But this can be extended through setup hooks that components can optionally provide. For instance, if Perl is in scope, then all lib/site_perl subdirectories of components will be added to Perl's module search path. For instance, the attribute

```
buildInputs = [perl perlXMLWriter];
```

will cause the bin directory of the perl derivation's output to be added to PATH, and the lib/site_perl directory of perlXMLWriter (the Perl module XML::Writer) to be added to Perl's search path.

The generic builder The setup script also provides the *generic builder*, which is a shell function genericBuild that can automatically build many software components. An example of its use was shown in Figure 2.10. A full description of the generic builder and the ways in which it can be customised is given in the Nix manual [161], and its details are not relevant here. Briefly, the generic builder has a number of *phases*:

- The *unpack phase* unpacks the sources denoted by the attribute src (which may be a list). The current directory is switched to the top-level directory of the source tree.
- The *patch phase* applies the patches denoted by the attribute patches. This is useful to maintain Nixpkgs-specific modifications to a component's source.

- The *configure phase* brings the source into a buildable state by running the component's configure script with the argument --prefix=$out.
- The *build phase* builds the component by running make.
- The *check phase* runs make check to perform any tests (such as unit tests) included in the component.
- The *install phase* installs the component into the store path denoted by the environment variable out by running make install.

Each of these phases can be overridden by the caller to support components whose build system does not follow the general pattern. In addition, the default implementations of the phases can be customised in many ways (e.g., additional flags can be specified for the configure and make invocations).

Logging A builder can write arbitrary text to its standard output and standard error file descriptors [152]. Nix combines these streams and writes the text to its own standard error, as well as to a per-derivation log file in the directory /nix/var/nix/log/drvs.

A practical problem in Unix software development that seems to become more painful with each passing year is the log output generated by build processes. Gone are the days that an invocation of make only printed easily-understood lines such as

```
cc -c foo.c
```

and if anything else appeared, it was a relevant error message. The use of tools such as Automake and Libtool [172] and Pkgconfig [75] routinely leads to single Make actions that are thousands of characters and dozens of lines long, and in which it is almost impossible to see *what* is being compiled and *how*. Also, the size of the log output can be overwhelming: the Firefox builder produces a log of 6.4 MiB. This makes it often difficult to pinpoint the cause of a build failure.

For this reason, the generic builder produces *structured* or *nested log output*. This means that the log output becomes an *ordered tree* of log lines (i.e., the children of each node are ordered) rather than a list. For instance, the output of each phase of the generic builder is a subtree of the root. Furthermore, I patched GNU Make to produce nested output. The log messages of recursive invocations of Make form a subtree, and the output of each Make action is placed in a clearly labelled subtree.

Subtrees are delimited by ECMA-48 *control sequences* [53]. This allows nesting to be easily retro-fitted onto "legacy" tools. For instance, the byte sequence 1b 5b 33 70 (in hexadecimal) or "\e[3p" (as a C string) begins a subtree with priority 3 (higher priority numbers mean that the subtree contains less important information), while the byte sequence 1b 5b 71 or "\e[q" ends a subtree. The sequence 1b 5b is the ECMA-48 Control Sequence Introducer (CSI), which according to the standard must be followed by any number of *Parameter Bytes* in the hexadecimal range 30–3f, and a single Final Byte in the range 40–7e. An ECMA-48 conformant terminal will not display the control sequences used for log nesting since it does not know how to interpret them. Thus, when the log output is echoed to a terminal, it appears as if no control sequences are present.

```
<logfile>
  <nest><head>unpacking...</head>
    <line>...</line> ...
  </nest>
  <nest><head>building...</head>
    ...
    <nest><head>Entering directory ‘src’</head>
      <nest><head>building foo.o</head>
        <line>gcc -o foo.o -c foo.c</line>
      </nest>
    </nest>
  </nest>
</logfile>
```

Figure 7.4.: Structured log output converted to XML

The structured log output (containing escape sequences) can be processed in a variety of ways. One such way is the tool log2xml (included in Nix), which converts a log file into an XML file [20], an example of which is shown in Figure 7.4. This XML file can then be transformed into something useful using, e.g., XSLT [27]. For instance, there is an XSL stylesheet log2html that produces a nicely formatted HTML page of the log, using JavaScript and Dynamic HTML to allow parts of the tree to be collapsed or expanded. Only subtrees that contain error messages are expanded by default, allowing the developer to find relevant parts of the log easily. An example produced by the Nix build farm (Chapter 8) is shown in Figure 7.5.

Bootstrapping the standard environment The standard environment is used by the build processes of almost all other components in Nixpkgs, which raises the question: how do we build the standard environment itself? To build the components in stdenv, we need a C compiler, standard Unix tools, a shell, and so on. In other words, we need a stdenv to build stdenv. Thus, we have a classic *bootstrapping problem*.

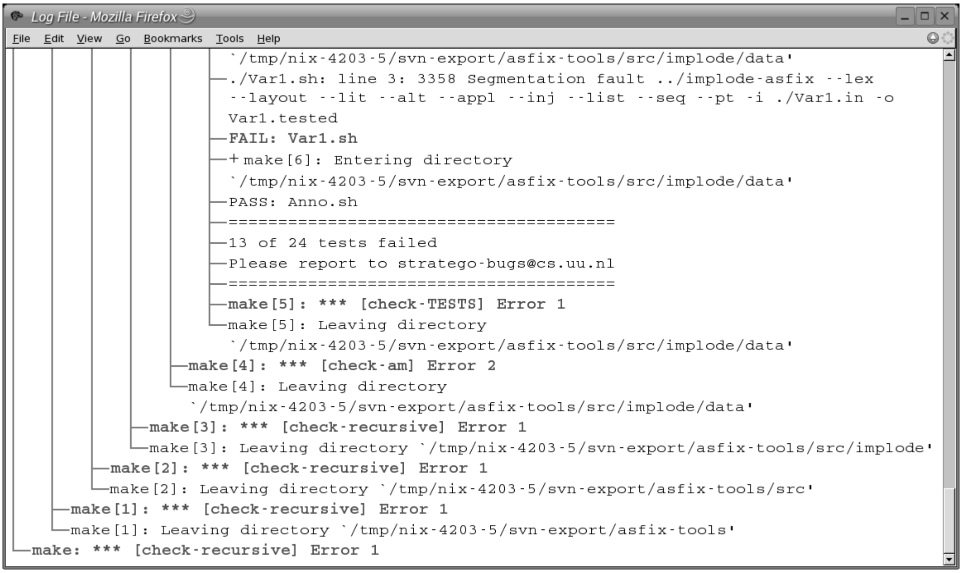
The approach initially used, and still used to build stdenv on all platforms except stdenv-linux, is to build stdenv using the “native” tools of the system, e.g., the C compiler in /usr/bin/gcc. This is of course *impure*: it depends on inputs outside of Nix’s control. Thus, a build of stdenv may not always succeed (e.g., if the C compiler is missing). However, once stdenv has been built, we can build components with no outside interference. The build process of stdenv has several stages:

- First a trivial *initial* standard environment is built (using /bin/sh for the builder) that constructs a setup script that sets the PATH to /usr/bin, etc.

- The initial stdenv is used to build GCC.

- A second stdenv is then built, identical to the first one, except that GCC is added to the PATH.

- This stdenv is used to build all other stdenv components.



```
 │    │     │     │     │    `/tmp/nix-4203-5/svn-export/asfix-tools/src/implode/data'
 │    │     │     │     ├─./Var1.sh: line 3: 3358 Segmentation fault ../implode-asfix --lex
 │    │     │     │     │  --layout --lit --alt --appl --inj --list --seq --pt -i ./Var1.in -o
 │    │     │     │     │  Var1.tested
 │    │     │     │     ├─FAIL: Var1.sh
 │    │     │     │     ├─+ make[6]: Entering directory
 │    │     │     │     │  `/tmp/nix-4203-5/svn-export/asfix-tools/src/implode/data'
 │    │     │     │     ├─PASS: Anno.sh
 │    │     │     │     ├─======================================
 │    │     │     │     ├─13 of 24 tests failed
 │    │     │     │     ├─Please report to stratego-bugs@cs.uu.nl
 │    │     │     │     ├─======================================
 │    │     │     │     ├─make[5]: *** [check-TESTS] Error 1
 │    │     │     │     └─make[5]: Leaving directory
 │    │     │     │        `/tmp/nix-4203-5/svn-export/asfix-tools/src/implode/data'
 │    │     │     ├─make[4]: *** [check-am] Error 2
 │    │     │     └─make[4]: Leaving directory
 │    │     │        `/tmp/nix-4203-5/svn-export/asfix-tools/src/implode/data'
 │    │     ├─make[3]: *** [check-recursive] Error 1
 │    │     └─make[3]: Leaving directory `/tmp/nix-4203-5/svn-export/asfix-tools/src/implode'
 │    ├─make[2]: *** [check-recursive] Error 1
 │    └─make[2]: Leaving directory `/tmp/nix-4203-5/svn-export/asfix-tools/src'
 ├─make[1]: *** [check-recursive] Error 1
 └─make[1]: Leaving directory `/tmp/nix-4203-5/svn-export/asfix-tools'
└─make: *** [check-recursive] Error 1
```

Figure 7.5.: Structured log output

- From these components the final stdenv is constructed. It has a "pure" PATH.

The stdenv thus produced has a compiler and linker that produces binaries that link against the system's C library. This is still impure, but on many systems we do not have the ability to build our own copy of the C library. If we do, as on Linux, the build process has a few extra steps. We first use the trivial stdenv to build the C library. Then we build GCC such that it links dynamically against the C library, since we want all programs in the final stdenv to use our own C library instead of the system C library. A stdenv is then constructed that has environment variables set up so that the compiler and linker use our own C library.

The ideal however is a completely pure stdenv: one that requires no external components to build *and* run, and produces components that need no external components to run. Such a stdenv has been implemented on i686-linux. Bootstrap tools such as a shell, compiler, and so on, are provided as "sources", or more properly, atoms (see Section 5.3). These programs are all statically linked executables so that they do not need external components. Roughly, the builder for stdenv looks like this:

```
derivation {
  name = "stdenv-linux";
  system = "i686-linux";
  builder = ./bash; # statically linked Bash
  args = [./builder.sh]; # build script
  tar = ./tar; # statically linked GNU Tar
  bzip2 = ./bzip2; # statically linked BZip2
  tools = ./tools.tar.bz2; # everything else, statically linked
}
```

Thus, the statically linked executables for Bash et al. are part of the Nixpkgs source tree. (In reality, tools.tar.bz2 is downloaded from the Internet, like sources obtained through fetchurl, since it is rather large. Therefore, a statically linked curl download program is also included.) These executables have of course been built themselves in some way, but they have been "re-injected" into the build process as sources—things that, as far as Nix can see, have not been derived.

Removing unnecessary dependencies Since we prefer static compositions over dynamic compositions, the standard environment tries to ensure that any executables produced by it are statically composed. That means that if an ELF (Unix) executable or library [160] refers to another library, the directory of the latter must appear in the RPATH of the former. To ensure that this is the case, stdenv's linker adds the flag -rpath *path* to the linker flags for every library directory mentioned through -L flags. Thus, a linker invocation

```
$ ld ... -L/nix/store/abcd...-foo/lib -lfoo
```

is transformed into

```
$ ld ... -L/nix/store/abcd...-foo/lib -lfoo \
    -rpath /nix/store/abcd...-foo/lib
```

However, we do not know in advance whether library foo is actually used by the linker. Regardless, the path /nix/store/abcd...-foo/lib is added to the RPATH of the output. Thus, the component gets a retained but unnecessary dependency on /nix/store/abcd...-foo.

For this reason, the standard environment after the install phase applies a tool called patchelf to all ELF executables and libraries in the output. This utility has the ability to "shrink" the RPATH of arbitrary ELF executables, i.e., remove any directories from the RPATH that do not contain any referenced library. It is effective in preventing many unnecessary retained dependencies. For instance, it prevents Hello's unnecessary dependency on GCC that we saw back on page 41.

7.1.3. Ensuring purity

If we build derivations in an environment that contains software components that are not under Nix control, there is a danger of impurity since the builder might make use of inputs outside of the Nix store. This is the main threat to the validity of the Nix approach. For instance, there is no way that we can prevent a builder from calling /usr/bin/gcc. Thus, while Nix's hashing scheme ensures isolation between components in the store, it does not ensure isolation between components in the store on the one hand and component outside the store on the other hand.

However, we can use some "countermeasures" to reduce the risk of "infection" by non-Nix components. One such countermeasure—the clearing of the environment—is built into Nix. Nixpkgs implements a range of additional measures. These include:

- The GNU C Library on i686-linux does not have any default search path for libraries (such as /usr/lib).

- GCC has been patched not to search for header files in standard locations such as /usr/include.
- The linker, ld, has been patched not to search for libraries in standard locations such as /usr/lib.
- In addition, GCC and ld ignore or reject any command-line arguments that refer to a non-store location (the only exception being the builder's temporary directory). For instance, a GCC flag like -I/opt/include is ignored; a ld argument like /usr/lib/crt1.o causes a fatal error. In practice, these measures prevent configure scripts, which frequently scan for optional dependencies in a variety of "well-known" system directories, from finding dependencies outside of the store.

It should be noted that the threat of impurity described here can only occur if Nix is used in a "hosted" environment, i.e., under an existing operating system that does not use Nix for its component storage. If *all* components are deployed through Nix, that is, if we have a *pure* environment, then this danger disappears. This leads to the as-yet unrealised ideal of a "NixOS," a fully Nix-based Unix distribution (discussed in Section 11.1).

7.1.4. Supporting third-party binary components

Most of the components in Nixpkgs are open source or free software components, i.e., their builders compile them from source. However, it *is* possible to deploy third-party closed-source components as well, provided that the Nix expression for the component can automatically obtain a binary distribution from somewhere (e.g., through fetchurl). The builder for such a component is in a way "trivial": it does not compile anything; it essentially just unpacks the input binary distribution and copies it to out. Note that this is perfectly fine: Nix requires no semantics from builders other than that they produce an FSO in the path denoted by out.

Of course, many closed-source components do not come in a form that allows us to write a builder that installs them, since they have their own (frequently interactive) installers. Sometimes we can trick our way past this. Sun's Java SDK has an interactive installer that requires the user to approve a license by pressing "Y". By simply piping that character into the installer process, the installer becomes non-interactive. Unfortunately we cannot expect such tricks to be possible in general.

Even if we can unpack the component, there is the issue of the component's dependencies. It may have runtime dependencies on external programs or libraries. However, contrary to open source components, distributors of closed-source components cannot hard-code paths to dependencies into programs since they cannot make assumptions about the target environment where the component will run. Thus, paths to dependencies are usually configurable through environment variables. These dependencies can therefore be met through late static composition using wrapper scripts, as described above.

An exception on Linux is the dependency on the GNU C library, and possibly some other well-known libraries in /lib and /usr/lib. Worst of all is the path of the *ELF dynamic linker*, which is hard-coded into every ELF executable. When a dynamically linked executable is invoked on Linux, the kernel will load the dynamic linker specified by the executable. The dynamic linker is almost always /lib/ld-linux.so.2. This is an impure external dependency, and should not be used. Rather, the ld-linux.so of Nixpkgs's Glibc should be used. But there is no way to accomplish this through environment variables[5].

The utility patchelf, which we already saw above for shrinking RPATHs, can also deal with this situation. It can change the dynamic linker path embedded in an ELF executable. Technically speaking, this is done by changing the contents of the INTERP segment of the executable, which specifies the dynamic linker (or "ELF interpreter") [160]. We have successfully deployed applications such as Sun's JDK 5.0, Eclipse and Adobe Reader in this way. The purity of the resulting components has been validated in the pure environment provided by NixOS (discussed in Section 11.1).

7.1.5. Experience

The main reason for the development of Nixpkgs was to have a substantial body of real world software components that could serve as a validation for the Nix approach. A particularly important goal was to see to what extent the assumptions from Section 6.8 hold. Did these assumptions hold in practice?

Let's evaluate the assumptions one by one:

- *"The component can be built with and installed at an arbitrary prefix"*: this assumption holds for virtually all components, since in general they cannot assume that a single fixed path is acceptable to all users. A handful of components did require a fixed location, but these were easily patched. Also, some groups of components require a *shared* prefix, i.e., they want to be installed in the same location. This was true for some components in the modular X.org tree [182]. No closed-source components were encountered that have a fixed prefix.

- *"The component does not expect that its dependencies are installed at certain fixed locations, and the locations of the dependencies can instead be specified at build or runtime"*: again, since most components cannot assume that users have installed dependencies at a fixed prefix, this assumption holds almost always. An exception is fixed dependencies on /bin/sh. This path is very widely hard-coded into shell scripts due to the "hash-bang" convention [152] (i.e., specifying the script's interpreter on the first line of the script). This is a source of impurity.

  As a result, it is not clear whether a pure Nix environment can dispense with providing /bin/sh unless a substantial effort is undertaken to patch components. (Probably most of this patching can be performed automatically, e.g., after the unpack phase in the generic builder.)

- *"The component can be built automatically, i.e., with no user interaction whatsoever"*: no decent software component requires an interactive build process, as that flies in the face of good software engineering practice (e.g., preventing the use of a build farm as in Chapter 8). But as mentioned above, some closed-source components provide only interactive installers.

[5] It is however possible to invoke the dynamic linker *directly*, specifying the program as an argument, e.g., /nix/store/72by2iw5wd8i...-glibc-2.3.5/lib/ld-linux.so.2 acroread. But this changes the calling convention of the program.

- *"The build process of the component is essentially pure"*. There are two major sources of impurity. First, the component uses file system inputs that are not explicitly specified as inputs, i.e., are not in the set *inputs* in Figure 5.11. These impurities can in turn be decomposed into undeclared dependencies on FSOs *inside* the Nix store and *outside* the Nix store. The former has not been observed at all. Thus, isolation works very well. This is good news for a pure Nix-based environment. The latter *does* occur, as shown above, and cannot be prevented in general; though, as we have seen in Section 7.1.3, the threat can be mitigated.

  The second source of impurity is a reliance on non-file system information sources, such as the system time or the network. In fact, only the system time is a real possibility; the network is seldom used by builders. However, virtually all components in practice are pure in the sense of the equivalence relations in Chapter 6. That is, if there are impurities, they do not affect the operation of the component in an observable way. One exception was observed: static libraries on Mac OS X (page 163).

- *"The component does not need to be modified after it has been built, e.g., at runtime"*: this assumption holds for almost all Unix components, since in a multi-user environment a component cannot assume to have write permission to itself. Most development components do not maintain state, and Unix application components maintain state in the user's home directory. A notable exception is IBM's Eclipse development environment, which in some pre-releases required write permission to its own prefix. Unless such components are patched to write state to another location, they will not work in Nix. But such problems generally *will* be fixed, since requiring write permission is just as unacceptable to other deployment systems such as RPM.

  It is possible that we will encounter problems with system components (which are underrepresented currently) that maintain global, as opposed to user-specific, state. For instance, the password maintenance utility passwd might (hypothetically) require that the password file is stored under $out/etc/passwd.

- *"The component has no environment dependencies"*: we have encountered no problems in practice with this assumption. Again, development components tend to have no external state, and applications keep state in the user's home directory, which they automatically initialise on first use. Thus, no special actions are necessary to make these components work, other than assuring the presence of their closures in the file system.

  But there certainly are counter-examples. A web server component might require the existence of a special user account under which the component is to execute. However, such environment dependencies are usually distinct from the component as a code-level entity. For instance, the same web server component might be used for several actual servers on the same machine, running under different user accounts as specified by the servers' respective configuration files. We will deal with this type of state in Chapter 9.

- *"Retained dependencies can be found in the component through scanning"*: we have not encountered a single instance of "pointer hiding". Of course, that does not mean that it cannot happen; it is trivial to construct a component that hides references to retained dependencies. But Nixpkgs provides strong evidence that it does not occur in practice.

- *"Hash rewriting does not change the semantics of the component"* (intensional model only): the prototype implementation of the intensional model was applied to a number of large closures, including Firefox and MonoDevelop, a C#-based graphical development environment. The experiment included equivalence class collision resolution as a result of combining builds from several users. No failures were encountered in any component[6].

In summary, we can conclude the following:

- The hashing scheme presents no problems and works very well.
- Fixed dependencies on /bin/sh and a handful of other paths are a source of impurity. This impurity does not occur in a pure Nix environment, but in that case components need to be adapted to build and work at all. In impure environments, the countermeasures of Section 7.1.3 prevent most "large" dependencies, but cannot prevent direct calls to external paths. Thus vigilance is necessary; build logs for third-party components should be inspected to verify that no external tools are being used.

There is a caveat to these results: since the components were not selected randomly, but rather "on-demand" to suit user needs, there may be a selection bias. For instance, development components may be over-represented. Also, since users are generally more inclined to add small components than large components or entire systems consisting of many components (e.g., KDE), there is a bias towards smaller components. Nevertheless, large components (such as Firefox) and systems (such as a substantial part of Gnome) have been added.

Enforcing purity Is it possible to *enforce* purity? We cannot do this in general, but operating system-specific tricks, or even extensions, might be employed for this purpose in the future. For instance, impurity due to non-Nix components can be prevented using appropriate access control rights that prevent the builder from accessing them in the first place.

Impurity due to the system time can be prevented by disallowing builders from querying the current time or timestamps on files. This can be accomplished by modifying operating system calls to not return this information (e.g., always return a timestamp of 0). However, impurity may even be present *in the instruction set of the processor*. For example, the Intel Pentium has an instruction RDTSC to read the number of clock ticks elapsed since the last restart [37]. However, this instruction can be disabled by the kernel.

[6] And this is literally a historical footnote: in [52], we wrote that "patching files [by rewriting hashes] is unlikely to work in general, e.g., due to internal checksums on files being invalidated in the process." It turns out that this assessment was too pessimistic.

7.2. User environments

Section 2.3 introduced user environments as the mechanism through which end-user components (i.e., applications) are activated. They are components that are automatically generated by nix-env to activate other components. There are many ways through which a component can be activated, i.e., made available to the user. The sets of symlinks to programs in bin directories of components shown in Figure 2.11 are just one example of how user environments can be implemented. One can imagine several other *policies* as to how components can be activated:

- On Windows and similar GUI environments, components can be activated by adding them to the Start menu. Thus, a user environment would consist of a hierarchy of *.lnk files (on Windows) or *.desktop files (on KDE) that specify the location of a program, its icon, associated media types, and so on.

- Similarly, the objects on the *desktop* in many GUI environments can be synthesised.

However, even in the Unix user environments described in Section 2.3, there are a number of policy decisions. One is the question of what constituent parts of installed components are placed in the user environment, i.e., to which files in the installed components we create symlinks. There are several options.

- Everything. The user environment is a hierarchy of symlinks that mirrors the union of the directory hierarchies of the installed components. This is what the current implementation does. However, this approach creates very large user environments, often consisting of hundreds or even thousands of symlinks. The creation of such user environments can take several seconds, which is too long. The disk space consumption may also be considerable, depending on the implementation details of the underlying file system.

- Programs only, i.e., the executables in the bin subdirectory of each installed component. This is much more efficient. However, components typically also have non-code artifacts that we wish to activate, such as documentation. For instance, the man subdirectory of each component may contain Unix manual pages in Troff format. Users will expect those manual pages to be found by the man command.

- Allow the user to specify which parts of installed components should be symlinked. This requires an extension to the nix-env installation syntax, e.g.,

```
$ nix-env -i firefox --activate bin,man
```

Another issue is how to deal with *collisions*, which occur when two installed components have one or more files with identical relative paths. Such collisions are detected by the builder of the user environment component. There are several possibilities:

- The builder can print an error message and abort, thus causing the nix-env operation to fail. In this case no new generation is produced and the current generation symlink does not change.

- If the collision is caused by multiple versions of a component, the builder can give precedence to the newest version (under some ordering of versions). Also, the older version could be made available under a different name. If we have Firefox 1.0.5 *and* 1.0.6 installed concurrently in a user environment, the symlink to the former might be called bin/firefox-1.0.5, and the latter bin/firefox.

- The builder can give precedence to the component that was added most recently. (Currently, however, the builder does not have this information.)

- All conflicting files can be renamed, e.g., to bin/firefox-1.0.5 and bin/firefox-1.0.6.

These are all policy decisions regarding the computation of the user environment. The current Nix implementation does not have an easy mechanism to change the built-in policy (other than editing the user environment builder, a simple Perl script). However, it would be fairly trivial to add an option for users to override the user environment builder, thus allowing different policies.

7.3. Binary deployment

As we have seen previously, Nix (through Nix expressions) is at its core a source deployment system, but the substitute mechanism (Section 5.5.3) allows binary deployment as an optimisation of source deployment, yielding a transparent source/binary deployment model (Section 2.6). But the substitute mechanism is just that—a mechanism. It does not implement any *particular* binary deployment method. It just consists of database registrations of the fact that an FSO for a certain store path can be produced by executing some program.

This section shows a specific implementation of binary deployment using the substitute mechanism. It consists of three fairly simple tools:

- nix-push, executed on the deployer's side, which computes the closure of a given store path, packs the store paths in the closure into archives, and places them on a web server.

