Taxonomy of Package Management in Programming Languages and Operating Systems

Hisham Muhammad
Kong Inc.
hisham@konghq.com

Lucas C. Villa Real
IBM Research - Sao Paulo, Brazil
lucasvr@br.ibm.com

Michael Homer
Victoria University of Wellington - Wellington, New Zealand
mwh@ecs.vuw.ac.nz

Abstract

Package management is instrumental for programming languages and operating systems, and yet it is neglected by both areas as an implementation detail. For this reason, it lacks the same kind of conceptual organization: we lack terminology to classify them or to reason about their design trade-offs. In this paper, we share our experience in both OS and language-specific package manager development, categorizing families of package managers and discussing their design implications beyond particular implementations. We also identify possibilities in the still largely unexplored area of package manager interoperability.

Keywords package management, operating systems, module systems, filesystem hierarchy

1 Introduction

Package managers are programs that map relations between files and packages (which correspond to sets of files), and between packages (dependencies), allowing users to perform maintenance of their systems in terms of packages rather than at the level of individual files. Package management is an area that lies somewhere in the border between programming languages and operating systems: packaging is a step that sits after a language's build process, and before an operating system's component installation. For this reason, it seems to be overlooked by both fields as an implementation issue. In the meantime, package management keeps growing in complexity. New languages, new deployment models, and new portability requirements all give rise to new package management systems. Further, this is not simply a matter of competing implementations: modern complex environments often require several package managers to be used in tandem.

PLOS 2019, October 29, 2019, Huntsville, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For example, when writing JavaScript web applications on a Mac environment, a developer may require using Bower [1], a package manager for client-side JavaScript components. Bower is installed using `npm` [2], a package manager for `node.js` [3], a JavaScript environment. On a Mac system, the typical way to install command-line tools such as `npm` is via either Homebrew [4] or MacPorts [5], the two most popular general-purpose package managers for macOS. This is not a deliberately contrived example; it is the regular way to install development modules for a popular language in a modern platform.

The combinations of package managers change as we move to a different operating system or use a different language. Learning one's way through a new language or system, nowadays, includes learning one or more packaging environments. As a developer of modules, this includes not only using package managers but also learning to deploy code using them, which includes syntaxes for package specification formats, dependency and versioning rules and deployment conventions. Simply ignoring these environments and managing modules and dependencies by hand is tempting, but the complexity of heterogeneous environments and keeping track of dependency updates can become burdensome — all these package managers were created to solve practical problems which the developer would have to otherwise directly handle, after all. Another alternative that is often proposed, especially by users of operating systems that feature a system-provided package manager (as is the case of most Linux distributions), is to avoid using multiple package managers and use a single general-purpose package manager. This is, of course, as much of a solution as trying to make everyone agree on a single programming language — one of many analogies between package management and programming languages. The result is that the ecosystem is not getting any simpler, and at first glance it seems that package management is indeed a largely unsolved problem.

However, maybe the statement "package management is an unsolved problem" simply does not make sense, and is akin to saying that "programming languages are an unsolved problem". In the programming languages world we accept that the multitude of languages is a given. Beyond that, we understand that there are families of languages with different paradigms, with well-known tradeoffs. We also accept that there is room for domain-specific languages (DSLs) and for

general-purpose languages. Most importantly, we know how to set boundaries for each language and how to make DSLs and general-purpose languages interact. Most existing package management systems, however, are still oblivious to the fact that they exist as part of a larger ecosystem, with parts of it handled by other package managers.

In this paper we draw on our own unique combination of experiences on all sides of this topic: developing a system-wide package manager for a Linux distribution [6, 7], creating a language-specific package manager [8], and integrating system and language-specific package managers [7], as well as being simply developers and end users of other software. By building a taxonomy for package management and sharing our experiences with package management development in both the programming language and operating system spaces, we aim in this paper to frame the design choices, layers, and trade-offs attendant to package managers for developers, maintainers, and users.

