Language of ‘purely functional’ operating systems.

Camille Akmut

Introduction

Due to the multitude of terminologies brought on by the emergence of the “purely functional” approach in operating systems, such a document seemed warranted; Some terms are brand new, others should be familiar but have been re-purposed while others yet though established have been replaced.

Based on an extensive review of the existing literature, this language summary aims to be an entry point for researchers and others interested in this novel, and active field. Its vocabulary will hopefully not be a hindrance anymore to their various activities (theory or practice).

component	“*What we call a component typically corresponds to the ambiguous notion of a package in package management systems. (. . .) As far as Nix is concerned a component is just a set of files in a file system.*”	Dolstra 2006 : 19
store	“*Nix stores components in a component store, also called the Nix store.*”	Dolstra 2006 : 19
Nixpkgs	“*the Nix Packages collection, . . . a large set of Nix expressions for common and not-so-common software components.*” e.g. gcc, nix or firefox	Dolstra 2006 : 25; 167-68
derivation	“*Nix-speak for a component build action, which derives the component from its inputs.*”; “*build action that produces a single path in the Nix store*”	Dolstra 2006 : 27; Dolstra et al. 2008b : 4
`.drv`	“store derivation” (found in /nix/store and /gnu/store)	Dolstra 2006 : 39 ff.
hashes	“*an SHA-256 hash of all inputs used to build the component: - Sources - Libraries - Compilers - Build scripts - Build parameters . . .* ” e.g. /nix/store/jjp9pirx8b3nqs9k. . . -firefox	Dolstra 2006b; also Dolstra 2005 : 1

atomicity	"*Component upgrades in conventional systems are not atomic. That is, while a component is being overwritten with a newer version, the component is in an inconsistent state and may well not work correctly. This lack of atomicity extends beyond the level of individual components. When upgrading an entire system, for instance, it may be necessary to upgrade shared components such as shared libraries first. If they are not backwards compatible, then there will be a timing window in which components that use them fail to work properly.*"	Dolstra et al. 2004 : 80
atomic rollbacks	"*If a program does not work correctly it should be possible to roll back to an older version easily and atomically, complete with its configuration information.*" (the above handled mostly atomic upgrades)	Hemel 2006 : 9
Nix expressions*	"*Installation of components in the store is driven by Nix expressions. These are declarative specifications that describe all aspects of the construction of a component, i.e., obtaining the sources of the component, building it from those sources, the components on which it depends…*"	Dolstra et al. 2004 : 81

Nix expressions (language)	“*a simple functional language for computing with sets of attributes.*”; “*a dynamically typed, lazy, purely functional language.*”	Dolstra et al. 2004 : 81; Dolstra et al. 2010[2008] : 6
`stdenv`	“*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...*”	Dolstra 2006 : 26-27
`stdenv.mkDerivation`	“*mkDerivation is a function provided by stdenv that builds a component from a set of attributes. An attribute set is just a list of key/value pairs*”	Dolstra 2006 : 27
fixed-output derivation	“*derivations of which we know the output in advance (...) The rationale for fixed-output derivations is derivations such as those produced by the fetchurl function. (...) It sometimes happens that the URL of the file changes (...) If a fetchurl derivation followed the normal translation scheme, the output paths of the derivation and all derivations depending on it would change. (...) Fixed-output derivations solve this problem by allowing a derivation to state to Nix that its output will hash to a specific value.*”	Dolstra 2006 : 106 ff.

closures (component closures)	“*With precise dependency information, we can achieve the goal of complete deployment. The idea is to always deploy component closures: if we deploy a component, then we must also deploy its dependencies, their dependencies, and so on. That is, we must always deploy a set of components that is closed under the "depends on" relation. Since closures are self-contained, they are the units of complete software deployment. After all, if a set of components is not closed, it is not safe to deploy, since using them might cause other components to be referenced that are missing on the target system.*”; “*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.*”	Dolstra et al. 2004 : 84; Dolstra 2006 : 24

pointers (dangling)**	"*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.*"	Dolstra et al. 2004b
garbage collection	"*To ensure that no dangling pointers can occur, Nix does not provide an operation to delete components. Rather, paths are deleted from the store when they become garbage, i.e., when they are no longer reachable from outside the store.*"	Dolstra et al. 2004b, '7.6. Garbage collection'
imperative model	"*Most package management tools can be viewed as having an imperative model. That is, deployment actions performed by these tools are stateful; they destructively update files on the system. For instance, most Unix package managers, such as the Red Hat Package Manager (RPM), Debian's apt and Gentoo's Portage...*"	Dolstra et al. 2010[2008] : 3