- nix-pull, executed on the clients, which obtains information about the archives available on a web server, and registers the program download-using-manifests as a substitute for each corresponding store path.

- download-using-manifests, called by the substitute function (Figure 5.15), which downloads and unpacks the requested FSO.

We shall now look at the various parts in detail. Suppose that we have a Nix expression foo.nix that evaluates to a set of store derivations that we want to build and put on a web server http://server/nix-cache from where the binaries can be fetched by clients. We can do this as follows:

```
$ nix-push \
    http://server/nix-cache \
    http://server/nix-cache/MANIFEST \
    $(nix-instantiate ./foo.nix)
```

Recall that nix-instantiate translates a Nix expression to store derivations, and prints out their store paths; and that the shell syntax $(...) causes those paths to be passed as arguments to nix-push. The first URL is the location to which the archives should be uploaded. The second URL is the location to which the *manifest* of the binary deployment is uploaded. The manifest (as we briefly saw in Section 2.6) contains information about every uploaded archive.

Given a call nix-push *archiveURL manifestURL paths*, the steps that nix-push performs are as follows:

- The closure of the set *paths* is computed using the command nix-store -qR --include-outputs *paths*. As we saw on page 42, this performs a combined source/binary deployment, i.e., it includes the store derivations and their inputSrcs, but also the outputs of the store derivations. Each derivation is also built.

- For each path p in the closure, the canonical serialisation $\mathsf{serialise}(\mathsf{readPath}(p))$ (Section 5.2.1) is compressed using the bzip2 compression program. This archive is uploaded to the URL $\mathit{archiveURL} + \text{"/"} + \mathsf{printHash}_{32}(\mathsf{hash}_{\mathsf{sha256}}(c)) + \text{".nar.bz2"}$, where c is the compressed serialisation (this means that archives are stored under content-addressable names). The archive is uploaded using an HTTP PUT request [57], but only if a HEAD request reveals that the archive does not already exist on the server[7]. This makes it useful to use the same *archiveURL* between nix-push operations for multiple software releases, since common subpaths will be uploaded and stored only once.

- A manifest is then created. For each path p that we have packed into a compressed archive c and uploaded to *url*, it contains an entry of the following form:

  `{`
  `StorePath:` p
  `NarURL:` *url*
  `Hash: sha256:`$\mathsf{printHash}_{32}(\mathsf{hash}_{\mathsf{sha256}}(c))$
  `NarHash: sha256:`$\mathsf{printHash}_{32}(\mathsf{hash}_{\mathsf{sha256}}(\mathsf{serialise}(\mathsf{readPath}(p))))$
  `Size:` $|c|$
  `References:` $\mathsf{references}[p]$
  `Deriver:` $\mathsf{deriver}[p]$
  `}`

  The meaning of such an entry is that if we want to produce a store path p, we can do so by downloading a compressed serialisation from *url* and unpacking it into p. The field Hash is the hash of the compressed serialisation of p, while NarHash is the hash of the *uncompressed* serialisation of p. The former allows clients to verify that the downloaded archive has not been modified, while the latter will be of use in the binary patch deployment scheme described in Section 7.5. The References and Deriver fields are omitted if their corresponding database entries are equal to ε.

[7] The actual nix-push program takes an additional URL to be used for the HEAD requests. This allows the PUT and HEAD requests to use different URLs. E.g., the first can refer to a CGI script [124] that handles uploads, and the second is the actual location from which archives can be downloaded.

- The manifest is uploaded using a PUT request to *manifestURL*.

On the client side, the availability of the pre-built binaries created by nix-push can be registered using nix-pull *manifestURL*, e.g.,

```
$ nix-pull http://server/nix-cache/MANIFEST
```

The command nix-pull downloads the manifest and stores it in the directory /nix/var/nix/-manifests. For each store path entry *p* in the manifest, a substitute is registered using the command nix-store --register-substitutes (page 119). Note that this also sets the references database entry for *p* so that the closure invariant can be maintained. The substitute program is download-using-manifests, and there are no declared command-line arguments. But recall from Figure 5.15 that the substitute program is always called with *p* as command-line argument. Since all other necessary information for the download (such as the URL) is stored in the manifests in /nix/var/nix/manifests, no further command-line arguments are necessary.

The script download-using-manifests, when it is called by the function substitute in Figure 5.15 to produce a path *p*, reads all manifests in /nix/var/nix/manifests. Note that there may be multiple manifest entries for *p*. The script picks one arbitrarily. (An alternative implementation is to try each until one succeeds.) It downloads the archive from the URL specified by the NarURL field of the selected manifest entry, uncompresses it, deserialises it to path *p*, and checks that the cryptographic hash specified in the NarHash field of the entry matches the actual hash. In Section 7.5, we will see an extension to this script that supports binary patching.

7.4. Deployment policies

Since Nix provides basic *mechanisms* to support deployment, it is fairly easy to define all sorts of specific deployment *policies*. In general, a deployment policy consists of two elements:

- A way to get Nix expressions to the clients; this defines source deployment.
- A way to get binaries to the clients as an optimisation of source deployment through substitutes; this defines binary deployment.

However, either can be omitted. Clearly, if the latter is omitted, we have pure source deployment. If the former is omitted, clients cannot install through Nix expressions (since they don't have them), but they can install store paths directly, as we have seen in Section 5.5, e.g.,

```
$ nix-env -i /nix/store/1ja1w63wbk5q...-hello-2.1.1.drv
```

or

```
$ nix-env -i /nix/store/bwacc7a5c5n3...-hello-2.1.1
```

This section describes a number of concrete policies that have been implemented.

Direct download The most primitive deployment mechanism is to have users download Nix expressions, e.g.,

```
$ wget http://example.org/nixpkgs.tar.bz2
$ tar xvfj nixpkgs.tar.bz2
$ nix-env -f ./nixpkgs/pkgs/system/all-packages.nix -i ...
```

Of course, this source deployment policy can be made into a binary deployment policy by having clients perform a nix-pull of a manifest corresponding to the set of Nix expressions.

While this policy is primitive, it is quite useful in many environments. For instance, it can be used in many typical networked environments to automatically distribute components to (say) all desktop machines. The clients could periodically execute the displayed command sequences (a *pull model*), or the commands could be executed automatically by remote login from a script executed centrally by the system administrator (a *push model*).

Distribution through a version management system A slight modification of the previous scheme is to obtain the Nix expressions from a version management repository instead of a direct download. The following obtains a working copy of a set of Nix expressions from a Subversion repository:

```
$ svn co http://example.org/nixpkgs
$ nix-env -f ./nixpkgs/pkgs/system/all-packages.nix -i ...
```

The command svn up can then be used to keep the Nix expressions up to date.

An advantage of this policy is that it makes it easy to maintain local modifications to the set of Nix expressions. The version management tool ensures that updates from the repository are merged with local modifications.

Again, nix-pull can be used to turn this into a binary deployment scheme. However, with local modifications, the deployer can no longer ensure that the expressions that the client installs have been pre-built. Of course, that is the whole point of transparent source/binary deployment: it enables binary deployment to "degrade" automatically to source deployment.

Channels Channels are a convenient way to keep a set of software components up to date. A channel is a URL u such that there is an archive containing Nix expressions at $u +$ "/nixexprs.tar.bz2" and a manifest at $u +$ "/MANIFEST". The archive file nixexprs.tar.bz2 must unpack to a single directory that must contain (at the very least) a Nix expression default.nix. The derivations produced by the evaluation of that expression are the derivations provided by the channel. The manifest provides optional binary deployment.

Channels are managed on the client side through a simple tool called nix-channel. Users can "subscribe" to a channel:

```
$ nix-channel --add http://example.org/foo-channel
```

This causes the specified URL to be added to a list of channel URLs maintained per user. Next, the client can obtain the latest Nix expressions and manifests from all subscribed channels:

```
$ nix-channel --update
```

The update operation, for each subscribed channel u, performs a nix-pull on $u +$ "/MANIFEST", and downloads and unpacks $u +$ "/nixexprs.tar.bz2". It then makes the union of the derivations in all Nix expressions the default for nix-env operations (through the command nix-env -I). So after the update, components can be installed from the channels using nix-env:

```
$ nix-env -i foo
```

Thus, a typical usage scenario is the following command sequence:

```
$ nix-channel --update
$ nix-env -u '*'
```

which updates all components in the user's profile to the latest versions available from the subscribed channels. A typical higher-level policy is to perform this sequence automatically at certain times (e.g., from a Unix cron job).

One-click installations A one-click installation is a convenient mechanism to install a specific component. For many users it is the easiest way to install a component. One-click installations are based on Nix *packages*, which are plain-text files that contain just the following bits of information about a component:

- Its symbolic name.
- The store path of the store derivation.
- The store path of the output, i.e., the output field of the store derivation.
- Its platform identifier (e.g., i686-linux) to prevent the component from being installed on inappropriate systems.
- The URL of a manifest.

An example of the full contents of a package file is the following:

```
NIXPKG1
http://nix.cs.uu.nl/dist/nix/nixpkgs-0.9pre3530/MANIFEST
firefox-1.0.6
i686-linux
/nix/store/cbqkqc039srl48yswgzx1gs5ywnkbidp-firefox-1.0.6.drv
/nix/store/66zq88a8g36kmjl6w0nm5lfi2rvjj566-firefox-1.0.6
```

Nix packages are intended to be associated in web browsers (through the MIME media type [74] application/nix-package) with the package installer program nix-install-package. Thus, by clicking on links in web pages to packages, the package is installed (hence the term "one-click installation"). An example was shown in Figure 2.14 (page 44). The installer nix-install-package may one day be a fancy graphical user interface showing installation progress, but is currently just a simple script that performs the following actions:

- It asks the user to confirm the installation.
- It performs a nix-pull on the manifest.

- It installs the store path of the derivation output *p* directly by executing nix-env -i *p*.

This suffices to perform a binary installation of the component and all its runtime dependencies. A slight variation install the store path of the store derivation, rather than the output. In that case, source deployment is also possible.

Note that this deployment method completely bypasses Nix expressions on the client side. The store path *p* is installed directly. The closure of *p* follows from the References fields in the manifest. If the Nix expression language changes in incompatible ways, the users of Nix packages are not affected. This demonstrates one of the advantages of separating the Nix build model into high-level Nix expressions and low-level store operations.

7.5. Patch deployment

As we have seen, Nix has a purely functional deployment model. This provides important advantages, such as side-by-side deployment of versions and variants, safe upgrades, the ability to roll back, and so on. However, there is a cost in that the purely functional model makes it hard to efficiently deploy *upgrades* to clients. In particular, upgrades to "fundamental" dependencies are expensive.

Consider the dependency graph of Firefox shown in Figure 1.5 (page 10). The component at the top of the graph, used by almost all other components, is the GNU C Library (Glibc). Suppose that we discover a bug in Glibc, and wish to deploy a new version. The most straightforward way to do this is to update the Nix expression in Nixpkgs that builds Glibc to use the new version, and rebuild and reinstall the top-level components (e.g., Firefox) that depend on it. That is, we deploy to the clients a new set of Nix expressions for all installed components, and the client can then simply perform

```
$ nix-env -i firefox
```

or

```
$ nix-env -u '*' --leq
```

to upgrade all installed components with versions that are higher *or equal* (e.g., simply built with newer dependencies).

But this is expensive! Recall from Figure 2.3 (page 22) that a change to an input (such as Glibc) propagates through the dependency graph. This has two effects. First, all affected components must be rebuilt. This is exactly what we want, since the change to the dependencies may of course affect the result of the builds. This is the case even if interfaces haven't changed. Consider statically linked libraries, smart cross-module inlining, changes to the compiler affecting the binary interface, and so on. By far the easiest way to produce outputs that are consistent with the Nix expressions is to build them from scratch. Thanks to transparent source/binary deployment, this rebuild needs to be done only on the distributor side; the clients just do an appropriate nix-pull.

Unfortunately, the second effect is that *all affected components must be re-deployed to the clients*, i.e., the clients must download and install each affected component. This creates a huge scalability problem, which this section attempts to address. If we want to deploy a 100-byte bug fix to Glibc, almost all components in the system must be downloaded again, since at the very least the RPATHs of dependent binaries will have changed to point at the new Glibc. Depending on network characteristics, this can take many hours even on fast connections. Note that this is much worse than the first effect (having to rebuild the components), since that can be done centralised and only needs to be done once. The re-deployment, on the other hand, must be done for each client.

This shows the price to be paid for a purely functional deployment model. By contrast, consider deployment models that use destructive upgrading, i.e., that overwrite the files of components with newer versions. For instance, in RPM [62], we just install a new Glibc RPM package that overwrites the old one.

This however prevents side-by-side deployment of variants (what if some component *needs* the old Glibc because it is incompatible with the new one?), makes rollbacks much harder (essential in server environments), and is generally bad from an SCM perspective (since it becomes much harder to identify the current configuration). Also, such destructive upgrades only work with dynamic linking and other late-binding techniques; if the component has been statically linked into other components at build time, we must identify all affected components and upgrade them as well. This was a major problem when a security bug was discovered in the ubiquitous Zlib compression library [1].

If we were to use destructive upgrading in Nix, it would violate the crucial deployment invariant that the hash of a path uniquely describes the component. (This is similar to allowing assignments in purely functional programming languages such as Haskell [135].) From a configuration management perspective, the hashes identify the configuration of the components, and destructive updates remove the ability to identify what we have on our system. Also, it destroys the component isolation property, i.e., that an upgrade to one component cannot cause the failure to another component. If an upgrade is not entirely backwards compatible, this no longer holds.

One might ask whether the relative difficulty (in terms of hardware resources, not developer or user effort) of deploying upgrades doesn't show that Nix is unsuitable for large-scale software deployment. However, Nix's advantages in supporting side-by-side variability, correct dependency, atomic rollbacks, and so on, in the face of a quasi-component model (i.e., the huge base of existing Unix packages) not designed to support those features, make it compelling to seek a solution to the upgrade deployment problem within the Nix framework.

Upgrading through wrappers One way to efficiently deploy upgrades that works in many instances of late static composition is to use *wrapper components*. These were already discussed in Section 7.1.1. The idea is to deploy a new set of Nix expressions that describe a new derivation graph that is exactly the same as the original one (thus previously installed components do not need to be rebuilt or redownloaded), except that at top-level a *wrapper* component is added. This wrapper must ensure that at runtime the upgraded component is used, rather than the old one.

Figure 7.6 shows an example for Mozilla Firefox, which has a dependency on Glibc, both directly and indirectly through GTK; this situation is shown on the left. Suppose that we want to deploy an upgraded version of Glibc. We deploy a Nix expression that evaluates to the derivations shown on the right. The wrapper component depends on Firefox (which it wraps and forwards to), and the new Glibc (glibc'). The wrapper script for Firefox calls the original Firefox like this:

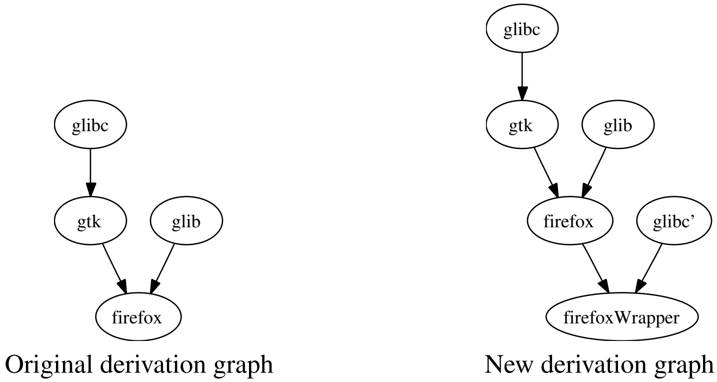


Figure 7.6.: Upgrading through wrappers

```
LD_LIBRARY_PATH=new-glibc-path:$LD_LIBRARY_PATH
exec original-firefox-path/bin/firefox "$@"
```

This causes the dynamic linker to use the new Glibc instead of the old one[8]. The old Glibc is still part of the closure, but it won't be used.

Clearly, the LD_LIBRARY_PATH is highly specific to dynamic linking on certain Unix platforms, although similar tricks are available for many other types of composition (e.g., juggling the PATH variable for composition through program calls). We also often cannot be certain that the override is used in all places. Finally, unscoped composition mechanisms make it quite hard to see what the effect of an override will be, leading to buggy wrappers.

A general solution: binary patching Thus, upgrading through wrappers is not a general solution. A general solution to the huge binary redeployment caused by a change to a fundamental component such as Glibc is by transparently deploying *binary patches* between component releases. For instance, if a bug fix to Glibc induces a switch from /nix/store/72by2iw5wd8i...-glibc-2.3.5 to /nix/store/shfb6q9yvk0l...-glibc-2.3.5-patch-1, then we compute the delta (the *binary patch*) between the contents of those paths and make the patch available to clients. We also do this for all components depending on it. Subsequently the clients can apply those patches to the old version to produce the new version. As we shall see in Section 7.5.4, patches for components affected by a change to a dependency are generally very small.

A binary patch describes a set of edit operations that transforms a *base* FSO stored at path p_{src} in the Nix store into a *target* FSO stored at path p_{dst}. Thus, if a client needs path p_{dst} *and* it has path p_{src} already installed, then it can speed up the installation of p_{dst} by downloading the patch from p_{src} to p_{dst}, copying p_{src} to p_{dst} in the Nix store, and finally applying the patch to p_{dst}.

This fits nicely into Nix's substitute mechanism (Section 7.3) used to implement transparent binary deployment. We just extend its download capabilities: if a patch is available,

[8] It's a bit more tricky in reality, since the path of the old Glibc is hard-coded into the RPATHs of the Firefox and GTK binaries. The dynamic linker has an option to override the RPATH, however.

rather than doing a full download, we download the patch instead. Manifests are extended to specify the availability of patches. For instance, the manifest entry

```
patch {
  StorePath: /nix/store/lhix54krdqkp...-firefox-1.0
  NarURL: http://server/0iihh1l7vvpa...-firefox-1.0.nar-bsdiff
  Hash: sha256:...
  NarHash: sha256:...
  Size: 357
  BasePath: /nix/store/mkmpxqr8d7f7...-firefox-1.0
  BaseHash: sha256:...
}
```

describes a 357-byte patch from the Firefox component shown in the previous section (stored at BasePath) to a new one (stored at StorePath) induced by a Glibc upgrade. If a patch is not available, or if the base component is not installed, we fall back to a full download of the new component; or even a local build if no download is available.

NarURL is the URL of the patch, Hash is the cryptographic hash of the contents of the patch, and NarHash is the cryptographic hash of the serialisation of the resulting FSO (just as in normal downloads). BaseHash is the cryptographic hash of the serialisation of the FSO at the base path to which the patch applies. In the extensional model, due to impure builders, the contents of an FSO at a given path need not always be the same. If the local contents differ from the contents on which the patch is based at the deployer's machine, then the patch cannot be applied. In the intensional model with its content-addressable Nix store, this is not an issue: if due to builder impurity a derivation has produced different output on the local machine than on the deployer's, the patch's base path simply will not exist.

7.5.1. Binary patch creation

There are many off-the-shelf algorithms and implementations to compute binary deltas between two arbitrary files. Such algorithms produce a list of *edit operations* such as copying a sequence of bytes from a position in the original file to a possibly different position in the new file, inserting a new sequence of bytes at a some position in the new file, and so on. I used the bsdiff utility [132] because it produces relatively small patches (see Section 7.6).

However, the components in the Nix store are arbitrary directory trees. How do we produce deltas between directories trees? A "simple" solution is to compute deltas between corresponding regular files (i.e., with the same relative path in the components) and distribute all deltas together. The full contents of all new files in the target should also be added, as well as a list describing file deletions, changes to symlink contents, etc. Files not listed can be assumed to be unchanged.

This method is both complicated and has the severe problem of not following renames. For instance, the Firefox component stores most of its files in a subdirectory lib/firefox-*version*. The method described above fails to patch, e.g., lib/firefox-0.9/libmozjs.so into lib/firefox-1.0/libmozjs.so since the path names do not correspond; rather, the latter file is stored in full in the patch. Hence, with this method patching is not very effective in the presence of renames.

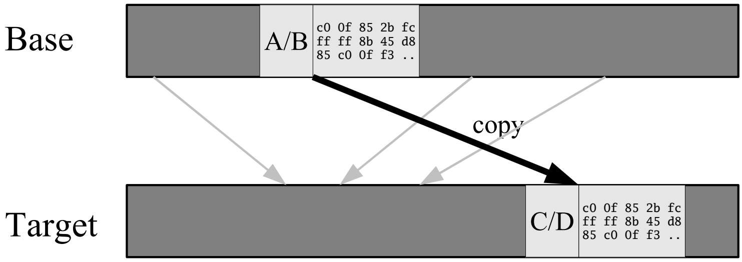


Figure 7.7.: Deltas between archives handle renames, deletions, and additions

There is however a much simpler and more effective solution: we take the patch between the *serialisations* of the components. That is, we use the function serialise (Section 5.2.1) to produce NAR archives for both the old and the new FSO, and compute the binary delta between those two files. Here we see why it is important that serialise produces a canonical representation of the FSO: it prevents a patch from failing to apply due to implementation differences in the archiving tool, or system dependencies (e.g., different orderings of files in directories). Formats such as TAR and ZIP do not have a canonical form.

Computing deltas between archives automatically takes renames, deletions, file type changes, etc. into account, since these are just simple changes within the archive files. Figure 7.7 shows an example of why this is the case: the original file A/B has been moved and renamed to C/D, which furthermore is stored in a different part of the NAR archive. However, the file contents are the same (or mildly changed). The binary delta algorithm will just emit an edit operation that changes the first file name into the second, followed by the appropriate edit operations for the file contents. It does not matter whether the position of the file in the archive has changed: contrary to delta algorithms like the standard diff tool, bsdiff can handle re-orderings of the data.

To apply a patch, a client creates an archive of the base component, applies the binary patch to it, and unpacks the resulting archive into the target path.

7.5.2. Patch chaining

It is generally infeasible to produce patches between every pair of releases of a set of components. The number of patches would be $O(n^2m)$, where n is the number of releases and m is the number of components. As an example, consider the Nix Packages collection. Pre-releases of Nixpkgs are made automatically by a build farm (Chapter 8) on every commit to its version management repository, which typically is several times a day. The pre-releases are made available in a channel. Since we do not want to impose full redownloads on developers whenever something changes, we want to make patches available.

Unfortunately, since pre-releases appear so often, we cannot feasibly produce patches between each pair of pre-releases. So as a general policy we only produce patches between immediately succeeding pre-releases. Given releases 0.7pre1899, 0.7pre1928 and 0.7pre1931, we produce patches between 0.7pre1899 and 0.7pre1928, and between 0.7pre-1928 and 0.7pre1931. This creates a problem, however: suppose that a user has Firefox

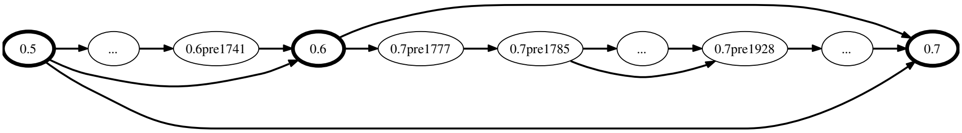


Figure 7.8.: Patch sets created between Nixpkgs releases

from 0.7pre1899 installed, and Firefox changed in both succeeding releases, then there is no patch that brings the user up-to-date.

The solution is to automatically *chain* patches, i.e., using a *series* of available patches $p_{src} \rightarrow ... \rightarrow p_n \rightarrow p_{dst}$ to produce path p_{dst}. In the example above, we have a Firefox component installed that can be used as the base path to a patch in the 0.7pre1928 release, to produce a component that can in turn serve as a base path to a patch in the 0.7pre1931 release.

However, such patch sequences can eventually become so large that their combined size approaches or exceeds the size of full downloads. In that case we can “short-circuit” the sequence by adding patches between additional releases. Figure 7.8 shows an example of patches between Nixpkgs releases thus formed. Arrows indicate the existence of a patch set between pairs of releases. Here, patch sets are produced by directly succeeding pre-releases, and between any successive stable releases. An additional “short-circuit” patch set between 0.7pre1785 and 0.7pre1928 was also made.

In the presence of patch sets between arbitrary releases, it is not directly obvious which sequence of patches or full downloads is optimal. To be fully general, the Nix substitute downloader runs a shortest path algorithm on a directed acyclic graph that, intuitively, represents components already installed, available patches between components, and available full downloads of components. Formally, the graph is defined as follows:

- The nodes are the store paths for which pre-built binaries are available on the server, either as full downloads or as patches, plus any store paths that serve as bases to patches. There is also a special start node.
- There are three types of edges:
  - *Patch edges* between store paths that represent available patches. The edge weight is the size of the patch (in bytes). In the extensional hash, edges from nodes representing valid store paths are only added if the cryptographic hash of the serialisation of the store path matches the patch’s BaseHash field, since the patch is inapplicable otherwise.
  - *Full download edges* from start to a store path for which we have a full download available. The edge weight is the size of the full download.
  - *Free edges* from start to a valid store path, representing an FSO that is already available on the system. The edge weight is 0.

We then find the shortest path between start and the path of the requested component using Dijkstra’s shortest path algorithm. This method can find any of the following:

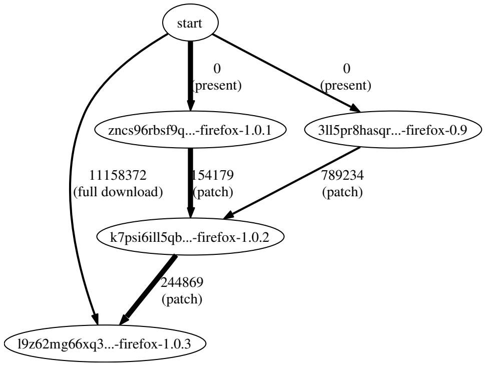

start
0
(present)
0
(present)
zncs96rbsf9q...-firefox-1.0.1
3ll5pr8hasqr...-firefox-0.9
11158372
(full download)
154179
(patch)
789234
(patch)
k7psi6ill5qb...-firefox-1.0.2
244869
(patch)
l9z62mg66xq3...-firefox-1.0.3

Figure 7.9.: Finding the optimal set of downloads

- A sequence of patches transforming an already installed component into the requested component.
- A full download of the requested component.
- A full download of some component *X* which is then transformed using a sequence of patches into the requested component. Generally, this will be longer than immediately doing a full download of the requested component, but this allows one to make *only* patches available for upgrades.

Figure 7.9 shows the graph for an instance of Firefox 1.0.2. It is available as a large full download, and through a chain of patches. There are two Firefox instances that are valid base paths for patches. (There might be other valid Firefox instances in the local store, but these do not serve as base paths for patches.) There are 3 patches: one from Firefox 0.9 to 1.0.2, one from 1.0.1 to 1.0.2, and one from 1.0.2 to 1.0.3. The patch sequence that applies the 1.0.1 to 1.0.2 and 1.0.2 to 1.0.3 patches to the valid 1.0.1 instance is the shortest path from the start node to the desired store path of 1.0.3, and is therefore selected.

Above, edge weight was defined as the size of downloads in bytes. We could take other factors into account, such as protocol/network overhead per download, the CPU resources necessary to apply patches, and so on. For instance, on a reasonably fast connection, a full download might be preferable over a long sequence of patches even if the combined byte count of those patches is less than the full download.

7.5.3. Base selection

To deploy an upgrade, we have to produce patches between "corresponding" components. This is intuitively simple: for instance, to deploy a Glibc upgrade, we have to produce patches between the old Glibc and the new one, but also between the components depending on it, e.g., between the old Firefox and the new one. However, a complication is that the dependency graphs might not be isomorphic. Components may have been removed or added, dependencies moved, component names changed (e.g., Phoenix to Firebird to Firefox to mention a real-world example), and so on. Also, even disregarding component renames, simply matching by name is insufficient because there may be multiple component instances with the same name (e.g., builds for different platforms).

When deploying a set of target store paths $\mathbb{Y}$, the *base selection problem* is to select from a set of base store paths $\mathbb{X}$ a set of patches $(X,Y) \in (\mathbb{X} \times \mathbb{Y})$ such that the probability of the Xs being present on the clients is maximised *within* certain resource constraints.

Clearly, we could produce patches between all Xs and Ys. This policy is "optimal" in the sense that the client is always able to select the absolutely shortest sequence of patches. However, it is infeasible in terms of time and space since producing a patch takes a non-negligible amount of time, and most such patches will be large since they will be between unrelated components (patching Adobe Reader into Firefox is obviously inefficient—though possible!).

Therefore, we need to select some subset of $(\mathbb{X} \times \mathbb{Y})$. The solution currently implemented is pragmatic: we use a number of properties of the components to guess whether they "match" (i.e., are conceptually the "same" component). Indeed, the selection problem appears to force us to resort to heuristics for two reasons. First, there can be arbitrary changes between releases. Second, we cannot feasibly produce all patches to select the "best" according to some objective criterion.

Possible heuristics include the following:

- *Same component name*. This is clearly one of the simplest and most effective criteria. However, there is a complication: there can be multiple components with the same name. For instance, Nixpkgs contains the GNU C Compiler gcc at several levels in the dependency graph (due to bootstrapping). Also, it contains two components called firefox—one is the "real thing", the other is a shell script wrapper around the first to enable some plugins. Finally, Nixpkgs contains the same components for multiple platforms.

- The *weighted number of uses* can be used to disambiguate between components at different bootstrapping levels such as GCC mentioned above, or disambiguate between certain variants of a component. It is defined for a component at path p as follows:

  $$w(p) = \sum_{q \in \text{users}(p)} \frac{1}{r^{d(q,p)}}$$

  where $\text{users}(p)$ is the set of components from which p is reachable in the build-time dependency graph, i.e., the components that are directly or indirectly dependent on p; where $d(q,p)$ is the unweighted distance from component q to p in the build-time dependency graph; and where $r \geq 1$ is an empirically determined value that causes less weight to be given to "distant" dependencies than to "nearby" dependencies.

  For instance, in the Nixpkgs dependency graph, there is a "bootstrap" GCC and a "final" GCC, the former being used to compile the latter, and the latter being used to compile almost all other packages. If we were to take the unweighted number of uses ($r = 1$), then the bootstrap GCC would have a slightly higher number of uses than the final GCC (since any component using the latter is indirectly dependent on the former), but the difference is too small for disambiguation—such a difference could also be caused by the addition or removal of dependent components. However, if we take, e.g., $r = 2$, then the weighted number of uses for the final GCC will be almost twice as large. This is because the bootstrap GCC is at least one step further away in the dependency graph from the majority of components, thus halving their contribution to its $w(p)$.

Thus, if the ratio between $w(p)$ and $w(q)$ is greater than some empirically determined value k, then components p and q are considered unrelated, and no patch between them is produced. A good value for k is around 2, e.g., $k = 1.9$.

- *Size* of the component. If the ratio between the sizes of two components differs more than some value l, then the components are considered unrelated. A typical value is $l = 3$; even if components differing in size by a factor of 3 *are* related, then patching is unlikely to be effective. This trivial heuristic can disambiguate between the two Firefox components mentioned above, since the wrapper script component is much smaller than the real Firefox component.

- *Platform.* In general, it is pointless to create a patch between components for different platforms (e.g., Linux and Mac OS X), since it is unlikely that a client has components for a different platform installed.

7.5.4. Experience

The binary patch deployment scheme described above has been implemented in the Nix system. To get some experimental results regarding the efficiency of binary patching, I used it to produce patches between 50 subsequent releases and pre-releases of the Nix Packages collection. Base components were selected on the basis of matching names, using the size and weighted number of uses to disambiguate between a number of components with equal names. The use of patches is automatic and completely transparent to users; an upgrade action in Nix uses (a sequence of) patches if available and applicable, and falls back to full downloads otherwise. The results below show that the patching scheme succeeds in its main goal, i.e., reducing network bandwidth consumption in the face of updates to fundamental components such as Glibc or GCC to an “acceptable” level.

I computed for each pair of subsequent releases the size of an upgrade using *full downloads* of changed components, versus the size of the required *patches* to changed components. Also, the average and median sizes of each patch for the changed components (or full download, if no patch was possible) were computed. New top-level components (e.g., applications introduced in the new release) were disregarded. Table 7.1 summarises the results for a number of selected releases, representing various types of upgrades. File sizes are in bytes unless specified otherwise. Omitted releases were typically upgrades of single leaf components such as applications. An example is the Firefox upgrade in revision 0.6pre1702.

Efficient upgrades or patches to fundamental components are the main goal of this section. For instance, release 0.7pre1980 upgraded the GNU C Compiler used to build all other components, while releases 0.7pre1820 and 0.7pre1977 provided bug fixes to the GNU C Library, also used at build time and at runtime by all other components. The patches resulting from the Glibc changes in particular are tiny: the median patch size is around 440 bytes. This is because such patches generally only need to modify the RPATH

Release	Comps. changed	Full size	Total patch size	Savings	Avg. patch size	Median patch size	Remarks
0.6pre1069	27	31.6M	162K	99.5%	6172	898	X11 libraries update
0.6pre1489	147	180M	71M	**60.5%**	495K	81K	Glibc 2.3.2 to 2.3.3, GCC 3.3.3 to 3.4.2, many other changes[9]
0.6pre1538	147	176.7M	364K	99.8%	2536	509	Standard build environment changes
0.6pre1542	1	9.3M	67K	99.3%	67K	67K	Firefox bug fix
0.6pre1672	26	38.0M	562K	98.6%	22155	6475	GTK updates
0.6pre1702	3	11.0M	190K	98.3%	63K	234K	Firefox 1.0rc1 to 1.0rc2
0.7pre1820	154	188.6M	598K	99.7%	3981	446	Glibc loadlocale bug fix
0.7pre1931	1	1164K	45K	96.1%	45K	45K	Subversion 1.1.1 to 1.1.2
0.7pre1977	153	196.3M	743K	99.6%	4977	440	Glibc UTF-8 locales patch
0.7pre1980	154	197.2M	3748K	98.1%	24924	974	GCC 3.4.2 to 3.4.3

Table 7.1.: Statistics for patch sets between selected Nixpkgs releases and their immediate predecessors

in executable and shared libraries. The average is higher (around 4K) because a handful of applications and libraries statically link against Glibc components. Still, the *total* size of the patches for all components is only 598K and 743K, respectively—a fairly trivial size even on slow modem connections.

On the other hand, release 0.6pre1489 is not small at all—the patch savings are only 60.5%. However, this release contained many significant changes. In particular, there was a major upgrade to GCC, with important changes to the generated code in all components. In general, compilers should not be switched lightly. (If *individual* components need an upgraded version, e.g., to fix a code generation bug, that is no problem: Nix expressions, being written in a functional language, can easily express that different components must be built with different compilers.) Minor compiler upgrades need not be a problem; release 0.7pre1980, which featured a minor upgrade to GCC, has a 98.1% patch effectiveness.

Patch generation is a relatively slow process. For example, the generation of the patch set for release 0.7pre1820 took 49 minutes on a 3.2 GHz Pentium 4 machine with 1 GiB of RAM running Linux 2.4.26. The bsdiff program also needs a large amount of memory; its documentation recommends a working set of at least 8 times the base file. For a large component such as Glibc, which takes 46M of disk space, this works out to 368M of RAM. In fact, we cannot currently compute patches for teTeX (a TeX distribution), regardless of how much physical memory or swap space is available, because the *address space* of 32-bit machines is not large enough to compute patches between teTeX's NAR archives, which are around 230 MiB large. The solution is a more efficient bsdiff implementation, or simply to migrate to a 64-bit system for patch computation.

A final point not addressed previously is the disk space consumption of upgrades. A

[9] First release since 0.6pre1398.

change to a component such as Glibc will still cause every component to be duplicated on disk, even if they do no longer have to be downloaded in full. However, after an upgrade, a user can run the Nix garbage collector that safely and automatically removes unused components. Nonetheless, as an optimisation, we observe that many files in those components will be exactly the same (e.g., header files, scripts, documentation, JAR files). Therefore, I implemented a tool that "optimises" the Nix store by finding all identical regular files in the store, and replacing them with hard links [152] to a single copy. On typical Nix stores (i.e., subject to normal evolution over a period of time) this saved between 15–30% of disk space. While this is useful, it is not an order of magnitude change as is the case with the amount of bandwidth saved using patches. Section 11.1 discusses as future work the possibility of using file systems that support *delta storage* to solve this problem.

7.6. Related work

This section compares Nix's suitability for deployment to other tools. Section 1.2 already touched on some of these. Most deployment tools are based on a destructive upgrade paradigm. Thus non-interference is not guaranteed at all, rollbacks are not easily supported, and upgrading is certainly not atomic. Every system depends on the deployer to specify correct runtime dependencies, although some systems have (optional) provisions to determine build time dependencies. All systems have a fairly strict separation between source and binary deployment, if both are supported at all. Interestingly, in terms of supporting correct deployment, the tools that come nearest to the ideal are not classical deployment systems (e.g., package managers) but "developer side" SCM systems such as Vesta, as discussed below.

Traditional Unix deployment systems What follows is a list of previous, "conventional" approaches to deployment in the Unix community, in more or less chronological order. I then contrast these approaches to Nix.

The Depot [116] is a deployment scheme for heterogeneous environments based on sharing components through Sun's Network File System (NFS) [23] or a similar technology. Such sharing is trivial in homogeneous environments, as each client machine can mount and use exactly the same server file system. In a heterogeneous environment, however, it may be necessary to build components for many different platforms. The Depot stores each component in its own isolated directory hierarchy (e.g., /depot/.primary/anApp), with subdirectories for sources, platform-independent files (e.g., .../include), and platform-dependent files (e.g., .../arch.sun4-os4). On the client machines, a platform-specific view on these components is synthesised. For instance, the component may be made available under /depot/anApp, with the platform-specific directory (e.g., /depot/.primary/anApp/-arch.sun4-os4) mounted on /depot/anApp/arch. Thus, each client has the same logical view of the component; e.g., /depot/anApp/arch/bin/foo always denotes the foo program for the current platform. Of course, the client view can also be implemented through symlinks or copying.

The Depot paper does not describe how components are built from sources. That is, it does not appear to have an analogue to Nix expressions. Indeed, an interesting aspect of

Depot-like approaches is that they support deployment, but are lacking in certain configuration management aspects. For instance, most Depot papers leave unspecified *how* the components in the Depot are created. Presumably, these are "manually" built by the system administrator by unpacking sources in the appropriate directory, building the component, and installing it to the appropriate target location. This means that the build is not under CM control, so there is neither reproducibility nor traceability.

Another system called Depot is described in [32]. It uses symbolic links to activated components, like Nix's user environments. This is a very common technique; e.g., GNU Stow [78] uses it as well.

Depot-Lite [143] improves on Depot. It has a primitive scanning approach for finding retained dependencies: component directories are scanned for the names of other component directories (e.g., tk-3.3). However, such names are not very unique (contrary to cryptographic hashes) and may lead to many false positives. Many aspects of the tool are rather *ad hoc*. For instance, "safe" uninstallation is performed by making the component unreadable, then making it readable again if users complain during a certain time window. Similarly, Depot-Lite allows normal users to install software, but the security model is that components are made read-only after a "stabilisation period" of 21 days.

Depot and its descendents (including Store [26], Local Disk Depot [181], SEPP [125] and GNU Stow [78]) are based on a notion of isolation between components, i.e., storing components in their own private directory trees in the file system. Many other deployment tools work *within* the traditional Unix file system organisation, i.e., installing components into directories such as /usr/bin.

The FreeBSD Ports Collection [73] is an example of such a deployment system. As described in Section 1.2, it automatically fetches and builds components from sources, recursively installing dependencies as well. Thus the build process is at least mostly reproducible. As [84] points out, a problem with the FreeBSD Ports Collection is that it embeds dependency information into Makefiles, which makes it hard to query dependencies. A more recent source deployment system is Gentoo Linux's *Portage* [77], which places particular emphasis on customisability of components through its *USE flags* mechanism.

The Red Hat Package Manager (RPM) [62] is also designed to support the conventional Unix file system layout, but it makes the chaos of that layout manageable by tracking component metadata in a database. This metadata allows an administrator to query what files belong to what packages, and so on—an invaluable capability sorely lacking in many other systems. The knowledge provided in the database also allows RPM to prevent operations that would violate correctness constraints (e.g., two components occupying the same paths in the file system, or removal of a component that is required by another component).

RPM operates at the package storage level, rather than the deployment level. That is, it has no way to fetch missing dependencies automatically. Deployment systems such as the FreeBSD Ports Collection, yum [174], and Debian's venerable apt (*Advanced Packaging Tool*) [149] maintain knowledge of repositories of remotely available packages and can fetch these on demand. Thus they are complete deployment systems.

So how do all these systems compare to Nix? All deployment tools that use the traditional Unix file system layout must necessarily be based on destructive upgrading. Thus they cannot support atomic upgrades or rollbacks, unless the underlying file system supports transactions, that is, when an arbitrarily long sequence of file system operations can be wrapped into a single atomic transaction. (No widely used file system provides this capability.) Note that it is not enough for the file system to use transactions in its own implementation, e.g., to ensure recoverability; transactions should be made available to user space processes. Thus the operating system should have APIs to make transactions available to user space (conventional file system APIs do not). An example is the Inversion file system [128]. Deployme [129] is a system that uses unspecified existing deployment tools to implement *automatic* rollback on failure of the new components (i.e., when an automatic test set fails). However, automatic rollback on failure was found to be undesirable by users.

In a destructive model, side-by-side deployment of versions and variants is only possible if the package author specifically arranges it (by making sure that files in different versions or variants have non-overlapping paths).

None of the systems described above have any fundamental means to prevent undeclared dependencies. Some use specific tricks such as looking at ELF executable headers to find library dependencies, or non-portable methods such as tracking all file system accesses performed by builds (but note that this fails to discover retained dependencies!).

Automatic garbage collection of unused components is a rare feature, since it requires a notion of "root" components. In systems such as RPM, even though the system knows the dependency graph between components, it does not know what components constitute roots (e.g., applications) and so can never delete components automatically.

Nix's transparent source/binary deployment is a fairly unique feature. Gentoo does have a primitive form of transparent source/binary deployment, but correspondence between sources and binaries is only nominal. That is, dependencies, build flags, and so on are not taken into account. It is not widely used for general deployment, only for mass deployment in specific environments.

A neat feature that Nix does not support is automatic, on-demand installation of applications. There are few deployment systems that do: Zero Install [112] is an exception. When a user starts a program, the file system access is trapped and Zero Install automatically downloads the program from a remote repository. Of course, the problem with on-demand installation is that the remote component repository might not be available when it is needed.

Some environments such as Windows and Mac OS X dispense with having a system-wide, standard deployment system altogether[10]. That is, they do not have a package management system such as RPM or Nix that maintains a global view of all components in the system along with their interdependencies. Rather, actions such as upgrading and uninstallating are the responsibility of each application. This has important consequences. First, it is impossible for administrators to do queries about components in a generic way. For instance, they cannot ask to what component a given file belongs. Second, since each application is essentially independent, there can be no dependencies. As a result, applications in these environments are all but required to be monolithic: they must contain all their dependencies, i.e., each application must be its own closure.

Integrated CM systems An interesting class of systems that might support deployment is integrated configuration management systems such as Vesta [92, 93, 94],

[10] Recent versions of Windows do contain the Windows Installer, which provides a standardised method to install packages. However, its use is not yet universal.

ClearCase [109], and Camera [114, 58, 59]. These provide more than mere source version management: they also provide build management. All three use a virtual file system (implemented through the NFS protocol) to track file system accesses, allowing detection of build-time dependencies. However, they are classical CM systems in that they support the development process, but are not intended to extend further along the software timeline, i.e., to deployment on end-user machines (though ClearCase can be used to support deployment in conjunction with IBM's Tivoli Configuration Manager [83]). That is, they are used to build software artifacts such as executables, which are then shipped to the clients through some unrelated mechanism, outside the scope of the CM system. It is a fairly recent insight that CM tools should extend to deployment [167, 168].

However, perhaps we can use these tools to support deployment already, i.e., it is just a matter of focus rather than underlying technology? This is certainly possible, but Nix has some important differences:

- Since hashes are explicit in paths, we can scan for *retained dependencies*. The systems mentioned above can only detect dependencies when they are accessed, e.g., at build time. So we cannot compute a closure in advance, and it is possible that we discover too late that the deployment is incomplete. This makes them unsuitable for general deployment—that is, more restrictive constraints on components than the assumptions in Section 6.8 must be made.

- Nix focuses on deployment. Quite a few aspects that have been carefully designed in Nix to support deployment are absent in integrated CM systems, such as a notion of user environments, various deployment models (source, binary, transparent source/binary), secure sharing of a store, and so on.

- Nix does not rely on non-portable operating system extensions, i.e., a virtual file system. Such extensions may be fine for development systems (though this is questionable), but they are a barrier to use as a deployment tool, as each user must install the extension.

- Likewise, integration of version management and build management is a barrier to use, as it potentially forces developers to switch both their version and build management tools to the integrated system, even when they are only interested in one of its aspects.

A nice property of Camera is that builds are performed in a `chroot` environment [114, Section 8.3.2]. A `chroot` is a Unix feature that allows a process's file system accesses to be confined to a part of the file system by setting its root directory to a specific directory [152]. This allows Camera to prevent undeclared dependencies.

.NET and Java Web Start Section 1.2 briefly discussed Microsoft's .NET [17, 154], which can store components in a Global Assembly Cache. Assemblies are essentially .NET's units of deployment. Assemblies have unique *strong names* that allow versions or variants to coexist. While this is a big step forward from previous deployment technologies in the Windows environment, it differs in important ways from the work described in this thesis:

- It covers only executable resources, i.e., the assemblies in the GAC. It cannot see dependencies caused by, e.g., paths specified in configuration files.
- Most components are stored in unmanaged parts of the file system, i.e., outside of the GAC.
- There is no connection between a strong name and the contents of a component. Thus it is possible for multiple components produced by a vendor to have the same strong name. In practice this is not a problem.
- It is bound to a specific component technology.

The latter criticism also applies to technologies such as Sun's Java Web Start [117], which enables web-based deployment of Java applications.

Nix on the other hand *deals with components at the most fundamental level*: their storage in the file system. Thus it is agnostic with respect to particular component technologies.

Software updaters Many recent software products come with their own built-in package management system (Vendor Product Updaters in the terminology of [102]), primarily to support automatic updating and management of extensions. Such applications include Mozilla Firefox, Eclipse, Microsoft Windows XP, and many others. These built-in updaters have a terrible problem: they interact very badly with system-wide package management systems such as Nix or RPM. A package manager assumes that it alone modifies the set of installed components. If a component modifies itself or other components, the package manager's view of the system is no longer consistent with reality.

Furthermore, package managers provide useful functionality that product-specific updaters must reimplement—poorly. Indeed, they are often implemented in an *ad hoc* fashion and ignore many important issues. Dependency management, side-by-side versioning, rollbacks, and traceability are almost never properly supported. In particular, operation in a multi-user environment—where the running component typically does not have write permission to its own installation files—is almost always neglected. For instance, Firefox's automatic updater simply will not work in such environments.

The deployment lifecycle In [168, 24] the deployment process is structured into a *lifecycle* consisting of various activities such as release, install, activation, deactivation, adapt, update, adaptation, de-install and de-release. All of these activities are explicitly present in Nix and the policies described in Section 7.4, except for the following. Adaptation (modifying the configuration of installed components) is not an explicit operation. Rather, adaptation is typically done by modifying a Nix expression and re-installing it (cf. the Firefox wrapper in Section 7.1.1). Also, components are not de-installed explicitly but rather are removed implicitly by the garbage collector. Finally, the various policies do not have a well-defined way to de-release a component (i.e., make it unavailable). Of course, we can simply remove the Nix expressions, manifests, and binaries from the distribution site. However, this removal does not automatically propagate to clients; for instance, clients might have manifests that refer to removed binaries.

Surveys of the extent to which various deployment tools support the deployment lifecycle can be found in [24, 102].

The Software Dock The Software Dock [85, 84] is a distributed, agent-based deployment framework. It consists of two types of "docks": *release docks* that represent component producers and maintain information about available releases, and *field docks* that represent clients and maintain information about the state of the local machines (e.g., already installed software). An important aspect of the Software Dock is that it supports the full lifecycle mentioned above. For instance, release docks can inform field docks about new releases. This is implemented through the Siena event service [25]. Software components, when deployed to a client, are accompanied by *agents*, which are programs that implement component-specific behaviour for lifecycle activities such as upgrades. An important part of the Software Dock is the Deployable Software Description (DSD), which is a standard language for describing software system families; i.e., it allows variability in related sets of components to be described.

The most important difference between the Software Dock and the present work is that Nix addresses low-level building and storage of components. That is, it enforces isolation, complete dependencies, and so on, by imposing a discipline on the storage of components. In the Software Dock, it is up to the various agents that perform deployment activities how to store components in the file system, whether to support multiple versions or variants side-by-side, whether to support rollbacks, and so on.

The Software Dock's main strengths lie in its high-level policies that support the entire lifecycle of large systems, while this thesis emphasises the low-level aspects of components. For instance, none of the deployment policies in Section 7.4 are truly "push" based. Channels crudely can serve as a push model by having clients poll at short intervals, but this is clearly a poor approach. Thus, closer cooperation between producers and clients, such as enabled by the Software Dock's event model, would allow more deployment policies to be supported. However, this is something that can be implemented in policies; it does not affect the basic Nix mechanisms.

Release management Van der Hoek et al. [169] list a number of requirements on deployment systems. It is instructive to determine to what extent Nix meets these requirements.

- "Dependencies should be explicit and easily recorded." Enforcing correct dependencies was one of the main drivers behind the present research. As we have seen in Section 7.1.5, Nix is successful in preventing undeclared dependencies.