store expressions (`.store`)	"*Nix expressions are translated into the much simpler language of store expressions, just as 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.*"	Dolstra et al. 2004 : 85
scanning (approach)	"*The hash scanning approach gives us all runtime dependencies of a component*"	Dolstra 2006 : 24
source deployment model	"*This is the model used by source-based deployment systems such as the FreeBSD Ports Collection and Gentoo Linux.*"	Dolstra 2005b (see also Dolstra 2006 : 11 ff.; Hemel 2006 : 12 ff.)
transparent source/binary deployment model***	"*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*	Dolstra 2006 : 45
substitute	"*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.*"	Dolstra 2006 : 45 (see also Dolstra et al. 2004 : 85-87)

graft	"*when a package is changed, every package that depends on it must be rebuilt. This can significantly slow down the deployment of fixes in core packages (. . .) To address this, Guix implements grafts, a mechanism that allows for fast deployment of critical updates without the costs associated with a whole-distribution rebuild.*"	Guix Manual****
continuous integration		Dolstra 2008b : 1 (see also ch. 8 of Dolstra 2006)
build farm		Dolstra et al. 2004 : 89
"ad hoc" (package management)	As found in the Nix documentation*****, meaning presumably "in addition" to the declarative style (of the general configuration file).	
channel	~ a repository for software	

Notes :

*"Nix expressions" is routinely used in two different senses : one the language, two the (package) declarations or definitions written in that language.

**One attempt to paraphrase this follows : in this context, pointers are references to dependencies of a program or "component"; and, they are dangling if the dependency is not available for whatever reason e.g. their installation failed, they were removed -possibly by another program-, etc.. The common context is memory management in languages like C (see a C book).

Dolstra himself writes : "*In Chapter 3 the dependency problem is cast in terms of memory management in programming languages*" (Dolstra 2006 : 24)

*** "*Guix supports transparent source/binary deployment, which means that it can either build things locally, or download pre-built items from a server, or both. We call these pre-built items substitutes—they are substitutes for local build results. In many cases, downloading a substitute is much faster than building things locally.*" (https://guix.gnu.org/manual/en/html_node/Substitutes.html)

**** https://guix.gnu.org/manual/en/html_node/Security-Updates.html. Why is this especially an issue with core packages? Because, more software depends on them (thus more rebuilds needed).

Similar is found in the Nix literature : "*For instance, if we were to change (...) Glibc (...) - a component on which almost all other components depend - massive rebuilds will ensue.*" (Dolstra 2006 : 106)

*****e.g. https://nixos.org/manual/nixos/stable/#sec-package-management

BIBLIOGRAPHY

- Dolstra, Eelco. 2006. *The Purely Functional Software Deployment Model.*

`"This thesis is about getting computer programs from one machine to another - and having them still work when they get there."`

- Dolstra, Eelco et al.. 2010[2008]. "NixOS: A Purely Functional Linux Distribution". *Journal of Functional Programming* 20(5-6). Previous version : 2008 (significantly shorter)

`"Current operating systems are managed in an imperative way. With this we mean that configuration management actions such as upgrading software packages, making changes to system options, or adding additional system services are done in a stateful way"`

- Dolstra, Eelco et al.. 2004. "Nix: A Safe and Policy-Free System for Software Deployment". LISA 18 (USENIX).

`"we present Nix, a deployment system that addresses these issues through a simple technique of using cryptographic hashes to compute unique paths for component instances."`

- Dolstra, Eelco et al.. 2004b. "Imposing a Memory Management Discipline on Software Deployment".

`"As any computer user knows, software installation is a fragile process that fails surprisingly often for seemingly trivial reasons..."`

- Dolstra, Eelco. 2005. "Efficient Upgrading in a Purely Functional Component Deployment Model".

`"The Nix deployment system enables side-by-side deployment of different versions and variants of components, ... safe upgrades, and ... uninstalls through garbage collection."`

- Dolstra, Eelco. 2005b. "Secure Sharing Between Untrusted Users in a Transparent Source/Binary Deployment Model".

- Dolstra, Eelco. 2006b. "Software deployment with Nix". Previous versions : 2005c, 2004c 2004d

- Dolstra, Eelco et al.. 2008b. "Hydra: A Declarative Approach to Continuous Integration". (The provided year of publication, unclear, is based on citations.)

- Hemel, Armijn. 2006. *NixOS: the Nix based operating system.*

`"[show] how the Nix package management system can be applied to manage a whole Linux distribution."`