- "A system should be available through multiple channels." The policy-freeness of the substitute mechanism allows Nix to support a wide variety of distribution channels, such as HTTP, installation CD-ROMs, etc.

- "The release process should involve minimal effort on the part of the developer." This is somewhat hard to quantify, but Nix has several features that reduce developer effort. Nix expressions allow a system to be rebuilt automatically when the developer changes some part of it. Transparent source/binary deployment removes the burden of having to create binary packages explicitly (other than executing a nix-push command on the source "package"). Isolation and correct dependencies reduce maintenance effort by making components more robust.

- "The scope of a release should be controllable," i.e., developers should be able to restrict releases to certain parties. Access control of releases is an orthogonal issue to the low-level Nix mechanisms. For instance, if components are deployed through a channel accessed through HTTP, then the web server can be configured with appropriate access controls.

- "A history of retrievals should be kept." Again, this is a feature that can be provided by, e.g., a web server.

- "Sufficient descriptive information should be available" to the user. This is currently missing: there is nothing in Nix expressions to describe a component, other than its name attribute. However, additional attributes can be easily added (e.g., a description attribute) and presented to the user through commands such as nix-env or on web pages for one-click installations.

- "Physical distribution should be hidden," i.e., the user should not have to be aware of the physical locations of components. Nix expressions frequently involve many physical locations (e.g., calls to fetchurl), but these are all handled automatically and do not concern the user (unless, of course, one of the locations becomes unavailable).

- "Interdependent systems should be retrievable as a group." Dependent components (e.g., servers that access each other) can always be composed into a wrapper component that starts and stops them as appropriate (examples of this are shown in Chapter 9). Assertions in Nix expressions can express consistency requirements between components.

- "Unnecessary retrievals should be avoided." This is certainly the case: once a store path is valid, it does not need to be built or downloaded again.

Deployment languages Hall et al. [86] list a number of requirements for a software deployment language, i.e., a formalism that contains the necessary information to support the deployment process. They identify the following bits of information that must be expressible in such a language:

- *Assertion constraints* express consistency requirements between components. The Nix expression language provides assertions for this purpose. However, many constraints of this type are actually unnecessary in the purely functional model, since constraints cannot be invalidated after a component has been built. For instance, the paper gives an example of a Java application that requires exactly version 1.0.2 of the Java Virtual Machine. This constraint can be true at installation time, but become invalid if the JVM is updated later on. In Nix, if the JVM is part of the closure of the application, this situation cannot happen.

- *Dependency constraints* are a more flexible kind of constraint that assertion constraints.

- *Artifacts* are the things that make up the components in the system. Nix expressions list all artifacts necessary to build components from source. However, they do *not* list the artifacts in the resulting derivation outputs. This means, for instance, that there currently is no way to query what component provides a certain command (e.g., bin/firefox) since that information is only available after components have been built—it does not follow from the Nix expression.

- *Configuration information* describes the variability provided by components. Functions in Nix expressions serve this role.

- *Activities* are the types of deployment actions that a component or system supports, such as installation, upgrading, and uninstallation. But there might be many others, such as starting or stopping a server component, so the language must be flexible. The Nix expression language has no direct provisions for supporting such activities, but we can support them as a *policy* on components. For instance, Chapter 9 shows how server components can support start and stop actions simply by providing a script called control in their output paths.

Binary patching Binary patching has a long history, going back to manual patching of binaries on mainframes in the 1960s, where it was often a more efficient method of fixing bugs than recompiling from source. Binary patching has been available in commercial patch tools such as .RTPatch, and interactive installer tools such as InstallShield. Most Unix binary package managers only support upgrades through full downloads. Microsoft recently introduced binary patching in Windows XP Service Pack 2 as a method to speed up bug fix deployment [39]. Recent releases of SuSE Linux provide *Delta RPMs*, which are binary patches between serialisations of the contents of RPM packages in cpio archive format. This is quite similar to the method of patch computation described in Section 7.5.1. An interesting point is that the cpio archive format does not have a well-defined canonical form. So it is possible that in the future patches may fail to apply due to cpio implementation differences. Also, no automatic patch chaining is provided.

A method for automatically computing and distributing binary patches between FreeBSD releases is described in [131]. It addresses the additional complication that FreeBSD systems are often built from source, and the resulting binaries can differ even if the sources are the same, for instance, due to timestamps being stored in files. In the Nix patching scheme we guard against this possibility by providing the MD5 hash of the archive to which the patch applies. If it does not apply, we fall back to a full download. In general, however, this situation does not occur because patches are obtained from the same source as the original binaries.

In Nix, the use of patches is completely hidden from users, who only observe it as a speed increase. In general, deployment methods often require users to figure out what files to download in order to install an upgrade (e.g., hotfixes in Windows). Also, if sequences of patches are required, these must be applied manually by the user, unless the distributor has consolidated them into a single patch. The creation of patches is often a manual and error-prone process, e.g., figuring out what components to redeploy as a result of a security bug like [1]. In our approach, this determination is automatic.

The bsdiff program [132] that Nix uses to generate patches is based on the *qsufsort* algorithm [108]. In our experience bsdiff outperformed methods such as ZDelta [162] and

VDelta [97, 107], but a comparison of delta algorithms is beyond the scope of this thesis. An overview of some delta algorithms is given in [97].

The problem of keeping derivates consistent with sources and dependency graph specifications occurs in all build systems, e.g., Make [56]. To ensure correctness, such systems must rebuild all dependent objects if some source changes. If a source is fundamental, then a large number of build actions may be necessary. So this problem is not unique in any way to Nix. However, the problems of build systems affect developers, not end-users, while Nix is a *deployment* system first and foremost. This is why it is important to ensure that end-users are not affected by the use of a strict update propagation semantics.

8. Continuous Integration and Release Management

This chapter shows that the Nix system can be applied to the problem of running a *build farm*, which is a tool to support *continuous integration* and *automated release management*. The former is the practice of automatically building and testing revisions of software components during the development process. The latter is the process of automatically producing *releases* of components that can be downloaded and installed by users. The two are related: if a software revision builds correctly and passes the tests, the build can be made available as a release. In fact, such a release can be made available to Nix users through the mechanisms described in Section 7.4, such as channels or one-click installations.

8.1. Motivation

Continuous integration [72] is a good software engineering practice. The idea is that each software development project should have a fully automated build system. Then we can run the build system automatically to continuously produce the most recent version of the software. Every time a developer commits a change to the project's version management system, the continuous integration system checks out the source of the project, runs its automated build process, and creates a report describing the result of the build. The latter might be presented on a web page and/or sent to the developers through e-mail.

Of course, developers are supposed to test their changes before they commit. The added advantage of a continuous integration system (apart from catching developers who *don't* test their changes) is that it allows much more in-depth testing of the component(s) being developed:

- The software may need to be built and tested on many different platforms (i.e., *portability testing*). It is infeasible for each developer to do this before every commit.
- Likewise, many projects have very large test sets (e.g., regression tests in a compiler, or stress tests in a DBMS) that can take hours or days to run to completion.
- It may also be necessary to build many different *variants* of the software. For instance, it may be necessary to verify that the component builds with various versions of a compiler.
- Developers will typically use incremental building to test their changes (since a full build may take too long), but this is often unreliable with many build management tools (such as Make). That is, the result of the incremental build might differ from a full build.

- It ensures that the software can be built from the sources under revision control. In systems such as CVS [60] and Subversion [137], developers commonly forget to place source files under revision control.

- The machines on which the continuous integration system runs ideally provides a clean, well-defined build environment. If this environment is administered through proper SCM techniques, then builds produced by the system can be reproduced. In contrast, developer work environments are typically not under any kind of SCM control.

- In large projects, developers often work on a particular component of the project, and do not build and test the composition of those components (again since this is likely to take too long). To prevent the phenomenon of "big bang integration", where components are only tested together near the end of the development process, it is important to test components together as soon as possible (hence *continuous integration*).

A continuous integration system typically sits in a loop building and releasing software components from a version management system. These are its *jobs*. For each job, it performs the following tasks:

1. It obtains the latest version of the component's source code from the version management system.

2. It runs the component's build process (which presumably includes the execution of the component's test set).

3. It presents the results of the build (such as error logs) to the developers, e.g., by producing a web page.

A continuous integration system can also produce *releases* automatically. That is, when an automatic build succeeds (and possibly when it fails!), the build result can be packaged and made available in some way to developers and users. For instance, it can produce a web page containing links to the packaged source code for the release, as well as binary distributions. The production of releases fits naturally in the actions described above: the build process of the component should produce the desired release artifacts, and the presentation of the build result will be the release web page.

The machines on which the continuous integration system runs are sometimes referred to as a *build farm* [91], since to support multi-platform projects or large sets of projects, a possibly large number of machines is required. (As a "degenerate case", a build farm can also be a single machine.)

However, existing build farm tools (such as CruiseControl, Anthill, and Tinderbox) have various limitations, discussed below.

Managing the environment One of the main problems in running a build farm is its manageability. Build farms scale poorly in terms of administrative overhead. The jobs that we want to run in the build farm require a certain environment (such as dependencies). Thus we have to make sure that the proper environment exists on every machine in the build farm. If this is done manually, the administration effort of the build farm scales linearly in the number of machines ($\Theta(n)$).

Suppose that we want to build a component that requires a certain compiler X. We then have to go to each machine and install X. If we later need a newer version of X, the process must be repeated all over again. An ever worse problem occurs if there are conflicting dependencies. A real-life example is that some components require different, *mutually exclusive* versions of the Autoconf package [64]. That is, simply installing the latest version is not an option. Of course, we can install these components in different directories and manually pass the appropriate paths to the build processes of the various components. But this is a rather tiresome and error-prone process.

Of course, Nix nails this problem: since Nix expressions describe not just how to build a single component but also how to build all its dependencies, Nix expressions are an excellent way to describe build jobs. Also, the problem of dealing with variability in the environment (such as conflicting dependencies), are automatically resolved due to Nix's hashing scheme: different dependencies end up in different paths, and Nix takes care of calling builders with the appropriate paths to dependencies. Finally, Nix's support for distributed and multi-platform builds (through the build hook mechanism of Section 5.5.2) addresses the scalability problem: as we will see below, a configuration change needs to be made only once (to the Nix expression), and Nix through the build hook will take care of rebuilding the new configuration on all platforms.

Distributed and multi-platform builds Another problem in supporting multi-platform projects is how to actually perform the build. Should each machine in the build farm independently perform jobs? If yes, then it is hard to generate a combined multi-platform release page. Therefore a centralised model is preferable, in which a single designated machine selects jobs to build and forwards build actions to particular machines in the build farm. This is far from trivial, since it entails copying build inputs and outputs to and from machines. In fact, a build action on one machine can depend on the result of a build action on another (an example of which will be shown below).

Again, the build hook mechanism enables a solution to this problem, since a single Nix expression can describe derivations for different platforms (i.e., different values for the system attribute). Derivations with different system values can also depend on each other. The build hook described below takes care of copying the appropriate closures between machines.

Building multiple configurations It should be easy to build components in many different variants, as mentioned above. A special case is building many different *compositions*, as this can reveal information useful from the perspective of continuous integration. Consider the real-life example shown in Figure 8.1 of two compilers: Stratego [176], a language based on strategic term rewriting; and Tiger, an implementation of the Tiger language [4] in Stratego. Suppose that both compilers have stable releases (e.g., Stratego 0.16 and Tiger 1.2) but are also under active development. Then there are various kinds of information that the Stratego and Tiger developers might want to obtain from the continuous integration system:

- The Tiger developers want to know whether the most recent development version of

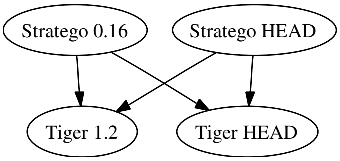


Figure 8.1.: Multiple ways of combining component revisions

Tiger (its *HEAD revision*) still builds. Here it is appropriate to build Tiger against a *stable* release of Stratego (i.e., 0.16), since when a build failure occurs the developers can then be reasonably certain that the cause is in Tiger, not Stratego.

- However, the Tiger developers may also want to know whether their current source base is synchronised with possible changes in the Stratego compiler; i.e., whether the HEAD revision of Tiger builds against the HEAD revision of Stratego.
- Likewise, the Stratego developer may want to build the *stable* release of Tiger against the HEAD revision of Stratego, so that Tiger can act as a large regression test for Stratego.

This pattern is quite common in large projects where development is split among several development groups who every so often make releases available to other groups. It is quite easy to implement building all these variants, since we can just turn the Nix expressions that build the various components into *functions* that accept their dependencies as parameters (as in Section 2.2).

Scheduling policies A build farm should support a variety of different scheduling policies. For instance, rather than building after each commit, the build can be performed on a fixed schedule, e.g., at 02:00 at night every day. This is often referred to as a "daily build" [118]. It is however generally preferable to build as quickly as possible to ensure speedy feedback to developers. This is sometimes infeasible. In a build farm used by several jobs, a particular job may take a very long time to build, starving the other jobs. In that situation it may be better to schedule the offending job to run (say) at most once a week, or to run at a fixed time of day.

More elaborate policies are also possible in a build farm that cleanly supports building of different variants of a component. Consider for instance a compiler project that must be built on many machines and has a huge regression test set. We can schedule *two* jobs for this project: a continuously running quick one that builds the compiler on only one platform, and runs only a few tests; and a weekly scheduled slow one that builds on all platforms and runs all tests.

8.2. The Nix build farm

This section gives a sketch of the build farm that we implemented using Nix. The build farm at present is not much more than a set of fairly simple scripts to run jobs, to build

```
<job id='patchelf-head'>
  <input id='job' type='svn'
    url='https://.../repos/trace/release/trunk' /> 139
  <input id='patchelfHead' type='svn'
    url='https://.../repos/trace/patchelf/trunk />
  <job-script>generic-dist/build+upload.sh</job-script> 140
  <arg>./jobs/nix/patchelf.nix</arg> 141
  <arg>patchelfHeadRelease</arg> 142
  <arg>http://nix.cs.uu.nl/dist/stratego</arg> 143
  <notify-address>somebody@example.org</notify-address> 144
</job>
```

Figure 8.2.: A build job (simplified)

the desired release products such as various kinds of binary distributions, and to produce release pages. The "heavy lifting" of managing the environment is provided by the Nix technology described in previous chapters. Thus, the process of adding a job to the build farm consists essentially of writing Nix expressions that describe components and their dependencies.

The current Nix build farm consists of a number of components. At the highest level, there is a *supervisor* script (supervisor.pl) that reads build jobs from a file (jobs.conf) and executes them in circular order. The jobs file is in XML format. Figure 8.2 shows an example of the declaration of a build job for the HEAD revision of the PatchELF component (a small component that we saw on page 179). It specifies the locations of the inputs to the build (as URLs of Subversion repositories) 139, the name of the script that performs the job 140, and its command-line arguments 141. Each input is fetched from its Subversion repository. The path of the job script is relative to the input that has ID job.

The supervisor sends e-mail notification if a job fails (or if it succeeds again after it has failed previously) to the address specified in the job 144. To prevent a flood of repeated e-mail messages for a failing build, after a job fails, the supervisor will not schedule it again until a certain time interval has passed. This interval increases on every failure using a binary exponential back-off method.

Note that the supervisor is completely Nix-agnostic: it does not care how jobs are performed. It is up to the job script to perform the build in some arbitrary way.

The job script build+upload.sh actually uses Nix to perform a build. It instantiates and builds a Nix expression specified as a command-line argument (e.g., ./jobs/nix/patchelf.nix at 141). This Nix expression is actually a function that takes as arguments the paths of the inputs declared in the XML job declaration (i.e., at 139), and the target URL of the release page (specified at 143). The job script expects the top-level derivation in the Nix expression to produce a *release page*, which is an HTML page describing the release, plus an arbitrary set of files associated with the release (such as source or binary distributions, manuals to be placed online, and so on).

Figure 8.3 shows the Nix expression ./jobs/nix/patchelf.nix that builds a release for PatchELF. Release pages are produced by the function patchelfRelease 149 that accepts a single argument input that points to the component's source code as obtained by the script build+upload.sh by performing a checkout from PatchELF's Subversion repository.

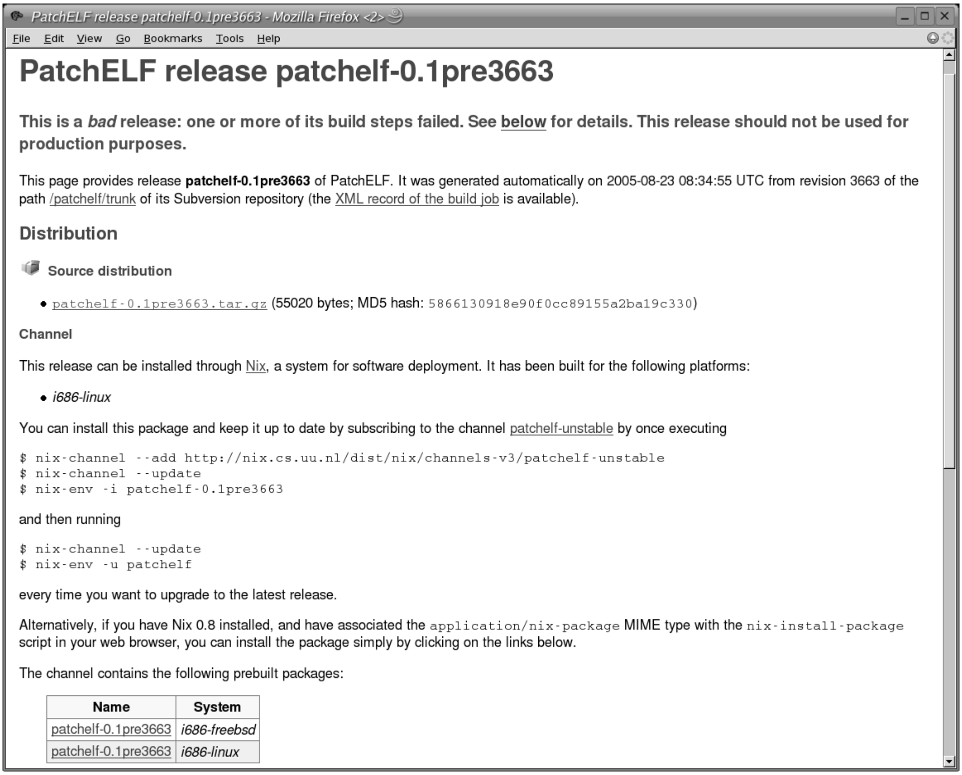
```
inputs: distBaseURL: 145

with (import ../..) inputs.nixpkgs.path; 146

rec {

  patchelfTarball = input: svnToSourceTarball "patchelf" input { 147
    inherit (pkgs) stdenv fetchsvn;
    buildInputs = [ pkgs.autoconf pkgs.automake ];
  };

  patchelfNixBuild = input: pkgs: nixBuild (patchelfTarball input) { 148
    inherit (pkgs) stdenv;
  };

  patchelfRelease = input: makeReleasePage { 149
    fullName = "PatchELF";
    contactEmail = "eelco@cs.uu.nl";
    sourceTarball = patchelfTarball input;
    nixBuilds = [
      (patchelfNixBuild input pkgsLinux)
    ];
    inherit distBaseURL;
  };

  patchelfHeadRelease = patchelfRelease (inputs.patchelfHead); 150
}
```

Figure 8.3.: Build farm Nix expression for PatchELF

A release page produced by patchelfRelease is shown in Figure 8.4.

The actual production of the release page is done by the generic release page builder function makeReleasePage (brought into scope in the with-expression at $\boxed{146}$). It accepts many arguments, only some of which are shown in the PatchELF example:

- The name of the component (e.g., "PatchELF").
- A contact e-mail address placed on the generated release pages.
- The target URL (e.g., distBaseURL) of the release page. The release page builder does not perform uploads itself (since that is impure) but it needs the target URL for self-references in the release page.
- A derivation that builds a source distribution (sourceTarball), e.g., the file patchelf-0.1pre3663.tar.gz that can be downloaded, compiled, and installed by users. The source distribution is produced from the Subversion sources (i.e., input) by the function patchelfTarball $\boxed{147}$. (A tarball is a Unix colloquialism for a source distribution.) Here too the actual work is done by an external generic function svnToSourceTarball.



PatchELF release patchelf-0.1pre3663 - Mozilla Firefox <2>
File Edit View Go Bookmarks Tools Help

PatchELF release patchelf-0.1pre3663

This is a *bad* release: one or more of its build steps failed. See below for details. This release should not be used for production purposes.

This page provides release **patchelf-0.1pre3663** of PatchELF. It was generated automatically on 2005-08-23 08:34:55 UTC from revision 3663 of the path /patchelf/trunk of its Subversion repository (the XML record of the build job is available).

Distribution

Source distribution

- `patchelf-0.1pre3663.tar.gz` (55020 bytes; MD5 hash: `5866130918e90f0cc89155a2ba19c330`)

Channel

This release can be installed through Nix, a system for software deployment. It has been built for the following platforms:

- *i686-linux*

You can install this package and keep it up to date by subscribing to the channel patchelf-unstable by once executing

```
$ nix-channel --add http://nix.cs.uu.nl/dist/nix/channels-v3/patchelf-unstable
$ nix-channel --update
$ nix-env -i patchelf-0.1pre3663
```

and then running

```
$ nix-channel --update
$ nix-env -u patchelf
```

every time you want to upgrade to the latest release.

Alternatively, if you have Nix 0.8 installed, and have associated the `application/nix-package` MIME type with the `nix-install-package` script in your web browser, you can install the package simply by clicking on the links below.

The channel contains the following prebuilt packages:

Name	System
patchelf-0.1pre3663	*i686-freebsd*
patchelf-0.1pre3663	*i686-linux*

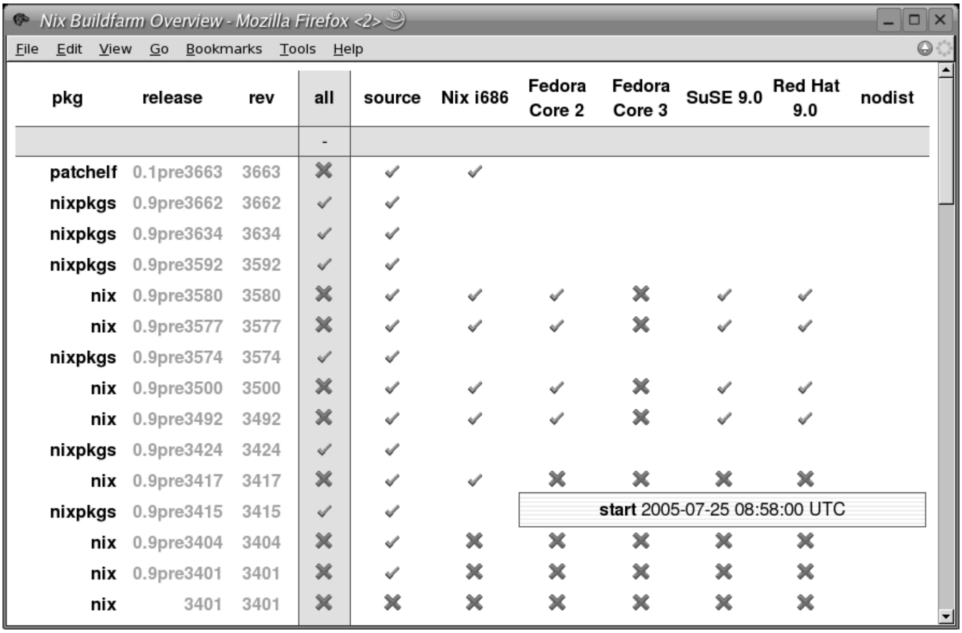
Figure 8.4.: Release page for PatchELF

- A list of derivations that perform normal builds of the component from the source distribution (nixBuilds). These are used to automatically populate a *channel* to which users can subscribe, and to generate packages for one-click installation (see Section 7.4). In this case the builds are produced by the function patchelfNixBuild **148**, which in turn uses the (poorly named) generic function nixBuild.

- Similarly, makeReleasePage accepts attributes for the component's manual, coverage analysis builds, and RPM packages.

The top-level derivation is produced by evaluation of the value patchelfHeadRelease **150**. The name of this attribute was specified in the XML job description at **142**. Building of this derivation will produce the release page and all the distributions included on the release page (in the example, a source distribution and a Nix channel distribution).

The build+upload.sh script, as its name implies, not only builds the derivation but also uploads the release page to the server. Each release is stored under its own URL, e.g., http://nix.cs.uu.nl/dist/nix/patchelf-0.1pre3663/. It also performs a nix-push to build and upload the Nix expressions in the channels provided by the release. The uploading of the release is assisted by a server-side CGI script that stores the uploaded files and, when the upload is done, updates various index pages listing the release. Figure 8.5 shows the


Nix Buildfarm Overview - Mozilla Firefox <2>
File Edit View Go Bookmarks Tools Help

pkg	release	rev	all	source	Nix i686	Fedora Core 2	Fedora Core 3	SuSE 9.0	Red Hat 9.0	nodist
			-							
patchelf	0.1pre3663	3663	✗	✓	✓					
nixpkgs	0.9pre3662	3662	✓	✓						
nixpkgs	0.9pre3634	3634	✓	✓						
nixpkgs	0.9pre3592	3592	✓	✓						
nix	0.9pre3580	3580	✗	✓	✓	✓	✗	✓	✓	
nix	0.9pre3577	3577	✗	✓	✓	✓	✗	✓	✓	
nixpkgs	0.9pre3574	3574	✓	✓						
nix	0.9pre3500	3500	✗	✓	✓	✓	✗	✓	✓	
nix	0.9pre3492	3492	✗	✓	✓	✓	✗	✓	✓	
nixpkgs	0.9pre3424	3424	✓	✓						
nix	0.9pre3417	3417	✗	✓	✓	✗	✗	✗	✗	
nixpkgs	0.9pre3415	3415	✓	✓						
nix	0.9pre3404	3404	✗	✓	✗	✗	✗	✗	✗	
nix	0.9pre3401	3401	✗	✓	✗	✗	✗	✗	✗	
nix	3401	3401	✗	✗	✗	✗	✗	✗	✗	

start 2005-07-25 08:58:00 UTC

Figure 8.5.: Release overview

automatically generated index of the most recent releases, concisely showing the extent to which each release succeeded.

Reproducing releases An important configuration management property is the ability to reproduce releases in the future. E.g., when we need to fix a bug in some old release of a component, we need to be able to reproduce the entire build environment, including compilers, libraries, and so on. So it is important that we have a record that describes exactly what inputs went into a release.

Therefore the build farm stores a file job.xml as part of every release, e.g., under http://nix.cs.uu.nl/dist/nix/patchelf-0.1pre3663/job.xml. This file is the same as the XML job description that went into the supervisor (e.g., the one in Figure 8.2), except that each input element that referred to a non-constant input such as a HEAD revision has been "absolutised". For instance, the patchelfHead input element has been changed into:

```
<input id='patchelfHead' type='svn'
  url='https://.../repos/trace/patchelf/trunk
  rev='3663' hash='b252b5740a0d...' />
```

That is, it no longer refers to the HEAD revision of the Subversion repository of PatchELF, but to a specific revision. Since this job.xml is a perfectly valid build job, we can feed it into the supervisor to reproduce the build.

Release process As can be seen in Figure 8.5, each release has a symbolic name, such as patchelf-0.1pre3663. The current build farm implements the policy that names including the string pre in the version string are “unstable” releases (i.e., only intended for developers or bleeding edge users), and are “stable” otherwise. Stable releases are intended to be kept indefinitely, while unstable releases can be deleted eventually. More importantly, stable and unstable releases appear in separate channels. For instance, the URL of the channel for stable PatchELF releases is

```
http://nix.cs.uu.nl/dist/nix/channels/patchelf-stable
```

while the channel for unstable releases is

```
http://nix.cs.uu.nl/dist/nix/channels/patchelf-unstable
```

The release name is computed by the build jobs themselves. The component name (e.g., patchelf) is generally hard-coded into the build job (e.g., at 147). The version name is usually computed by taking the version number hard-coded into the source (e.g., 0.1) and appending pre*N*, where *N* is the revision number of the Subversion repository (e.g., 3663), *if* the release is unstable. Whether the release is stable or unstable is also hard-coded in the sources.

For instance, for Autoconf-based components, the release name is usually computed by the configure script, which contains a line

```
STABLE=0
```

in the sources in the main development branch of the project (trunk in Subversion terminology [137]). Thus all releases built from the main branch will be unstable releases. To build a stable release, it suffices to change the line to

```
STABLE=1
```

and rebuild.

In practice, a more controlled process is used to build stable releases. To build a stable release, e.g., patchelf-0.1, the development branch is copied to a special *release branch*, e.g., branches/patchelf-0.1-release. In this branch, the stable flag is set. A one-time job for this branch is then added to jobs.conf. After the release succeeds, the release branch is tagged and removed. That is, branches/*X*-release is moved to tags/*X*; e.g., branches/patchelf-0.1-release is moved to tags/patchelf-0.1.

8.3. Distributed builds

Nix expressions allow multi-platform builds to be described and built in a centralised manner. In the Nix expression for the PatchELF job, we have one channel build for Linux:

```
nixBuilds = [
  (patchelfNixBuild input pkgsLinux)
];
```

Here, pkgsLinux is an attribute set consisting of the derivations in the Nix Packages collection built for i686-linux. That is, the function patchelfNixBuild accepts the dependencies to use as an argument 148. Note that this includes the standard environment stdenv, which

```
nix@mcflurry.labs.cs.uu.nl  powerpc-darwin  1
nix@losser.labs.cs.uu.nl    i686-freebsd    1
nix@itchy.labs.cs.uu.nl     i686-linux      1
nix@scratchy.labs.cs.uu.nl  i686-linux      2
```

Figure 8.6.: remote-systems.conf: definition of build farm machines

determines the platform on which to build (since the function mkDerivation in stdenv sets the system attribute for the derivation).

So if we also have sets of dependencies built for other platforms, we can trivially perform channel builds for these platforms, e.g.,:

```
nixBuilds = [
  (patchelfNixBuild input pkgsLinux)
  (patchelfNixBuild input pkgsFreeBSD)
  (patchelfNixBuild input pkgsDarwin)
];
```

where pkgsFreeBSD is Nixpkgs built for i686-freebsd, and pkgsDarwin is for powerpc-darwin (i.e., Apple's Mac OS X). Thus this Nix expression *abstracts over platforms and machines*; it is up to Nix to somehow build each derivation on a machine of the appropriate type.

Of course, the supervisor and build+upload.sh run on some particular platform, e.g., i686-linux. Nix cannot build derivations for other platforms, and will fail if it has to do so ($\boxed{76}$ in Figure 5.11). This is where build hooks come in (Section 5.5.2). The script build+upload.sh runs Nix with the environment variable NIX_BUILD_HOOK pointing at a concrete build hook script build-remote.pl.

This hook uses a table of machines to perform builds, remote-systems.conf, shown in Figure 8.6. Each entry in the table specifies a user account and remote host on which a build can be performed (e.g., nix@mcflurry.labs.cs.uu.nl), the machine's platform type (e.g., powerpc-darwin), and its *maximum load*, which is the maximum number of jobs that the hook will concurrently execute on the machine (e.g., 1). Generally, the maximum load is equal to the number of CPUs in the machine. There can be multiple machines with the same platform type, allowing more derivations to be built in parallel. The hook maintains the load on each machine in a persistent state file.

The hook uses the Rsync protocol [163] over Secure Shell (SSH) [61] to copy the input closures to the Nix store of the selected remote machine. The build is then performed by running Nix on the remote machine (also executed through SSH). Finally, the output is copied back to the local Nix store using Rsync.

8.4. Discussion and related work

This section describes some of the advantages and disadvantages of the Nix build farm relative to other continuous integration tools.

The main advantage is the use of Nix expressions to describe and build jobs. It makes the management of the build environment (i.e., dependencies) quite easy and scalable. This aspect is completely ignored by tools such as CruiseControl [158] and Tinderbox [69], which expect the environment to be managed by the machine's administrator. Anthill [165] has the notion of "dependency groups" that allows an ordering between build jobs.

Most of these systems are targeted at testing, not producing releases. Sisyphus [171] on the other hand is a continuous integration system that is explicitly intended to support deployment of upgrades to clients. It uses a destructive update model, which makes it easy to use with existing deployment tools, but bars side-by-side versioning and rollbacks.

The centralised view of the build job for a release is also a big plus. Systems such as Tinderbox have a more "anarchistic" approach: build farm machines perform jobs essentially independently, and send the results to a central machine that presents them on a web page. This is fine for continuous integration per se, but is not a good model if we want integration with release management. Since each build independently selects which revision to build, there is no guarantee that any particular revision will always be built by all machines. Thus there is no guarantee that a complete release page will ever be made.

A fundamental downside to the Nix build farm is that by building in Nix, by definition we are building in a way that differs from the "native" method for the platform. If a component builds and passes the tests on powerpc-darwin, we can conclude that the component *can* work on that platform; but we cannot conclude that it will work if a user were to download the source and build using the platform's native tools (e.g., the C compiler provided in /usr/bin). That is, while the build farm builds the component in a Nix environment, most users will use it in a non-Nix environment. This limits the level of portability testing attained by the Nix build farm.

On the other hand, this situation can be improved by simulating the native environment as closely as possible, e.g., by providing the same tool versions. Nevertheless, there is no getting around the fact that the *paths* of those tools differ; they are in the store, not in their native locations in the file system.

However, we can still build "native" binary distributions in some cases. For example, we use User-Mode Linux (UML) [45] to build RPM packages for various platforms. User-Mode Linux is a normal user space program that runs a complete Linux operating system inside a virtual machine. Thus, the builder of the derivation that builds RPMs is entirely pure: it simply runs UML to build the RPM.

Of course, we can also do "native" builds directly in Nix builders if we are willing to accept a level of impurity (e.g., by adding /usr/bin to PATH). We can even test whether the component installs properly to an impure location (such as /usr/local/my-package) if the builder has sufficient permissions. In fact, the latter need not even be impure as long as the following conditions hold:

- Subsequent derivations do not depend on output in impure locations. Thus, the builder should *remove* the impure output at the end of the build script.

- Locking should be used to prevent multiple derivations from installing into the same impure location. E.g., derivations that want to install into /usr/local/my-package can acquire an exclusive lock on a lock /usr/local/my-package.lock.

A more transient limitation of the prototype build farm is its poor scheduling. (A better supervisor is currently being implemented.) It simply runs jobs after each other in a continuous loop. It supports none of the more advanced scheduling methods discussed earlier.

It also does not run jobs in parallel, which leads to poor utilisation of the build farm. For instance, builds on Macs generally take (much) longer than on other machines. A multi-platform job can therefore cause many machines to be idle, as the job waits for the Mac build to finish. It is then useful to start a new job to put the idle machines to good use.

9. Service Deployment

This chapter shows that Nix extends naturally into the domain of *service deployment*. A software service is a set of processes running on one or more machines that cooperate to provide a useful facility to an end-user or to another software system. An example might be a bug tracking service, implemented through a web server running certain web applications and a back-end database storing persistent data. A service is generally implemented through a set of software components, static data files, dynamic state (such as databases and log files), and configuration files that tie all these together.

The deployment of such software services is very often a time-consuming and error-prone activity for developers and system administrators. In order to produce a working service, one must typically install a large set of components, put them in the right locations, write configuration files, create state directories or initialise databases, make sure that all sorts of processes are started in the right order on the right machines, and so on. These activities are quite often performed manually, or scripted in an *ad hoc* way.

A particularly troubling aspect of common service deployment practice is the lack of good configuration management. For instance, the software environment of a machine may be under the control of a package manager, and the configuration files and static data of the service under the control of a version management system. That is, these two parts of the service are both under CM control. However, the *combination* of the two *isn't*: we don't have a way to express that (say) the configuration of a web server consists of a composition of a specific instance of Apache, specific versions of configuration files and data files, and so on.

This means that we lack important features of good CM practice. There is no *identification*: we do not have a way to name what the running configuration on a system is. We may have a way to identify the configurations of the code and data bits (e.g., through package management and version management tools), but we have no handle on the composition. Likewise, there is no *derivation management*: a software service is ideally an automatically constructed derivate of code and data artifacts, meaning that we can always automatically rebuild it, e.g., to move it to another machine. However, if administrators ever manually edit a file of a running service, we lose this property.

In practice, we see that important service deployment operations are quite hard. Moving a service to another machine can be time-consuming if we have to figure out exactly what software components to install in order to establish the proper environment for the service. Having multiple instances of a service (e.g., a test and production instance) running side-by-side on the same machine is difficult since we must arrange for the instances not to overwrite each other, which often entails manually copying files and tweaking paths in configuration files. Performing a rollback of a configuration after an upgrade might be very difficult, in particular if software components were replaced.

Of course, these properties—automatic derivations, side-by-side existence, rollbacks, identifiability and reproducibility—are all provided by Nix. Up till now, we have shown

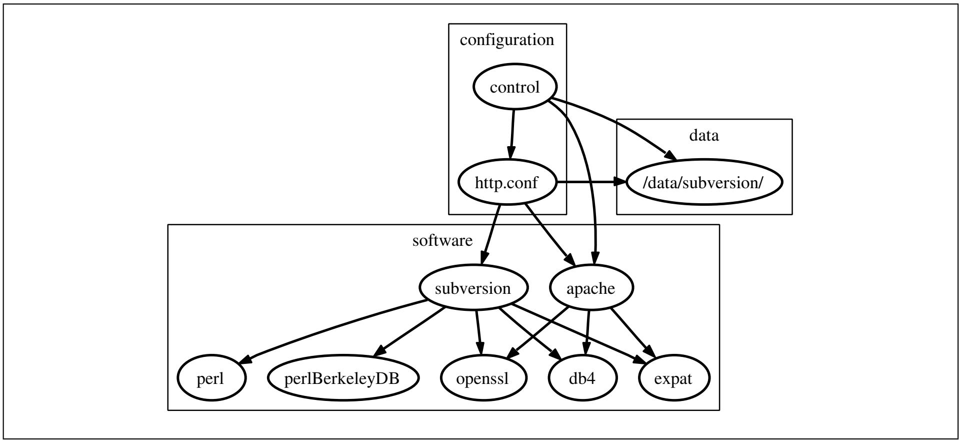


Figure 9.1.: Dependencies between code, configuration, and data components in a Subversion service

how software components can be built and deployed through Nix. This chapter shows that simply by treating services as components, we can build and deploy those as well.

9.1. Overview

9.1.1. Service components

From the perspective of a user, a service is a collection of data in some format with a coherent set of operations (use cases) on those data. Typical examples are a Subversion version management service [137] in which the data are repositories and the operations are version management actions such as committing, adding, and renaming files; a TWiki service [157] in which the data are web pages and operations are viewing, editing, and adding pages; and a JIRA issue-tracking service in which the data consists of entries in an issue data-base and the operations are issue management actions such as opening, updating, and closing issues. In these examples, the service has persistent data that is stored on the server and that can be modified by the operations of the server. However, a service can also be *stateless* and just provide some computation on data provided by the client at each request, e.g., a translation service that translates a document uploaded by the client.

From the perspective of a system administrator, a service consists of *data directories* containing the persistent state, *software components* that implement the operations on the data, and a *configuration* of those components binding the software to the data and to the local environment. Typically, the configuration includes definitions of paths to data directories and software components, and items such as the URLs and ports at which the service is provided. Furthermore, the configuration provides *meta-operations* for initialising, starting, and stopping the service. Figure 9.1 illustrates this with a diagram sketching the composition of a version management service based on Apache and Subversion components. The *control* component is a script that implements the meta-operations, while the

```
ServerRoot "/var/httpd"
ServerAdmin eelco@cs.uu.nl
ServerName svn.cs.uu.nl:8080
DocumentRoot "/var/httpd/root"
LoadModule dav_svn_module /usr/lib/modules/mod_dav_svn.so
<Location /repos>
    AuthType Basic
    AuthDBMUserFile /data/subversion/db/svn-users
    ...
    DAV svn
    SVNParentPath /data/subversion/repos
</Location>
```

Figure 9.2.: Parts of httpd.conf showing hosting of Subversion by an Apache server

file httpd.conf defines the configuration. The software components underlying a service are generally not self-contained, but rather composed from a (large) number of auxiliary components. For example, Figure 9.1 shows that Subversion and Apache depend on a number of other software components such as OpenSSL and Berkeley DB.

9.1.2. Service configuration and deployment

Setting up a service requires installing the software components and all of their dependencies, ensuring that the versions of the installed components are compatible with each other. The data directories should be initialised to the required format and possibly filled with an initial data set. Finally, a configuration file should be created to point to the software and data components. For example, Figure 9.2 shows an excerpt of an Apache httpd.conf configuration file for a Subversion service. The file uses absolute pathnames to software components such as a WebDAV module for Subversion, and data directories such as the place where repositories are stored.

Installation of software components using traditional package management systems is fraught with problems. Package managers do not enforce the completeness of dependency declarations of a component. The fact that a component works in a test environment, does not guarantee that it will work on a client site, since the test environment may provide components that are not explicitly declared as dependencies. As a consequence, component compositions are not reproducible. The installation of components in a common directory makes it hard to install multiple versions of a component or to roll back to a previous configuration if it turns out that upgrading produces a faulty configuration.

These problems are compounded in the case of service management. Typically, management of configuration files is not coupled to management of software components. Configuration files are maintained in some specific directory in the file system and changing a configuration is a destructive operation. Even if the configuration files are under version management, there is no coupling to the versions of the software components that they configure. Thus, even if it is possible to do a rollback of the configuration files to a previous time, the installation of the software components may have changed in the meantime and become out of synch with the old configuration files. Finally, having a global

```
{ stdenv, apacheHttpd, subversion }:
stdenv.mkDerivation {
  name    = "svn-service";
  builder = ./builder.sh; # Build script.
  control = ./control.in; # Control script template.
  conf    = ./httpd.conf.in; # Apache configuration template.
  inherit apacheHttpd subversion;
}
```

Figure 9.3.: services/svn.nix: Function for creating an Apache-based Subversion service

```
pkgs = import ../pkgs/system/all-packages.nix;
subversion = import ../subversion/ {
  # Get dependencies from all-packages.nix.
  inherit (pkgs) stdenv fetchurl openssl httpd ...;
};
webServer = import ../services/svn.nix {
  inherit (pkgs) stdenv apacheHttpd;
};
```

Figure 9.4.: configurations/svn.cs.uu.nl.nix: Composition of a Subversion service for a specific machine

configuration makes it hard to have multiple versions of a service running side by side, for instance a production version and a version for testing an upgrade.

9.1.3. Capturing component compositions with Nix expressions

A service can be constructed just like a software component by composing the required software, configuration, and control components. For example, Figure 9.3 defines a function for producing a Subversion service, given Apache (httpd) and Subversion components. The build script of the service creates the Apache configuration file httpd.conf and the control script bin/control from templates by filling in the paths to the appropriate software components. That is, the template httpd.conf.in contains placeholders such as

```
LoadModule dav_svn_module @subversion@/modules/mod_dav_svn.so
```

rather than absolute paths. The funcion in Figure 9.3 can be instantiated to create a concrete instance of the service. For example, Figure 9.4 defines the composition of a concrete Subversion service using an ./httpd.conf file defining the particulars of the service. Just like installing a software component composition, service composition can be reproducibly installed using the nix-env command:

```
$ nix-env -p /nix/var/nix/profiles/svn \
    -f configuration/svn.nix -i
```

I.e., the service component is added to the specified profile. The execution of the service, initialisation, activation, and deactivation, can be controlled using the control script with commands such as

```
# Recall the old server
oldServer=$(readlink -f $profiles/$serverName || true)

# Build and install the new server.
nix-env -p $profiles/$serverName -f "$nixExpr" -i

# Stop the old server.
if test -n "$oldServer"; then
    $oldServer/bin/control stop || true
fi

# Start the new server.
$profiles/$serverName/bin/control start
```

Figure 9.5.: upgrade-server: Upgrading coupled to activation and deactivation

```
$ /nix/var/nix/profiles/svn/bin/control start
```

Similarly, the stop action stops the server.

9.1.4. Maintaining the service

A service evolves over time. New versions of components become available with new functionality or security fixes; the configuration needs to be adapted to reflect changing requirements or changes in the environment; the machine that hosts the service needs to be rebooted; or the machine is retired and the service needs to be migrated to another machine. Such changes can be expressed by adapting the Nix expressions appropriately and rebuilding the service. The upgrade-server script in Figure 9.5 implements a common sequence of actions to upgrade a service: build the new service, stop the old service, and start the new one. Since changes are non-destructive, it is possible (through a similar command sequence) to roll back to a previous installation if the new one is not satisfactory. By keeping the Nix expressions defining a service under version management, the complete history of all aspects of the service is managed, and any version can be reproduced at any time. An even better approach is to have upgrade-server build directly from a version management repository, rather than from a working copy. In this way we can always trace a running service configuration back to its sources in the version management system.

9.2. Composition of services

The previous section showed a sketch of an Apache-based Subversion service deployed using Nix. The major advantage is that such a service is a closure: all code and configuration elements of the service are described in the Nix expression, and can thus be reproduced at any time. Clearly, we can deploy other services along similar lines. For instance, we might want to deploy an Apache-based TWiki service by providing Nix expressions to build the TWiki components and to compose them with Apache (which entails writing a parameterised httpd.conf that enables the TWiki CGI scripts in Apache). However, suppose we

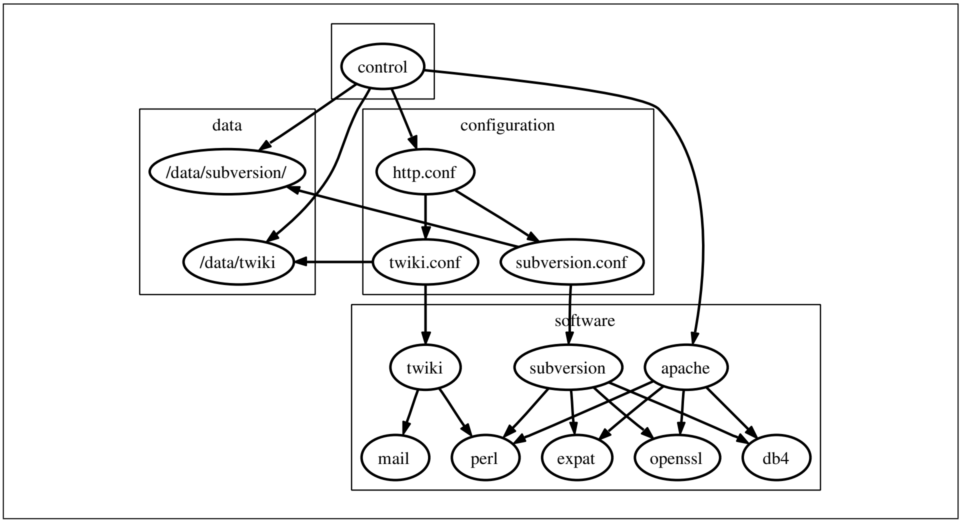


Figure 9.6.: An Apache-based composition of the Subversion and TWiki services

now want a *single* Apache server that provides both the Subversion and TWiki services. How can we easily *compose* such services?

We can solve this problem by factoring out the service-specific parts of httpd.conf into separate components called *subservices*. That is, we create configuration components that contain httpd.conf fragments such as twiki.conf and subversion.conf. The top-level httpd.conf merely contains the global server configuration such as host names, and includes the service-specific fragments. A sketch of this composition is shown in Figure 9.6.

Figure 9.7 shows a concrete elaboration. Here, the functions imported from ../subversion-service and ../jira-service build the Subversion and JIRA configuration fragments and store them under a subdirectory types/apache-httpd/conf/ in their prefixes. The function imported from ../apache-httpd then builds a top-level httpd.conf that includes those fragments. That is, there is a *contract* between the Apache service builder and the subservices that allows them to be composed.

The subServices argument of the function in ../apache-httpd specifies the list of service components that are to be composed. By modifying this list and running upgrade-server, we can easily enable or disable subservices.

9.3. Variability and crosscutting configuration choices

A major factor in the difficulty of deploying services is that many configuration choices are *crosscutting*: the realisation of such choices affects multiple configuration files or multiple points in the same file. For instance, the Apache server in Figure 9.7 requires a TCP *port number* on which the server listens for requests, so we pass that information to the top-level webServer component. However, the user management interface of the Subversion service also needs the port number to produce URLs pointing to itself. Hence, we see that

```
subversionService = import ../subversion-service {
  httpPort = 80; # see Section 9.3.
  reposDir = "/data/subversion"; ...
};

jiraService = import ../jira-service {
  twikisDir = "/data/twiki"; ...
};

webServer = import ../apache-httpd {
  inherit (pkgs) stdenv apacheHttpd;
  hostName = "svn.cs.uu.nl";
  httpPort = 80;
  subServices = [subversionService jiraService];
};
```

Figure 9.7.: Nix expression for the service in Figure 9.6

the port number is specified in two different places. In fact, Apache's configuration itself already needs the port in several places, e.g.,

```
ServerName example.org:8080
Listen 8080
<VirtualHost _default_:8080>
```

are some configuration statements that can occur concurrently in a typical httpd.conf. This leads to obvious dangers: if we update one, we can easily forget to update the other.

A related problem is that we often want to build configurations in several variants. For instance, we might want to build a server in test and production variants, with the former listening on a different port. We could make the appropriate edits to the Nix expression whenever we build either a test or production variant, or maintain two copies of the Nix expression, but both are inconvenient and unsafe.

Using the Nix expression language we have the abstraction facilities to easily support possibly crosscutting variation points. Figure 9.8 shows a refinement of the Apache composition in Figure 9.7. This Nix expression is now a *function* accepting a single boolean argument productionServer that determines whether the instance is a test or production configuration. This argument drives the value selected for the port number, which is propagated to the two components that require this information. Thus, this crosscutting parameter is specified in only one place (though implemented in two). This is a major advantage over most configuration file formats, which typically lack variables or other means of abstraction. For example, Enterprise JavaBeans deployment descriptors frequently become unwieldy due to crosscutting variation points impacting many descriptors.

Of course, due to Nix's cryptographic hashing scheme, building the server with different values for productionServer (or manually editing in the Nix expression any aspect such as the port attribute) yields different hashes and thus different paths. Therefore, multiple configurations automatically can exist side by side on the same machine.

```
{productionServer}:

let {
  port = if productionServer then 80 else 8080;

  webServer = import ./apache-httpd {
    inherit (pkgs) stdenv apacheHttpd;
    hostName = "svn.cs.uu.nl";
    httpPort = port;
    subServices = [subversionService jiraService];
  };

  subversionService = import ./subversion {
    httpPort = port;
    reposDir = "/data/subversion"; ...
  };

  jiraService = import ./jira {
    twikisDir = "/data/twiki"; ...
  };
}
```

Figure 9.8.: Dealing with crosscutting configuration choices

9.4. Distributed deployment

Complex services are often composed of several subservices *running on different machines*. For instance, consider a simple scenario of a *JIRA bug tracking system*. This service consists of two separately running subservices, possibly on different machines: a Jetty servlet container, and a PostgreSQL database server. These communicate with each other through TCP connections.

Such configurations are often labourious to deploy and maintain, since we now have two machines to administer and configure. This means that administrators have to log in to multiple machines, make sure that services are started and running, and so on. That is, without sufficient automation, the deployment effort rises linearly.

This section shows how one can implement distributed services by writing a single Nix expression that describes each subservice and the machine on which it is to run. A special *service runner component* will then take care of distributing the closure of each subservice to the appropriate machines, and remotely running their control scripts.

An interesting complication is that the various machines may be of different machine types, or may be running different operating systems. For instance, the Jetty container might be running on a Linux machine, and the PostgreSQL database on a FreeBSD machine. As we saw in Section 8.3, we can build multi-platform Nix expressions using the build hook mechanism.

Figure 9.9 shows the Nix expression for the JIRA example. We have two generic services, namely PostgreSQL and Jetty. There is one concrete subservice, namely the JIRA web application. This component is plugged into both generic services as a subservice,

```
# Build a Postgres server on FreeBSD.
postgresService = import ./postgresql {
  inherit (pkgsFreeBSD) stdenv postgresql;
  host = "losser.labs.cs.uu.nl"; # Machine to run on.
  dataDir = "/var/postgres/jira-data";

  subServices = [jiraService];
  allowedHosts = [jettyService.host]; # Access control.
};

# Build a Jetty container on Linux.
jettyService = import ./jetty {
  inherit (pkgsLinux) stdenv jetty j2re;
  host = "itchy.labs.cs.uu.nl"; # Machine to run on.

  # Include the JIRA web application at URI path.
  subServices = [ { path = "/jira"; war = jiraService; } ];
};

# Build a JIRA service.
jiraService = import ./jira/server-pkgs/jira/jira-war.nix {
  inherit (pkgsLinux) stdenv fetchurl ant postgresql_jdbc;
  databaseHost = postgresService.host;  # Database to use.
};

# Compose the two services.
serviceRunner = import ./runner {
  inherit (pkgsLinux) stdenv substituter;
  services = [postgresService jettyService];
};
```

Figure 9.9.: 2-machine distributed service

though it provides a different interface to each (i.e., implementing different contracts). To PostgreSQL, it provides an initialisation script that creates the database and tables. To Jetty, it provides a WAR that can be loaded at a certain URI path.

The PostgreSQL service is built for FreeBSD; the other components are all built for Linux. This is accomplished by passing input packages to the builders either for FreeBSD or for Linux (e.g., inherit (pkgsFreeBSD) stdenv ...), which include the standard environment and therefore specify the system on which to build.

The two generic servers are combined into a single logical service by building a *service runner component*, which is a simple wrapper component that at build time takes a list of services, and generates a control script that starts or stops each in sequence. It also takes care of distribution by deploying the closure of each service to the machine identified by its host attribute, e.g., itchy.labs.cs.uu.nl for the Jetty service. For instance, when we run the service runner's start action, it copies each service and executes its start action remotely.

An interesting point is that the Nix expression nicely deals with a crosscutting aspect of the configuration: the host names of the machines on which the services are to run. These are crosscutting because the services need to know each other's host names. In order for JIRA to access the database, the JIRA web application needs to know the host name of the database server. Conversely, the database server must allow access to the machine running the Jetty container. Here, host names are specified only once, and are propagated using expressions such as allowedHosts = [jettyService.host].

9.5. Experience

We have used the Nix-based approach described in this chapter to deploy a number of services, some in production use and some in education or development. The production services are:

- An Apache-based Subversion server (svn.cs.uu.nl) with various extensions, such as a user and repository management system. This is essentially the service described in Section 9.2.
- An Apache-based TWiki server (http://www.cs.uu.nl/wiki/Center), also using the composable infrastructure of Section 9.2. Thus, it is easy to create an Apache server providing both the Subversion and TWiki services.
- A Jetty-based JIRA bug tracking system with a PostgreSQL database back-end, as described in Section 9.4.

Also, Nix services were used in a Software Engineering course to allow teams of students working on the implementation of a Jetty-based web service (a Wiki) to easily build and run the service.

In all these cases, we have found that the greatest advantage of Nix service deployment is the ease with which configurations can be reproduced: if a developer wants to create a running instance of a service on his own machine, it is just a matter of a checkout of the Nix expressions and associated files, and a call to upgrade-server. Without Nix, setting up the required software environment for the service is much more work: installing software, tweaking configuration files to the local machine, creating state locations, and so on. The Nix services described above are essentially "plug-and-play". Also, developer machines can be quite heterogeneous. For instance, since Nix closures are self-contained, there are no dependencies on the particulars of various Linux distributions that might be used by the developers.

The ability to easily set up a test configuration is invaluable, as it makes it fairly trivial to experiment with new configurations. The ability to reliably perform a rollback, even in the face of software upgrades, is an important safety net if testing has failed to show a problem with the new configuration.

9.6. Related work

It is not a new insight that the deployment of software should be treated as part of software configuration. In particular the SWERL group at the University of Colorado at Boulder has argued for the application of software configuration management to software deployment [168], developed a framework for characterizing software deployment processes and tools [24], and implemented the experimental system Software Dock integrating software deployment processes [85, 84]. However, it appears that our work is unique in unifying the deployment of software and configuration components.

Make is sometimes used to build various configuration files. However, Make doesn't allow side-by-side deployment, as running make overwrites the previously built configurations. Thus, rollbacks are not possible. Also, the Make language is rather primitive. In particular, since the only abstraction mechanisms are global variables and patterns, it is hard to instantiate a subservice multiple times. As I discuss in Section 10.3, there are other build managers that have more functional specification languages.

Cfengine is a popular system administration tool [22]. A declarative description of sets of machines and the functionality they should provide is given, along with imperative actions that can realise a configuration, e.g., by rewriting configuration files in /etc. The principal downside of such a model is that it is *destructive*: it realises a configuration by overwriting the current one, which therefore disappears. Also, it is hard to predict what the result of a Cfengine run will be, since actions are typically specified as edit actions on system files, i.e., the initial state is not always specified. This is in contrast to the fully generational approach advocated here, i.e., Nix builders generate configurations fully indepently from any previous configurations. Finally, since actions are specified with respect to fixed configuration file locations (e.g., /etc/sendmail.mc), it is not easy for multiple configurations to coexist. In the present work, fixed paths are only used for truly mutable state such as databases and log files.

10. Build Management

This chapter shows the final application of Nix: *build management*. It should be clear by now that the low-level Nix system is essentially a build manager: it manages the automatic construction of software artifacts based on a formal description of the build actions to be performed (a Nix expression). This is exactly what tools such as Make [56] do.

However, the derivations that we have seen up till now have been *large-grained*: they build complete components such as Mozilla Firefox. Such component build actions typically consist of many smaller build steps that are performed using conventional build tools like Make. These are called by the builders of the derivations. There is no reason why these smaller build steps (such as the compilation of a single source file, or linking object files together into an executable) cannot be expressed in a Nix expression directly, obviating the need for separate build managers. Indeed, there are many advantages to using Nix as a build manager over conventional (Make-like) approaches. Nix expressions constitute a simple but powerful language to express the building of software systems, and Nix's isolation properties prevent the dependency problems that plague Make-like tools.

10.1. Low-level build management using Nix

Figure 10.1 shows an example of a Nix expression that builds the ATerm library [166] from a set of C source files. The library (in this example) has a single variation point: whether it is build as a shared library, or as a static library. Hence it is a function that takes a boolean parameter sharedLib $\boxed{151}$. The constant sources defines the source files, which reside in the same directory as the Nix expression $\boxed{153}$.

The library is produced by the attribute libATerm $\boxed{155}$. The result of this attribute is a call to the function makeLibrary, defined externally (and imported in $\boxed{152}$). This function takes a number of attributes: the name of the library, the set of object files to link, and a flag specifying whether to create a dynamic or static library. The object files are produced by compiling each source file in sources:

```
objects = map compile sources;
```

That it, the function compile is applied to each source file. It is merely a wrapper around the external function compileC to specify whether the object file should support dynamic linking $\boxed{154}$. (On Unix, dynamic linking requires that source files are compiled as "position independent code" [160].)

There are a few things worth noting here. First, we did not specify any header file dependencies for the source files in sources, even though they include a number of system and local header files (e.g., #include "aterm1.h"). However, these dependencies are found automatically by compileC. How it does this is discussed below.

```
{sharedLib ? true}: [151]

with (import ../../../lib); [152]

rec {

  sources = [ [153]
    ./afun.c ./aterm.c ./bafio.c ./byteio.c ./gc.c ./hash.c
    ./list.c ./make.c ./md5c.c ./memory.c ./tafio.c ./version.c
  ];

  compile = main: compileC {inherit main sharedLib;}; [154]

  libATerm = makeLibrary { [155]
    libraryName = "ATerm";
    objects = map compile sources;
    inherit sharedLib;
  };

}
```

Figure 10.1.: Nix expression for building the ATerm library

Second, every build step only uses sources in the Nix store. For instance, each compile action (e.g., compile ./afun.c) compiles a source that resides in the Nix store; recall that the Nix expression translation process (Section 5.4) copies each local file referenced in a Nix expression to the Nix store. Header files found by compileC are also copied to the store.

Third, since the builder for each derivation (i.e., the compile and link steps) is pure, a change to any input of a derivation will cause a rebuild. This includes changes to the main C sources, header files, C compiler flags, the C compiler and other tools, and so on.

Figure 10.2 shows another Nix expression that imports the one in Figure 10.1 to build a few small programs that link against the ATerm library; these are actually from the tests directory of the ATerm package. The function compileTest [156] compiles and links a single test program. This function is called for each test program [158]. Of course, body could also have been written as

```
body = map compileTest [./fib.c ./primes.c];
```

Note that λ-abstraction (i.e., functions) and map take the role of *pattern rules* in Make. But contrary to Make's pattern rules, they are not global: we can define any number of functions to build things in different ways in different parts of the Nix expression.

The test programs are linked against the ATerm library built by the function in Figure 10.1, which is called here to obtain a specific instance [157]. It is absolutely trivial to use multiple variants of the library in a single Nix expression. If we want two programs that link against the dynamic and static library, respectively, that's easy:

```
foo = link {...;
  libraries = [(import ../aterm {sharedLib = true;}).libATerm];
```

```
with (import ../../../lib);

let {
  compileTest = main: link { 156
    objects = [(compileC {inherit main; localIncludePath = [../aterm];})];
    libraries = [(import ../aterm {sharedLib = true;}).libATerm]; 157
  };

  body = [
    (compileTest ./primes.c) 158
    (compileTest ./fib.c)
  ];
}
```

Figure 10.2.: Nix expression for building clients of the ATerm library

```
}
bar = link {...;
  libraries = [(import ../aterm {sharedLib = false;}).libATerm];
}
```

Of course, the hashing scheme ensures that the two variants of the library end up in different paths in the store.

Finding dependencies In general, build formalisms require that the parts that constitute a system are declared. That is, we have to specify the dependencies of every build action. In Figure 10.1 we could have specified the header files referenced by each C source file explicitly, e.g.,

```
compileC {
  main = ./afun.c
  localIncludes = [./aterm2.h ./memory.h ./util.h ...];
}
```

An important advantage of Nix over build managers such as Make is that if we forget to specify a dependency, the build will fail since the compiler will not be able to locate the missing file (after all, the build takes place in a temporary directory, using only inputs in the store, and has no reference to the location from which the derivation was instantiated). Thus, if a build succeeds, we know that we have specified all the dependencies. In Make on the other hand, it is very common for dependency specifications to be incomplete. This leads to files not being rebuilt even though they should be, thus causing an inconsistency between sources and build results.

But specifying all dependencies is a lot of work, so it scales badly in terms of developer effort. So we wish to discover dependencies automatically. This has to be done in advance, since if the dependency is not present at build time, it is too late. (Systems like Vesta [92] on the other hand can discover dependencies *during* the build.) So we wish to "generate" part of the Nix expression, e.g., the value of the localIncludes attribute in the example above.

```
compileC = {main, localIncludePath ? [], cFlags ? "", sharedLib}:
  stdenv.mkDerivation {
    name = "compile-c";
    builder = ./compile-c.sh;

    localIncludes = 159
      dependencyClosure {
        scanner = file: import (findIncludes file);
        searchPath = localIncludePath;
        startSet = [main];
      };

    cFlags = [ ... ];
    inherit main;
  };

findIncludes = file: stdenv.mkDerivation { 160
  name = "find-includes";
  builder = ./find-includes.pl;
  inherit file;
};
```

Figure 10.3.: Nix expression for compiling C sources, with automatic header file determination

This is possible in the Nix expression language, since the import primitive can import Nix expressions from derivation outputs (page 81). That is, if we have a source file *X* and a derivation function that can compute a Nix list expression containing its dependencies, we can import it at translation time. The function compileC is implemented in this way, as shown in Figure 10.3. The set of include files 159 is computed by the primop dependencyClosure, which “grows” a set of files startSet using the dependencies discovered by calling the function scanner for each file. In this case, scanner is a function that returns the header files included by a file:

```
import (findIncludes file)
```

The function findIncludes 160 yields a derivation that builds in its output path a Nix expression containing the header files included by file, using a builder find-includes.pl that scans for preprocessor include statements. E.g., if file contains

```
#include "foo/bar.h"
#include "../xyzzy.h"
```

then the output of the derivation will be

```
[ "foo/bar.h" "../xyzzy.h" ]
```

Since these are relative paths, dependencyClosure will absolutise them relative to the path of the including file. Of course, these might not be all the dependencies of file, since they might in turn include other header files. But dependencyClosure will apply scanner to those as well, thus eventually obtaining all dependencies[1]. The result of dependencyClosure is a list of path references, e.g.,

```
localIncludes = [./main.c ./foo/bar.h ../xyzzy.h];
```

These files will be copied to the Nix store during the translation to store derivation. The resulting build will therefore only make use of sources in the store.

Thus parts of Nix expressions can be generated dynamically, during the translation to store derivations, by leveraging the translation and build machinery. This also allows the Nix expression language to be "extended" in a way, since if the language does not provide a way to perform certain computations (e.g., string or list operations), then a derivation and builder can be written to implement them. Odin [28] has a similar approach to extensibility.

A nice aspect of automatically finding dependencies in Nix is that the dependency scanner does not need to be entirely perfect. For instance, keeping the dependencies of TEX documents (such as those caused by \input{*file*} macros) up to date in Make is hard, since TEX does not provide a tool analogous to GCC's -MM operation to print out all dependencies of a file. Since TEX is a programming language, writing such a tool is quite difficult. However, we can also write a simple script that searches for commands such as \input{*file*} and a few others using regular expressions. Such a script is not guaranteed to find *all* dependencies, but that's okay. If it does not find all dependencies, the build will fail deterministically, and the user can add the missing dependencies to the Nix expression manually.

This thesis is in fact built using Nix. The top-level Nix expression is essentially just

```
default = runLaTeX {
  rootFile = ./doc.ltx;
}
```

where runLaTeX uses the techniques described above to guess all LATEX source files and images included by doc.ltx. (The Nix expression also builds derived artifacts such as pretty-printed code, Graphviz-generated [54] pictures, and so on.)

Relative paths Often, source files expect that their dependencies are in certain relative paths. If we compile a C source that contains the following line:

```
#include "../foo.h"
```

then we can only compile this file if the header file exists in location ../foo.h relative to the C source file. To compile modules in such languages, the builder must *reproduce* the relative source tree as it existed before translation to store derivations.

For this reason, dependencyClosure actually produces a list of dependencies that also includes their paths relative to the start file. For the example above, it returns[2]

[1] The *closure* thus obtained should not be confused with the *closures in the Nix store* used in the rest of this thesis. They are closures of source files relative to the Nix expression (e.g., in the user's working copy), prior to translation to store derivations.

[2] The encoding used here (alternating between paths and strings) is rather poorly typed—if we had a type system. Clearly, it would be better to use tuples (if the language had them) or attribute sets. However, there is currently no way to pass attribute sets to builders.

```
localIncludes = [
  ./main.c    "./main.c"
  ./foo/bar.h "./foo/bar.h"
  ../xyzzy.h  "../xyzzy.h"
];
```

This extra information may seem redundant, but recall that path constants such as ../xyzzy.h are actually absolutised internally, e.g., to /home/eelco/project/xyzzy.h. And the path is certainly lost during translation to store derivations (see the processBinding case for path constants in Figure 5.7). Thus the relative paths, encoded as strings, are actually necessary to pass the required information to the builder.

The builder of compilerC, compile-c.sh, will use this information to create in its temporary build directory a *hierarchy of symlinks* such as

```
./xyzzy.h
./dotdot/main.c
./dotdot/foo/bar.h
```

to the actual sources in the Nix store. (A number of dotdot directories equal to the maximum number of .. references in relative paths is inserted to make those relative paths resolve properly.) Compiling ./dotdot/main.c will then yield the desired result.

Building So how do we actually *build* the derivations defined in, e.g., Figure 10.2? A tool such as nix-env is inappropriate, since we do not necessarily want to “install” the build result. Hence there is a small tool nix-build (briefly mentioned in Section 5.6.1), which is just a wrapper around nix-instantiate and nix-store --realise that builds a Nix expression and places a symlink to the derivation output (named result) in the current directory. If the top-level Nix expression yields multiple derivations, the links are numbered. By default, nix-build builds the expression default.nix in the current directory. E.g., when applied to the expression in Figure 10.2, we get

```
$ nix-build
...
$ ./result-2/program
fib(32) == 3524578
```

since the symlink result-2 refers to the second derivation (the fib.c program, which computes the Fibonacci sequence).

10.2. Discussion

Low-level build management using Nix has numerous advantages over most build managers (discussed below). Of course, most of these are exactly the advantages that it also has for deployment, showing that build management and deployment are very much overlapping problems.

- It ensures that dependency specifications are complete (though most dependencies can be found automatically).

- *All inputs*, not just source files but also flags, scripts, tools, etc., are dependencies. Running nix-build is guaranteed to produce a result consistent with the sources and the Nix expression. In Make, by contrast, changes to the build actions do not trigger recompilation. This is why Make users have a habit of running make clean often, just to be on the safe side.

- Since dependencies are complete and builds produce no output other than in out, parallel and distributed builds are safe.

- There is no pollution of the working copy in which the source resides. Thus there is no need for an operation like make clean. Garbage collection takes care of freeing disk space.

- Due to locking (page 98), it is safe to run multiple Nix builds in parallel, even on the same Nix expression.

- It is safe to edit sources while a build is in progress. In contrast, with Make we can run into race conditions, e.g., if we start a big LaTeX job or C++ compile, then edit the source in Emacs and save while the build is in progress. The build will use the old sources, but subsequent Make invocations will believe that the output is up to date.

- The Nix expression language makes it very easy to specify variants. Since it is a functional language, almost any repetitiveness can be abstracted away. So the functionality provided by tools such as Automake and Libtool can be implemented through Nix expression libraries.

- If a build succeeds, we know all the source dependencies in the working copy, so we know which files at the least must be placed under version management.

- As mentioned, the automatic determination of header file dependencies is also safer than what we can get in Make derivates. It is common to use tricks like gcc -MM to find header file dependencies automatically in Makefile rules, but this has subtle failure conditions. For instance, gcc -MM causes the found header files to be declared as dependencies, but it neglects to *add the files that were not found*. If the C compiler search path contains directories A and B, and a header X is found in B/X, then the non-existent file A/X should also be a dependency. After all, if A/X were to be added by the user, a rebuild would be necessary. Since we recompute the closure of the dependencies on every invocation, the new file A/X *will* be found and will trigger a rebuild.

However, there is an unresolved problem with Nix as a build manager: how can we use it *recursively*, that is, from within a Nix builder? It is quite common for a Nix builder to call conventional build tools such as Make. That is what almost all components in Chapter 7 do. But can we call nix-build in a builder? The problem is that the Nix store is global, and nix-build would compute and create all sorts of paths that are not known inputs or outputs of the current Nix build. A solution might be for the recursive Nix invocations to use a private store (e.g., a subdirectory of the builder's temporary directory).

Instead of recursive Nix builds, the alternative is to have one gigantic build graph. For instance, if we are building a component that needs a C compiler, the Nix expression for that component simply imports the Nix expression that builds the compiler. The problem with this approach is scalability: the resulting build graphs would become huge. The graph for a simple component such as GNU Hello would include the build graphs for dozens of large components, such as Glibc, GCC, etc. The resulting graph could easily have hundreds of thousands of nodes, far exceeding the graphs typically occurring in deployment (e.g., the one in Figure 1.5). However, apart from its efficiency, this is possibly the most desirable solution because of its conceptual simplicity. Thus it is interesting to develop efficient ways of dealing with very large build graphs.

10.3. Related work

The archetypical build manager is Make [56], introduced in Seventh Edition Unix (1979). According to Stuart Feldman's 2003 ACM Software System Award citation, "there is probably no large software system in the world today that has not been processed by a version or offspring of MAKE." It builds systems from a *Makefile* that describes a dependency graph using rules of the form

$$\begin{array}{l} \textit{targets} : \textit{dependencies} \\ \qquad\qquad \textit{actions} \end{array}$$

with the semantics that if the files *targets* do not exist or are older than the files *dependencies*, then the shell commands listed in *actions* should be run to (re-)build *targets*. Makefiles have some abstraction facilities to reduce repetitive rules, such as *pattern rules* (generic rules that are applied automatically on the basis of file extensions) and variables.

Make has spawned dozens of derivates. Some are backwards-compatible extensions, such as BSD Make and GNU Make [68]. Others are not, e.g., Plan 9's mk [96], AT&T and Lucent's NMake [70], Microsoft's NMake, Jam [148], and so on. Some of these add support for parallel builds [7], have syntactic sugar to allow more concise Makefiles, or have built-in support for languages such as C and C++ (e.g., to find dependencies). There are also tools that *generate* Makefiles from higher-level specifications, e.g., Automake [65, 172]. The generators unfortunately do not shield the user from the lower layers; it is the user's job to map problems that occur in a Makefile back to the high-level specification from which it was generated.

Ant [63] is a very popular build manager, but its popularity stems mainly from the fact that it comes with a large set of useful tasks for Java development, contrary to Make. (Make's built-in rules essentially reflect the Unix programming language landscape of 1979.) However, Ant in many ways is a step backwards compared to Make[3]. The (XML-based) build formalism is actually quite crude, and describes a simple partial ordering of build tasks. Ant's core provides no incremental build facilities; these must be implemented by tasks themselves. Ant however shows that it is important for a build tool to have good libraries for building common languages.

[3] Ant's homepage [63] lists two reasons for its development: Make uses non-portable shell commands, and Makefiles use TAB characters. This indicates that Ant's development was not inspired by any fundamental reflection on Make's limitations.

An interesting performance improvement is provided by *optimistic Make* [21], which starts building targets before the user has issued the make command, putting unused CPU cycles to good use. A median reduction in response time as high as a factor of 5.1 is reported.

However, none of these tools solve Make's fundamental problems, which are:

- A lack of abstraction facilities. The abstraction mechanisms—variables and pattern rules—are all global. Hence, if we need to specify different ways of building targets, we cannot use them, unless we split the system into multiple Makefiles. This, however, creates the much greater problem of incomplete dependency graphs [121]. [71] shows some of the rather clumsy techniques used for stretching Make's expressiveness.

- A lack of support for variant builds. Makefile variables allow variability to be expressed, but variants will overwrite each other unless tortuous hacks are used to arrange otherwise.

- The inability to ensure complete dependency specifications. If builds are not performed in isolation but rather in the workspace that holds all the sources, there is no way to prevent undeclared dependencies (unless file system calls are intercepted, as discussed below).

However, there are a number of build managers that do fundamentally improve on Make. Odin [28, 29] 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 in [150].) For example, the expression hello.c denotes a source, while hello.c :exe denotes the executable obtained by compiling and linking hello.c. Odin also supports variants automatically. Variant builds can be expressed easily, e.g., hello.c +debug :exe.

Vesta (already discussed in Section 7.6) has a build manager, the *Vesta evaluator*, that builds software from specifications written in the System Description Language (SDL), which is a functional language [94]. Since Vesta uses a virtual (NFS-based) file system, it can detect all dependencies used by a build action. The results of builds are cached and can be shared between users. Since the language is functional, it is easy to specify and build variants. Thus Vesta solves all three of the Make limitations listed above. The price is the use of a virtual file system and the integration with a specific version management system.

Amake [9, 8, 164] is the build tool for the Amoeba distributed operating system. Like Odin, it separates the specification of build tools and system specifications. Given a set of sources Amake automatically completes the build graph, that is, it finds instances of tool definitions that build the desired targets from the given sources. This is contrary to the functional model of explicit tool application in Odin, Vesta, and Nix. The obvious advantage is that specifications become shorter; the downside is that it becomes harder to specify alternative ways of building, and to see what's going on.

Maak [46], by the author of this thesis, was in many ways a precursor to Nix. It has a functional language similar to the Nix expression language, and supported build variability automatically; in fact, the language has some features that the Nix expression language does not, such as automatic propagation of attributes. The Nix language backs away from this level of expressivity since a) the Nix language is intended foremost for deployment instead of build management, and so involves less complicated build descriptions; and b) concise syntax is not always a blessing; it may make the language harder to understand.

Also, Maak allows *circular dependencies*, a classical "hard" problem for build managers. A notable example is LaTeX, which requires any number of runs until certain outputs stop changing. Circular dependencies are fundamentally incompatible with Nix, since store derivations describe a static set of build actions from which all variability has been removed (and in the extensional model, the output paths must be known in advance). I now feel that circular dependencies are an unnecessary complication for build managers: they occur quite seldom—in fact, LaTeX is the only common example—and so are better fixed by wrapping the offending tool in a script that executes it an appropriate number of times.

The Maak paper [46] also introduced the notion of transparent source/binary deployment, but in the absence of a cryptographic hashing scheme, it used nominal version equality, which is of course less safe. But Maak's fundamental problem was that it could not ensure complete dependency specifications, since (like Make) it performed builds "locally", in the workspace holding the sources. (On Linux, it did have a feature to use the strace command to intercept file system accesses, allowing dependency detection.) This prompted the development of a component store based on cryptographic hashing.

Part IV.

Conclusion

11. Conclusion

The main objective of the research described in this thesis was to develop a system for *correct* software deployment that ensures that the deployment is *complete* and does not cause *interference*. This objective was successfully met in the Nix deployment system, as the experience with Nixpkgs described in Section 7.1.5 has shown.

The objective of improving deployment correctness is reached through the two main ideas described in this thesis. The first is the use of cryptographic hashes in Nix store paths. It gives us isolation, automatic support for variability, and the ability to determine runtime dependencies. This however can be considered an (important) implementation detail—maybe even a “trick”. However, it addresses the deployment problem at the most fundamental level: the storage of components in the file system.

The second and more fundamental idea is the purely functional deployment model, which means that components never change after they have been built and that their build processes only depend on their declared inputs. In conjunction with the hashing scheme, the purely functional model prevents interference between deployment actions, provides easy component and composition identification, and enables reproducibility of configurations both in source and binary form—in other words, it gives predictable, deterministic semantics to deployment actions.

Nix is a work in progress. The implementation of the extensional model (Chapter 5) is in use. The experience of the Nix Packages collection shows that it works very well for its primary goal of software deployment (Chapter 7), though more experience is necessary to make conclusive statements about the scalability of the purely functional model. The intensional model (Chapter 6) exists in prototype form and must be fully implemented. Services deployed using Nix (Chapter 9) are in production use, as is a build farm (Chapter 8). Build management (Chapter 10) works, as its properties are really a direct application of Nix, but more work is necessary to improve its usability (e.g., by providing good libraries for common languages and tasks) and scalability.

Our experience with Nix over the last two years allows the following conclusions to be drawn:

- The purely functional deployment model implemented in Nix and the cryptographic hashing scheme of the Nix store in particular give us important features that are lacking in most deployment systems, such as complete dependencies, complete deployment, side-by-side deployment, atomic upgrades and rollbacks, transparent source/binary deployment and reproducibility (see Section 1.5).

- The cryptographic hashing scheme is effective in preventing undeclared dependencies, assuming a fully Nixified environment. In other environments, the techniques of Section 7.1.3 are successful in preventing most impurities.

- Nix places a number of assumptions on components (Section 6.8), but the large and varied set of components deployed through Nix shows that these assumptions are quite reasonable (Section 7.1.5).

- Hash rewriting is a technique that intuitively might appear unlikely to work, but in fact does work. It enables the intensional model, which is a breakthrough in that it enables secure sharing, multiple output paths, and some future benefits discussed below.

However, there are also a number of things that we *cannot* conclude on the basis of our current experience:

- That a functional language is a good way to describe components and compositions. Certainly such a language makes it possible and easy to describe variability, and function arguments seem a natural way to describe dependencies. But the *usability* of functional languages (and the Nix expression language in particular) has not been demonstrated; all users of the Nix expression language had prior exposure to functional languages such as Haskell.

- Nix opens up the perspective of unifying the various points on the software build and deployment timeline into a single formalism; e.g., Nix is not just a deployment tool, but as we have seen in Chapter 10, it can replace "low-level" build management tools such as Make. This is good because it prevents "impedance mismatch" between build management and deployment. For instance, it is not necessary to specify the same dependency both in the build formalism and the deployment formalism. But it remains to be seen whether low-level build management can be made scalable. Also, it is necessary to subsume the *source configuration* stage, currently implemented by tools such as Autoconf; this has not been done yet.

Finally, there are also a number of reasonable criticisms that can be levelled at the purely functional model:

- Efficient upgrading remains a problem. Using patch deployment, upgrades can be done efficiently in terms of network bandwidth. But a change to a fundamental dependency can still cause disk space requirements to double. This is a real problem compared to destructive updates. However, it can be argued that disk space is abundant, and that software components no longer dominate disk space consumption.

- Preventing undeclared dependencies is good, but the lack of scoped composition mechanisms lessens its usefulness (as discussed on page 173).

- Most of our experience with Nix has been on Linux, which allows a self-contained, pure Nixpkgs (page 169). That is, every dependency of the components in Nixpkgs can be built and deployed through Nix. This is an ideal situation that cannot be achieved in most other operating systems. A GUI application on Mac OS X will typically have a dependency on the *Cocoa* or *Carbon* GUI libraries. Since these are closed source and cannot be copied legally, we can neither build them nor use the techniques of Section 7.1.4 to deploy them in binary form. Thus we are forced to resort to impurity (e.g., by linking against the non-store path /System/Library/Frameworks/Cocoa.framework/...), reducing Nix's effectiveness.

11.1. Future work

There are a number of possibilities for further research, as well as a several remaining practical issues.

A fully Nixified system Since Nix works best in a "pure" environment where *all* components are built and deployed by Nix, it is natural to consider an operating system distribution on that basis; that is, a Unix system (probably based on Linux or FreeBSD) that has no /bin, /lib, /usr, etc.—all components are in /nix/store. In addition, the configuration of the system can be managed along the lines of the services in Chapter 9. For instance, the configuration of which services (such as daemons) should be enabled or disabled can be expressed in a Nix expression that builds a component that starts and stops other components passed as build-time inputs. To change the configuration, the Nix expression is changed and rebuilt.

A pure[1] Nix-based Linux system (tentatively called NixOS) is currently in active development and in a bootable state. Building components in this pure environment has already revealed a few impurities in Nixpkgs. On the other hand, the number of such impurities has turned out to be surprisingly low: once NixOS was up and running, it took only two one-line modifications (to stdenv, no less!) to get Firefox and all its dependencies to build.

The intensional model There is a prototype implementation of the intensional model that shows that hash rewriting works in practice (i.e., does not break components), but the prototype is not usable yet. A full implementation is necessary to answer a number of open questions. In particular, what is a good policy for path selection (page 149)?

The intensional model has the potential to enable better build management. A common efficiency problem in incremental building is that a small change to some file causes many rebuilds. For instance, a change to a C header file (say, config.h) forces us to rebuild all source files that include it. This is the case even for source files that include the header but are unaffected by the change.

Redundant rebuilds can be prevented in the intensional model by making the C preprocessor invocation an explicit build step. For source files that are unaffected by the header change, the preprocessed output will remain unchanged. We can optimise the build algorithm as follows. If a derivation d_1 differs only from a derivation d_2 in subderivations that yielded the same output, we can just copy the output of d_2 to d_1.

A language for builders We need to find a better way to write builders. Shell scripts are simply not a good language. The Unix shell is quite good as component glue—which is what we use it for—but it is not safe enough. It is in fact virtually impossible to write "correct" shell scripts that do not fail, unexpectedly, for banal reasons on certain inputs or in certain situations. Here are some of the main problems in the language:

- Variable uses (e.g., $foo) are "interpolated" by default. This is the reason why most shell scripts fail on file names that contain spaces or newlines. The same applies for automatic "globbing" (expansion of special characters such as "*").

[1] With the exception of /bin/sh, which is required by Glibc and several standards.

- Undefined variables do not signal an error.
- Non-zero exit codes from commands are ignored by default. Worse, it is extremely hard to catch errors from the left-hand sides of pipes.

In fact, the whole Unix tool chain is not conducive to writing correct code:

- Command-line arguments in Unix are usually overloaded, e.g., there is an ambiguity in most Unix tools between options and file names starting with a dash.
- The pipe model is that all tools read and write flat text files without a well-defined structure. This makes it hard to extract relevant data in a structured way. Typically, regular expressions in conjunction with `grep`, `sed`, `awk` and `perl` are used to extract data, but this tends to be brittle—the regular expressions hardly ever cover all possibilities.
- Similarly, the fact that options are flat strings makes it hard to pass structure, necessitating escaping. E.g., the command `grep -q "$filename"` to see whether a certain file name occurs in a list fails to take into account that `grep` interprets some characters in `$filename` as regular expression meta-characters.

These problems are far from specific to Nix builders. They have been the cause of many serious bugs, including security problems. So what we need is a shell with a clean semantics founded on the principle of least surprise [140], and a tool set based on the exchange of structured data (e.g., XML or ATerms). MSH (the Microsoft Shell) may be a good model.

It is also necessary to improve the interaction between Nix expressions and builders. The mechanism by which information is communicated to builders is rather crude, namely, through environment variables that only allow flat strings to be passed. It is much better to pass structured information, such as attribute sets and lists. An obvious way is to convert such expressions to XML (which is especially attractive in conjunction with an XML-based tool set).

A type system A type system for the Nix expression language is desirable. Of course, to the greatest extent possible types should be inferred. We probably need a basic Hindley-Milner type system [122] extended with extensible records [111] to support attribute sets.

Efficient storage Binary patch deployment greatly reduces the bandwidth needed to deploy binary upgrades, but the upgraded components still need to be stored in full, and an amount of CPU time linear in the size of the upgraded components is also required. These problems may be solved by using a file system that supports *delta storage* [115], i.e., that can store file revisions as deltas to previous revisions. Actually, for the Nix store, we need a file system where revisions can refer to revisions of *other* files. Concretely, we need a kernel API $\mathsf{copy}(p_{dst}, p_{src}, patch)$ that creates a path p_{dst} with contents equal to those of p_{src} with *patch* applied. Then FSO creation using patches takes disk space and CPU time linear in the size of the patch, not the resulting FSO.

```
<item id=’zlib-1.2.1-security’ type=’security’>
  <condition>
    <within>
      <traverse> <true /> </traverse>
      <hasAttr name=’outputHash’ value=’ef1cb003448b...’ />
    </within>
  </condition>
  <reason>
    Zlib 1.2.1 is vulnerable to a denial-of-service condition.  See
    http://www.kb.cert.org/vuls/id/238678.  Upgrade to 1.2.2.
  </reason>
  <severity class="server" level="critical" />
  <severity class="client" level="medium" />
</item>
```

Figure 11.1.: A blacklist entry

Blacklisting The purely functional model creates a serious problem in some situations: how can we be sure that all uses of a certain “bad” component have been eliminated? For example, suppose that we find that a certain fundamental component (say, Zlib) contains a security bug [1]. This component is used by many other components. Thus we want to ensure that we upgrade all installed applications in all user environments to new instances that do not use the bad component anymore. Of course, we can just run nix-env -u "*" and hope that this gets rid of all bad components, but there is no guarantee of that, especially if we have installed components from arbitrary third parties (i.e., components not in Nixpkgs). The purely functional model’s non-interference property now bites us. Thus there should be a generic mechanism that detects uses of bad components on the basis of a *blacklist* that describes these components.

A simple prototype of this idea has been implemented. It evaluates an XML specification that contains declarations such as the one shown in Figure 11.1. This example specifies that any output path built from a derivation graph that at any point used the Zlib 1.2.1 source distribution (as identified by its cryptographic hash, ef1cb003448b...), is tainted. The condition is applied to every element of every user environment, and prints out a warning if a match is found. Note that since the condition inspects the derivation graph (found through the deriver link described in Section 5.5.1), it also matches statically linked uses of Zlib. Thus the blacklisting approach finds dangerous uses of libraries that cannot be fixed by destructive upgrading in conventional deployment models.

A blacklisting approach should also specify what should be done if a match is found, which may depend on the type of the machine, user environment, or application: for a server the Zlib bug is critical, for a client less so.

Upgrading in the intensional model The technique of hash rewriting in the intensional model (Section 6.3.2) allows something very similar to destructive upgrading in conventional systems, but in safe way. We can *generically* upgrade all components that reference some “bad” path p to a “good” path p', by rewriting (using a variation of algorithm resolve in Section 6.4.1) every use of p to p'. The difference with the normal mode of upgrading in Nix (deploying replacement Nix expressions) is that no Nix expressions are necessary. That's also the problem of this technique: the resulting components do not have a deriver, that is, they do not necessarily correspond to the output of any known derivation. But it is *not* destructive: since resolve works by rewriting copies, the old components are still available. Thus, atomicity and the ability to roll back still hold.

Feature selection and automatic instantiation Nix supports component variability both in the expression language and at the store level, but current deployment policies are not very good at making it available to users. For instance, nix-env does not provide a way to pass arguments to functions; the top-level Nix expression should evaluate to a set of derivations. Similarly, deployment policies such as channels do not support local customisation.

At the very least, tools such as nix-env should be extended to allow variation points to be bound (i.e., to allow the user to set parameters). One can imagine this being done through a nice graphical user interface, e.g., as in the Linux kernel configuration tool, Consul@GUI [13] or the Software Dock [85]. Such interfaces lead to an interesting problem: if the user binds some but not all of the variation points, how can we find component instances or compositions that satisfy all constraints (as expressed by assertions in Nix expressions)? This is of course an NP-complete problem (since it is essentially just the boolean satisfiability problem [35]), but that may not be a problem in practice. For example, CML2 [139] finds Linux kernel instances satisfying user-specified constraints automatically, and Binary Decision Diagrams (BDD) also appear to be a useful technique for this purpose [170].

Bibliography

[1] Mark Adler and Jean-loup Gailly. Zlib advisory 2002-03-11. http://www.zlib.net/advisory-2002-03-11.txt, 2002.

[2] Eric Anderson and Dave Patterson. A retrospective on twelve years of LISA proceedings. In *Proceedings of the 13th Systems Administration Conference (LISA '99)*, pages 95–107. USENIX Association, November 1999.

[3] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[4] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[5] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[6] Erik H. Baalbergen. Parallel and distributed compilations in loosely-coupled systems: A case study. In *Workshop on Large Grain Parallelism*, Providence, RI, October 1986. Also in [123].

[7] Erik H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, 1988.

[8] Erik H. Baalbergen. *The declarative operating system model*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1992.

[9] Erik H. Baalbergen, Kees Verstoep, and Andrew S. Tanenbaum. On the design of the Amoeba configuration manager. In *2nd International Workshop on Software Configuration Management (SCM-2)*, volume 17/7 of *ACM SIGSOFT Software Engineering Notes*, pages 15–22, November 1989.

[10] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume II of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, second edition, 1984.

[11] Bryan Kendall Beatty. Compact text encoding of latitude/longitude coordinates. United States Patent Application 20050023524, February 2005.

[12] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifiers (URI): Generic syntax. RFC 2396, August 1998.

[13] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. In Jilles van Gurp and Jan Bosch, editors, *Proceedings of the Software Variability Management Workshop (SVM '2003)*, February 2003.

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

[15] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[16] Don Box. *Essential COM*. Addison-Wesley, 1998.

[17] Don Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, November 2002.

[18] Richard S. Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[19] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[20] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau, editors. *Extensible Markup Language (XML) 1.0*. W3C, third edition, February 2004.

[21] Rick Bubenik and Willy Zwaenepoel. Optimistic Make. *IEEE Transactions on Software*, 41(2), February 1992.

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

[23] Brent Callaghan, Brian Pawlowski, and Peter Staubach. Nfs version 3 protocol specification. RFC 1813, June 1995.

[24] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, University of Colorado, April 1998.

[25] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[26] Anders Christensen and Tor Egge. Store — a system for handling third-party applications in a heterogeneous computer environment. In *Selected papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*, number 1005 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[27] James Clark, editor. *XSL Transformations (XSLT) Version 1.0.* W3C, November 1999.

[28] Geoffrey Clemm. *The Odin System — An Object Manager for Extensible Software Environments.* PhD thesis, University of Colorado at Boulder, February 1986.

[29] Geoffrey Clemm. The Odin system. In *Selected papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*, number 1005 in Lecture Notes in Computer Science, pages 241–262. Springer-Verlag, 1995.

[30] Geoffrey Clemm, Jim Amsden, Tim Ellison, Christopher Kaler, and Jim Whitehead. Versioning extensions to WebDAV (web distributed authoring and versioning). RFC 3253, March 2002.

[31] CollabNet. Subversion. http://subversion.tigris.org/, 2005.

[32] Wallace Colyer and Walter Wong. Depot: A tool for managing software environments. In *Proceedings of the 6th Systems Administration Conference (LISA VI)*, pages 153–162. USENIX Association, October 1992.

[33] International Electrotechnical Commission. *Letter symbols to be used in electrical technology — Part 2: Telecommunications and electronics.* IEC, second edition, November 2000. IEC 60027-2.

[34] The Unicode Consortium. *The Unicode Standard, Version 4.0.* Addison-Wesley, 2003.

[35] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[36] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[37] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 2B—Instruction Set Reference, N–Z. Intel Corporation, June 2005. Order number 253667-016.

[38] Microsoft Corporation. Joliet specification — version 1, May 1995.

[39] Microsoft Corporation. Binary delta compression. Whitepaper, March 2002.

[40] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[41] Homayoun Dayani-Fard, Yijun Yu, John Mylopoulos, and Periklis Andritsos. Improving the build architecture of legacy C/C++ software systems. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering: 8th International Conference (FASE 2005)*, volume 3443 of *Lecture Notes in Computer Science*, pages 96–110, Edinburgh, Scotland, April 2005. Springer-Verlag.

[42] Merijn de Jonge. *To Reuse or To Be Reused.* PhD thesis, University of Amsterdam, March 2003.

[43] Merijn de Jonge. Build-level components. *IEEE Transactions on Software Engineering*, 31(7):588–600, July 2005.

[44] Edsger W. Dijkstra. Notes on structured programming. Technical Report 70-WSK-03, Technische Hogeschool Eindhoven, March 1970.

[45] Jeff Dike. A user-mode port of the Linux kernel. In *4th Annual Linux Showcase & Conference (ALS 2000)*, October 2000.

[46] Eelco Dolstra. Integrating software construction and software deployment. In Bernhard Westfechtel, editor, *11th International Workshop on Software Configuration Management (SCM-11)*, volume 2649 of *Lecture Notes in Computer Science*, pages 102–117, Portland, Oregon, USA, May 2003. Springer-Verlag.

[47] Eelco Dolstra. Efficient upgrading in a purely functional component deployment model. In George Heineman et al., editor, *Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005)*, volume 3489 of *Lecture Notes in Computer Science*, pages 219–234, St. Louis, Missouri, USA, May 2005. Springer-Verlag.

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

[49] Eelco Dolstra, Martin Bravenboer, and Eelco Visser. Service configuration management. In *12th International Workshop on Software Configuration Management (SCM-12)*, September 2005.

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

[51] Eelco Dolstra and Eelco Visser. Building interpreters with rewriting strategies. In Mark van den Brand and Ralf Lämmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science Publishers.

[52] Eelco Dolstra, Eelco Visser, and Merijn de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.

[53] ECMA. *ECMA-48: Control Functions for Coded Character Sets*. ECMA, fifth edition, June 1991. Also ISO/IEC 6429:1992.

[54] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz — open source graph drawing tools. In *Graph Drawing 2001*, number 2265 in Lecture Notes in Computer Science, pages 483–485. Springer-Verlag, 2001.

[55] Robert Ennals and Simon Peyton Jones. HsDebug: debugging lazy programs by not being lazy. In Johan Jeuring, editor, *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 84–87, Uppsala, Sweden, August 2003. ACM Press.

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

[57] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616, June 1999.

[58] Gert Florijn, Ernst Lippe, Atze Dijkstra, Norbert van Oosterom, and Doaitse Swierstra. Camera: Cooperation in open distributed environments. In *Proceedings of the EurOpen and USENIX Spring 1992 Workshop/Conference*, pages 123–136. EurOpen and USENIX, April 1992.

[59] Gert Florijn, Leo Soepenberg, and Atze Dijkstra. Mapping objects to files: a UNIX file system interface to an object management system. In H. Wijshoff, editor, *Proceedings Computing Science in the Netherlands (CSN'93)*, 1993. Also published as Utrecht University, Department of Information and Computing Sciences technical report RUU-CS-93-08.

[60] Karl Fogel and Moshe Bar. *Open Source Development with CVS*. Paraglyph Press, July 2003.

[61] Internet Engineering Task Force. Secure shell (secsh) working group. http://www.ietf.org/html.charters/secsh-charter.html, 2005. Accessed 21 August 2005.

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

[63] Apache Software Foundation. Apache Ant. http://ant.apache.org/, 2005. Accessed 15 August 2005.

[64] Free Software Foundation. Autoconf. http://www.gnu.org/software/autoconf/, 2005.

[65] Free Software Foundation. Automake. http://www.gnu.org/software/automake/, 2005.

[66] Free Software Foundation. Bison. http://www.gnu.org/software/bison/, 2005.

[67] Free Software Foundation. GNU Hello. http://www.gnu.org/software/hello/, 2005.

[68] Free Software Foundation. GNU Make. http://www.gnu.org/software/make/, 2005.

[69] Mozilla Foundation. Tinderbox. http://www.mozilla.org/tinderbox.html, 2005.

[70] Glenn Fowler. The fourth generation Make. In *USENIX 1985 Summer Conference*, June 1985.

[71] Glenn Fowler. A case for make. *Software—Practice and Experience*, 20(S1):S1/35–S1/46, 1990.

[72] Martin Fowler and Matthew Foemmel. Continuous integration. http://www.martinfowler.com/articles/continuousIntegration.html. Accessed 11 August 2005.

[73] FreeBSD Project. FreeBSD Ports Collection. http://www.freebsd.org/ports/.

[74] Ned Freed and Nathaniel S. Borenstein. Multipurpose internet mail extensions (MIME) part two: Media types. RFC 2046, November 1996.

[75] freedesktop.org. pkg-config. http://pkgconfig.freedesktop.org/wiki/, 2005. Accessed 21 August 2005.

[76] Alan O. Freier, Philip L. Karlton, and Paul C. Kocher. The SSL protocol — version 3.0. IETF Internet Draft draft-freier-ssl-version3-02.txt, November 1996.

[77] Gentoo Foundation. Gentoo Linux. http://www.gentoo.org/.

[78] Bob Glickstein. GNU Stow. http://www.gnu.org/software/stow/, 2005.

[79] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, June 2005.

[80] Mark Grechanik and Dewayne Perry. Secure deployment of components. In Wolfgang Emmerich and Alexander L. Wolf, editors, *2nd International Working Conference on Component Deployment (CD 2004)*, volume 3083 of *Lecture Notes in Computer Science*. Springer-Verlag, May 2004.

[81] Filesystem Hierarchy Standard Group. Filesystem hierarchy standard, version 2.3. http://www.pathname.com/fhs/pub/fhs-2.3.html, 2004.

[82] Free Standards Group. Linux Standard Base Core Specification 3.0. http://www.linuxbase.org/, 2005.

[83] Nicholas Gulrajani and Karen Wade. Bridging the chasm between development and operations. Product white paper, ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/Bridging_the_chasm_CC-TCM.pdf, June 2004.

[84] Richard S. Hall, Dennis Heimbigner, André van der Hoek, and Alexander L. Wolf. An architecture for post-development configuration management in a wide area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, Maryland, USA, May 1997.

[85] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the Software Dock. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 174–183, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[86] Richard S. Hall, Dennis M. Heimbigner, and Alexander L. Wolf. Requirements for software deployment languages and schema. In *8th International Symposium on System Configuration Management (SCM-8)*, volume 1439 of *Lecture Notes in Computer Science*, pages 198–203. Springer-Verlag, 1998.

[87] John Hart and Jeffrey D'Amelia. An analysis of RPM validation drift. In *Proceedings of the 16th Systems Administration Conference (LISA '02)*, pages 155–166. USENIX Association, November 2002.

[88] Red Hat. Cygwin. http://www.cygwin.com/, 2005. Accessed 21 August 2005.

[89] Les Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, 1997.

[90] Jan Heering, Paul R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF — reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[91] Armijn Hemel. Using buildfarms to improve code. In *UKUUG Linux 2003 Conference*, August 2003.

[92] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The Vesta approach to software configuration management. Technical Report Research Report 168, Compaq Systems Research Center, March 2001.

[93] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The Vesta software configuration management system. Technical Report Research Report 177, Compaq Systems Research Center, January 2002.

[94] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 311–320. ACM Press, 2000.

[95] Graydon Hoare. Monotone. http://www.venge.net/monotone/, 2005.

[96] Andrew Hume. Mk: a successor to make. In *USENIX 1987 Summer Conference*, 1987.

[97] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[98] IEEE. *IEEE Std 1003.1-1988 Standard Portable Operating System Interface for Computer Environments (POSIX)*. IEEE, 1988.

[99] IEEE and The Open Group. The Open Group Base Specifications Issue 6 — IEEE Std 1003.1, 2004 Edition. http://www.opengroup.org/onlinepubs/009695399/, 2004.

[100] British Standards Institute. *The C Standard: Incorporating Technical Corrigendum 1*. Wiley, July 2003.

[101] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum No. 1*. Wiley, October 2003.

[102] Slinger Jansen, Gerco Ballintijn, and Sjaak Brinkkemper. A process model and typology for software product updaters. In *9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, March 2005.

[103] Steven C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[104] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, August 1996.

[105] Dan Kaminsky. MD5 to be considered harmful someday. http://www.doxpara.com/md5_someday.pdf, December 2004.

[106] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, March 1988.

[107] David G. Korn and Kiem-Phong Vo. vdelta: Differencing and compression. In Balachander Krishnamurthy, editor, *Practical Reusable UNIX Software*. John Wiley & Sons, 1995.

[108] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Technical Report LU-CS-TR-99-214, Lund University, 1999.

[109] David B. Leblang. The CM challenge: configuration management that works. In W. F. Tichy, editor, *Configuration management*, pages 1–37. John Wiley & Sons, New York, NY, USA, 1995.

[110] Daan Leijen. *The λ Abroad: A Functional Approach to Software Components*. PhD thesis, Universiteit Utrecht, November 2003.

[111] Daan Leijen. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

[112] Thomas Leonard. The Zero Install system. http://zero-install.sourceforge.net/. Accessed 22 July 2005.

[113] Michael E. Lesk and Eric Schmidt. lex: A lexical analyzer generator. In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[114] Ernst Lippe. *Camera — Support for Distributed Cooperative Work*. PhD thesis, Rijksuniversiteit Utrecht, October 1992.

[115] Joshua P. MacDonald. File system support for delta compression. Master's thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[116] Kenneth Manheimer, Barry A. Warsaw, Stephen N. Clark, and Walter Rowe. The Depot: A framework for sharing software installation across organizational and UNIX platform boundaries. In *Proceedings of the 4th Systems Administration Conference (LISA '90)*, pages 37–46. USENIX Association, October 1990.

[117] Mauro Marinilli. *Java Deployment with JNLP and WebStart*. Sams, September 2001.

[118] Steve McConnell. *Software Project Survival Guide*. Microsoft Press, 1998.

[119] Bertrand Meyer. The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 660–667, May 2003.

[120] Ondrej Mikle. Practical attacks on digital signatures using MD5 message digest. Cryptology ePrint Archive, Report 2004/356, 2004. http://eprint.iacr.org/.

[121] Peter Miller. Recursive make considered harmful. http://aegis.sourceforge.net/auug97.pdf, 1997. Accessed 12 August 2005.

[122] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[123] Sape J. Mullender, editor. *The Amoeba distributed operating system: selected papers 1984–1987*. Number 41 in CWI tracts. Centrum voor Wiskunde en Informatica, Amsterdam, 1987.

[124] NCSA. Common gateway interface, version 1.1. http://hoohoo.ncsa.uiuc.edu/cgi/. Accessed 4 August 2005.

[125] Tobias Oetiker. SEPP: Software installation and sharing system. In *Proceedings of the 12th Systems Administration Conference (LISA '98)*, pages 253–259. USENIX Association, December 1998.

[126] National Institute of Standards and Technology. Advanced encryption standard. http://www.nist.gov/aes/, 2001. Accessed 21 August 2005.

[127] National Institute of Standards and Technology. Secure hash standard, February 2004.

[128] Michael A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 205–218, January 1993.

[129] Kyle Oppenheim and Patrick McCormick. Deployme: Tellme's package management and deployment system. In *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, pages 187–196. USENIX Association, December 2000.

[130] Vern Paxson. Flex — a fast scanner generator. http://www.gnu.org/software/flex/, 2005.

[131] Colin Percival. An automated binary security update system for FreeBSD. In *Proceedings of BSDCON '03*. USENIX, September 2003.

[132] Colin Percival. Binary diff/patch utility. http://www.daemonology.net/bsdiff/, 2003.

[133] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

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

[135] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2004.

[136] Rob Pike. Systems software research is irrelevant. Research talk, http://www.cs.bell-labs.com/who/rob/utah2000.pdf, February 2000.

[137] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly, June 2004.

[138] The Fink Project. Fink. http://fink.sourceforge.net/.

[139] Eric S. Raymond. The CML2 language: Python implementation of a constraint-based interactive configurator. In *9th International Python Conference*, March 2001.

[140] Eric S. Raymond. *The Art of Unix Programming*. Addison-Wesley, September 2003.

[141] Ron Rivest. The MD5 message-digest algorithm. RFC 1321, April 1992.

[142] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[143] John P. Rouillard and Richard B. Martin. Depot-Lite: a mechanism for managing software. In *Proceedings of the 8th Systems Administration Conference (LISA '94)*, pages 83–91. USENIX Association, 1994.

[144] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[145] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, second edition, 1996.

[146] Bruce Schneier. New cryptanalytic results against SHA-1. Crypto-Gram newsletter, http://www.schneier.com/blog/archives/2005/08/new_cryptanalyt.html, August 2005.

[147] Bruce Schneier. SHA-1 broken. Crypto-Gram newsletter, http://www.schneier.com/crypto-gram-0503.html, March 2005.

[148] Christopher Seiwald. Jam — Make(1) redux. In *USENIX Applications Development Symposium*, April 1994.

[149] Gustavo Noronha Silva. APT HOWTO. http://www.debian.org/doc/manuals/apt-howto/index.en.html, 2004. Accessed 26 August 2005.

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

[151] Guy L. Steele. RABBIT: A compiler for SCHEME. Technical Report AI-TR-474, MIT AI Lab, May 1978.

[152] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley, second edition, June 2005.

[153] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[154] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[155] Clemens Szyperski. Component technology—what, where, and how? In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 684–693, May 2003.

[156] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, fourth edition, August 2002.

[157] Peter Thoeny. Twiki—an enterprise collaboration platform. http://twiki.org/, 2005.

[158] ThoughtWorks. Cruise Control. http://cruisecontrol.sourceforge.net/, 2005.

[159] Walter F. Tichy. Tools for software configuration management. In Jürgen F. H. Winkler, editor, *Proceedings of the International Workshop on Software Version and Configuration Control*, volume 30 of *Berichte des German Chapter of the ACM*, Grassau, January 1988. Teubner.

[160] TIS Committee. Tool Interface Specification (TIS) Executable and Linking Format (ELF) Specification, Version 1.2. http://www.x86.org/ftp/manuals/tools/elf.pdf, May 1995.

[161] TraCE Project. Nix deployment system. http://www.cs.uu.nl/wiki/Trace/Nix, 2005.

[162] Dimitre Trendafilov, Nasir Memon, and Torsten Suel. zdelta: An efficient delta compression tool. Technical Report TR-CIS-2002-02, Polytechnic University, 2002.

[163] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, February 1999.

[164] Vrije Universiteit. *The Amoeba Reference Manual — Programming Guide*, chapter Chapter 6.3 (Amake User Reference Manual), pages 83–104. Vrije Universiteit, Amsterdam, 1996.

[165] Urbancode. Anthill. http://www.urbancode.com/projects/anthill/default.jsp, 2005. Accessed 21 August 2005.

[166] Mark van den Brand, Hayco de Jong, Paul Klint, and Pieter Olivier. Efficient annotated terms. *Software—Practice and Experience*, 30:259–291, 2000.

[167] André van der Hoek. Integrating configuration management and software deployment. In *Working Conference on Complex and Dynamic Systems Architecture (CDSA 2001)*, December 2001.

[168] André van der Hoek, Richard S. Hall, Antonio Carzaniga, Dennis Heimbigner, and Alexander L. Wolf. Software deployment: Extending configuration management support into the field. *Crosstalk, The Journal of Defense Software Engineering*, 11(2), February 1998.

[169] André van der Hoek, Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. Software release management. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 159–175. Springer-Verlag, 1997.

[170] Tijs van der Storm. Variability and component composition. In Jan Bosch and Charles Krueger, editors, *Software Reuse: Methods, Techniques and Tools: 8th International Conference (ICSR-8)*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, June 2004.

[171] Tijs van der Storm. Continuous release and upgrade of component-based software. In *12th International Workshop on Software Configuration Management (SCM-12)*, September 2005.

[172] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders, October 2000. Also at http://sources.redhat.com/autobook/, accessed 21 august 2005.

[173] V. N. Venkatakrishnan, R. Sekar, Tapan Kamat, Sofia Tsipa, and Zhenkai Liang. An approach for secure software installation. In *Proceedings of the 16th Systems Administration Conference (LISA '02)*, pages 219–226. USENIX Association, November 2002.

[174] Seth Vidal. Yellow dog Updater, Modified (yum). http://linux.duke.edu/projects/yum/.

[175] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[176] Eelco Visser. Program transformation with Stratego/XT: Strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[177] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *Advances in Cryptology — CRYPTO 2005: 25th Annual International Cryptology Conference*, volume 3621 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2005.

[178] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 3494 of *Lecture Notes in Computer Science*. Springer-Verlag, May 2005.

[179] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1992.

[180] Stephen P. Berczuk with Brad Appleton. *Software configuration management patterns: effective teamwork, practical integration*. Addison-Wesley, 2003.

[181] Walter Wong. Local disk depot: customizing the software environment. In *Proceedings of the 7th Systems Administration Conference (LISA '93)*, pages 49–53. USENIX Association, November 1993.

[182] X.Org Foundation. X.Org homepage. http://wiki.x.org/wiki/, 2005. Accessed 27 November 2005.

Index

Samenvatting

De installatie van software en de daarmee samenhangende activiteiten lijken altijd te mislukken. Zo komt het regelmatig voor dat het installeren of upgraden van een softwaretoepassing leidt tot het falen van eerder geïnstalleerde toepassingen. Of het blijkt dat een applicatie afhankelijk is van componenten die niet aanwezig zijn op het systeem. Het is meestal ook niet mogelijk om dit soort acties terug te draaien als ze naderhand niet blijken te bevallen. Deze activiteiten worden aangeduid als *software deployment*, d.w.z. het in gebruik nemen of "uitrollen" van software. Het onderwerp van dit proefschrift is de ontwikkeling van betere technieken om deployment te ondersteunen.

De onderliggende problemen die de deploymentmalaise veroorzaken zijn een gebrek aan *isolatie* tussen componenten, en het feit dat het lastig is om *afhankelijkheden* (*dependencies*) tussen componenten correct te identificeren. Voorts schaalt deployment moeilijk op. Zo toont Figuur 1.5 op pagina 10 de afhankelijkheden van de browser Mozilla Firefox. Elke component kan in potentie de deployment van Firefox doen mislukken. Zo kan de component afwezig zijn, aanwezig zijn in een incompatibele versie, of aanwezig zijn in een compatibele versie maar met verkeerde compilerinstellingen gebouwd zijn.

Er zijn talloze deploymenttechnologieën, in de Unix-wereld ook wel *package managers* genaamd. Ik noem met name RPM, Debian APT, Gentoo Portage, .NET assemblies en Mac OS application bundles. Het belang van deployment—en het verlangen naar verbeteringen op dit vlak—wordt onderstreept door het feit dat Linux-distributies zich in de eerste plaats onderscheiden in hun package-managementsystemen. Deze hebben echter allemaal ernstige tekortkomingen. Ze zijn niet in staat om volledige afhankelijkheidsspecificaties af te dwingen, ze ondersteunen niet de aanwezigheid van meerdere versies van een component, enzovoorts.

Er is tot op heden betrekkelijk weinig onderzoek geweest naar deployment, en vrijwel helemaal niet naar de *low-level* aspecten zoals de opslag van componenten in het bestandssysteem (waar isolatie immers gerealiseerd moet worden). In plaats daarvan is vooruitgang op dit gebied voornamelijk geboekt door systeembeheerders en Unix-hackers. Dit is niet verbazingwekkend: deployment is een onderdeel van het vakgebied van systeembeheer, een onderwerp dat volgens Pike [136] vooralsnog "deeply difficult" is.

Het Nix deploymentsysteem

De Nix store Dit proefschrift beschrijft een betere aanpak, een zogenaamd *puur functioneel deploymentmodel*. Zo'n model is geïmplementeerd in een deploymentsysteem genaamd *Nix*. Nix slaat componenten in isolatie van elkaar op in een *Nix store*, een speciale directory in het bestandssysteem.

In Nix wordt elke component opgeslagen onder een speciaal pad zoals /nix/store/-2zbay49r2ihrznny6vbrcjvils4nqm1w-firefox-1.0.7. Het deel 2zbay49r2ihr... is een *cryptografische hash* van alle invoer die heeft bijgedragen aan de berekening van de component.

Daarom heet het model puur functioneel: net als in puur functionele programmeertalen wordt het resultaat van een bouwactie alleen bepaald door de opgegeven invoer. Cryptografische hashes hebben de plezierige eigenschap dat het erg moeilijk is om twee verschillende sets van invoeren te vinden die dezelfde hash opleveren. Hierdoor kunnen we er in de praktijk vanuit gaan dat als twee componenten op enige wijze verschillen, ze een verschillende hash opleveren en dus op verschillende locaties in het bestandssysteem worden opgeslagen. Dit zorgt voor de gewenste isolatie tussen componenten. De installatie van een applicatie kan niet langer leiden tot de beschadiging van reeds geïnstalleerde applicaties.

Het gebruik van hashes lost ook het andere grote deploymentprobleem op—onvolledige afhankelijkheidsinformatie. Ten eerste voorkomt het ongedeclareerde afhankelijkheden tijdens *bouwtijd*. Immers, bouwtools zoals compilers en linkers zoeken niet standaard in paden zoals /nix/store/2zbay49r2ihr...-firefox-1.0.7.

Ten tweede maakt het het ons mogelijk om te *scannen* naar de afhankelijkheden die kunnen optreden gedurende de executie van een component. Dit houdt in dat we de binaire inhoud van een component doorzoeken op de aanwezigheid van de hashonderdelen van de bestandsnamen van andere componenten. Stel dat de Nix store een GUI-component /nix/store/4kd0ma2pxf6w...-gtk+-2.8.6 bevat (een GUI-bibliotheek). Als we vervolgens de tekenreeks 4kd0ma2pxf6w... tegenkomen in de inhoud van /nix/store/2zbay49r2ihr...-firefox-1.0.7, kunnen we concluderen dat de Firefox-component een afhankelijkheid heeft op /nix/store/4kd0ma2pxf6w...-gtk+-2.8.6. De geldigheid van deze aanpak motiveer ik door middel van een analogie met technieken die worden gebruikt in de implementatie van programmeertalen, in het bijzonder *conservatieve garbage collection* (hoofdstuk 3).

De scanaanpak zorgt voor volledige kennis van de afhankelijkheden tussen componenten. Deployment van een component verloopt correct indien we de *afsluiting* (*closure*) van de component onder de afhankelijkheidsrelatie kopiëren. Voorts kunnen componenten veilig verwijderd worden: een component mag alleen verwijderd worden indien er geen componenten in de store meer naar verwijzen. Het is zelfs mogelijk om ongebruikte componenten volledig automatisch weg te laten gooien (*garbage collection*).

Abstractie over hashes Uiteraard dienen gebruikers en ontwikkelaars niet geconfronteerd te worden met bestandsnamen met eerdergenoemde hashes. Dat is gelukkig ook niet het geval. Ze worden verborgen voor gebruikers door zogeheten *user environments* die een verzameling "geactiveerde" componenten beschikbaar maken voor de gebruiker. User environments zijn zelf ook (automatisch gegenereerde) componenten. Ze kunnen dus gebroederlijk naast elkaar bestaan. Dit stelt de gebruikers van een systeem in staat om verschillende user environments te gebruiken. Ook maakt het een *rollback* mogelijk waarbij installatie- of upgradeacties teruggedraaid worden. Dit is een belangrijke eigenschap in bijvoorbeeld serveromgevingen.

Evenmin hoeven ontwikkelaars rechtstreeks met hashes te werken. Nix componenten worden namelijk gebouwd uit *Nix-expressies* (hoofdstukken 2 en 4). Dit is een eenvoudige functionele taal die beschrijft hoe componenten gebouwd en samengesteld moeten worden. Uit deze beschrijvingen berekent Nix de paden waar de componenten opgeslagen worden.

Transparant source/binary deploymentmodel Het uitrollen van Nix-expressies naar doelmachines levert een *source-deploymentmodel* op, omdat ze beschrijven hoe componenten uit broncode gebouwd kunnen worden. Zo'n model—ook toegepast in deploymentsystemen zoals Gentoo Linux—is flexibel en prettig voor ontwikkelaars, maar ook erg resource-intensief: het kost dikwijls veel tijd en schijfruimte om een component uit broncode te bouwen. Daarentegen is een *binair deploymentmodel*, waarin componenten in binaire vorm worden gedistribueerd naar doelmachines, efficiënter maar ook minder flexibel.

Nix combineert het beste van beide werelden door middel van een *transparant source/binary deploymentmodel*. Softwaredistributeurs of systeembeheerders bouwen een Nix-expressie en plaatsen de resulterende binaire componenten op een centrale server, geïndexeerd onder hun hashes. Doelmachines kunnen vervolgens het bestaan van deze voorgebouwde componenten registreren. Deze registraties heten *substituten*. De bouw van een Nix-expressie kan dan worden "kortgesloten" indien voor het te bouwen pad (zoals /nix/store/2zbay49r2ihr...-firefox-1.0.7) een substituut bekend is die opgehaald en uitgepakt kan worden. Op die manier verandert source-deployment op een voor de gebruiker transparante wijze automatisch in binaire deployment.

Extensioneel en intensioneel model Het proefschrift beschrijft twee subtiel verschillende modellen voor de implementatie van Nix. Het oorspronkelijke en momenteel gebruikte model is het *extensionele model* (hoofdstuk 5), waarin store-paden vooraf worden berekend uit Nix-expressies. Dit is nodig omdat het pad van een component bekend moet zijn vóórdat de component gebouwd wordt. Het maakt het echter lastig om een Nix store te delen tussen meerdere, elkaar wellicht wederzijds niet vertrouwende gebruikers. Immers, een kwaadwillende gebruiker kan zomaar een onzinnig substituut registreren voor een pad dat daarna geïnstalleerd wordt door een andere gebruiker.

Het *intensionele model* (hoofdstuk 6) heft deze beperking op door alle componenten op een *inhoudsgeadresseerde* wijze op te slaan. Hierbij wordt de naam van een component bepaald door de cryptografische hash over de inhoud van de component in plaats van over de Nix-expressie. Twee componenten kunnen dus alleen op dezelfde plaats in het bestandssysteem worden opgeslagen als ze inhoudelijk (intensioneel) gelijk zijn. Gebruikers kunnen hierdoor verschillende substituten registreren voor dezelfde Nix-expressie. Het resultaat is een Nix store die op veilige wijze door gebruikers gedeeld kan worden.

Toepassingen

De voornaamste toepassing van Nix is deployment (hoofdstuk 7). We hebben Nix gebruikt om meer dan 278 bestaande Unix-componenten uit te rollen. Deze pakketten zijn onderdeel van de *Nix Packages collection* (*Nixpkgs*, sectie 7.1). Nixpkgs dient zowel ter validatie van onze aanpak, als om gebruikers van Nix een bruikbare basis van pakketten aan te bieden waarop verder ontwikkeld kan worden.

Nix is echter ook toepasbaar op een aantal aan deployment grenzende probleemgebieden. Het streven is om zoveel mogelijk aspecten van het bouw- en deploymentproces onder één enkel formalisme te brengen, namelijk Nix-expressies.

Continue integratie en releasemanagement (hoofdstuk 8) Het is een goede gewoonte om tijdens het ontwikkelen van grote softwaresystemen het samenvoegen van de verschillende onderdelen niet te lang uit te stellen, daar dit kan leiden tot zogenaamde ‘Big Bang integratie” waar in een laat stadium geconstateerd wordt dat de onderdelen niet samenwerken. Bij *continue integratie* worden componenten voortdurend gebouwd vanuit het versiebeheersysteem, dus iedere keer dat een ontwikkelaar een verandering toepast. Een *build farm* kan deze praktijk ondersteunen. Dit is een verzameling machines die de software bouwt, typisch onder verschillende besturingssystemen en architecturen. Een build farm is niet alleen nuttig om software te testen, maar kan ook meteen *releases* van de software maken—kant en klare pakketten die door gebruikers geïnstalleerd kunnen worden.

Het beheer van zo’n build farm is vaak nogal tijdrovend, omdat elke gewenste verandering in de softwareomgeving (zoals een nieuwe versie van een compiler) op elke machine doorgevoerd moet worden. Dit kan echter geautomatiseerd worden met Nix, omdat het op een transparante wijze Nix-expressies voor meerdere platformtypes kan bouwen. Nix voorkomt ook dat componenten die niet gewijzigd zijn steeds opnieuw gebouwd worden. Tenslotte kunnen de in een op Nix gebaseerde build farm geproduceerde releases op betrouwbare wijze via Nix uitgerold worden.

Service deployment (hoofdstuk 9) Het uitrollen van draaiende netwerkdiensten (*services*) zoals webservers is een erg tijdrovend en meestal slecht beheerd proces. Veel acties—het installeren van software, aanpassen van configuratiebestanden, enz.—worden met de hand uitgevoerd en zijn daardoor niet automatisch reproduceerbaar. Hierdoor is het lastig om meerdere versies van een dienst (bijvoorbeeld test- en productieversies) naast elkaar te draaien, of om een *rollback* uit te voeren naar een eerdere toestand.

We kunnen dit probleem met Nix voor een groot gedeelte aanpakken door de relatief statische delen van een dienst (nl. de software en de configuratiebestanden) als een Nix-component te behandelen. Dit wil zeggen dat ze vanuit een Nix-expressie gebouwd worden en in de Nix store terecht komen. Alle voordelen van Nix voor deployment—zoals het gebruik van meerdere versies naast elkaar, reproduceerbaarheid, rollbacks, enz.—zijn op die manier onmiddelijk van toepassing op de uitrol van diensten. De dynamische toestand van een dienst (zoals databases) is echter niet onder Nix-beheer.

Build management (hoofdstuk 10) Het bouwen van software is een klassiek probleem dat typisch door tools zoals Make (1979) wordt opgelost. Zulke bouwtools hebben echter allerlei beperkingen die veel lijken op die van deploymentsystemen: ze kunnen niet garanderen dat afhankelijkheidsspecificaties volledig zijn, ze hebben slechte ondersteuning voor meerdere varianten van bouwacties, enz. Dit is niet verrassend, omdat bouwen en deployment goeddeels hetzelfde probleem zijn, met dien verstande dat bouwtools typisch betrekking hebben op “kleine” componenten (zoals individuele broncodebestanden), terwijl deploymentsystemen te maken hebben met “grote” componenten (zoals complete programma’s). Het is dus vrij eenvoudig om Nix te gebruiken als een Make-vervanger, simpelweg door Nix-expressies te schrijven die voornoemde “kleine” componenten bouwen.

Curriculum Vitae

Eelco Dolstra

18 augustus 1978
Geboren te Wageningen

1990–1996
Voorbereidend Wetenschappelijk Onderwijs aan de Christelijke Scholengemeenschap “Het Streek” te Ede
Diploma (gymnasium) behaald op 12 juni 1996

1996–2001
Studie Informatica aan de Universiteit Utrecht
Propedeutisch diploma behaald op 27 augustus 1997 (*cum laude*)
Doctoraal diploma behaald op 10 augustus 2001 (*cum laude*)

2001–2005
Assistent in Opleiding, Department of Information and Computing Sciences, Universiteit Utrecht

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping — A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation — CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata — A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad — A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing — Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima — A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java — Theory and Tool Support*. Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control — Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting — A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth — Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M. Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01