2 Paradigms of package management: filesystem-oriented vs. database-oriented

It is a typical didactic device to organize the landscape of programming languages into paradigms, such as imperative, functional, object-oriented, and so on. The paradigms a language is categorized into inform users about particular design choices, and with these choices come design trade-offs. In the world of package managers, we can also identify general paradigms, by looking at their core concepts and design trade-offs. Package managers map between files and packages, and between packages and their dependencies, allowing users to perform maintenance of their systems in terms of packages rather than at the level of individual files. The central design choice in a package manager, therefore, is how to perform those mappings.

There are two approaches on how to map files to packages: the mapping can be based on the hierarchical directory structure where the files reside, or can be separate. As this choice embodies a series of trade-offs and is the single decision that affects the design and implementation of a package manager the most, we identify these as two paradigms of package management. When the mapping is internal to the file hierarchy structure, we say that package management is *filesystem-oriented*[1]. When it is external to the hierarchy of files being managed, the mapping needs to be stored elsewhere. We say in these cases that management is *database-oriented*. Most package managers for Linux distributions, such as RPM and dpkg/APT, are database-oriented. Filesystem-oriented package management is more often seen in language-specific package managers, but as we will see in Section 5.1, it can also be performed system-wide.

[1] Being database-oriented does not imply an opaque, binary database format. Various database-oriented package managers store their file manifests in plain text files

The directory structure used by `pip`, the package manager for Python modules, is representative of the database-oriented style. Note that all installed modules are stored under `/usr/lib/python3.7/site-packages/`: this structure was already the default subdirectory for locally-installed modules prior to the introduction of `pip`. Database-oriented designs are often chosen when the package manager needs to accomodate a pre-existing directory structure. If Python packages have modules with the same name, clashes may occur. In a filesystem-oriented design, such as for example that of RubyGems, this problem would not happen. Each package has its own subtree under a versioned directory, and the `rubygems.rb` module, part of the default installation of Ruby, takes care of finding the appropriate files when modules are loaded with the `require` function. Figure 1 lists more examples of package managers and their classification.

The major trade-off between the filesystem-oriented and the database-oriented approaches is whether applications should be aware of the file structure defined by the manager or whether the manager should adapt to the file structure defined by applications. This affects how the manager tracks the mapping of files and how applications are configured to find their resource files.

In filesystem-oriented managers the mapping of files to packages is simple. File conflicts are naturally avoided by storing files of different packages in separate subtrees. Versioning conflicts between variants of the same package can also be handled via the tree structure. The structure also becomes more transparent to users, which can simplify their experience. The run-time lookup of files by applications, however, can be complicated, if they are oblivious to the structure defined by the package manager. Applications must either agree beforehand to this structure (which might be an option in domain-specific environments), or the package manager has to do extra work to configure them to use the structure, such as setting configuration options or environment variables, or in the worst case, patching them.

Conversely, in database-oriented managers the mapping of files to packages is more complicated. Applications may install files wherever they please, and the package manager needs to keep track. This includes handling potential conflicts if two packages want to use the same pathname. Database-oriented systems will usually report on these conflicts and forbid them. It is up to the integrator (such as a distribution developer) who is building packages to resolve the conflict somehow. Also, the package manager needs to verify that the database and the contents of the filesystem remain in sync, which is trivial in the filesystem-oriented approach. The run-time lookup of files on database-oriented systems, on its turn, is greatly simplified. In most cases it will be a non-issue, since each file is in the location the application expected it to be in the first place. However, it does become an issue when the file has been relocated by the integrator who built the package, perhaps for solving conflicts.

Filesystem-oriented managers also present their own set of challenges, as the description of packages as set of files does not present a full picture. Packages, especially in system-wide installations, often need to perform global changes to the system, such as adding users and setting environment variables. Some applications also include database-oriented portions which are assumed to be updated by installation scripts, such as refreshing global caches. Non-relocatable packages often assume hardcoded default paths in which resource files are expected to be found; if the package manager employs a different organization, it needs to reconfigure applications to make sure the required files are found. One common solution is to use environment variables, since applications often support setting custom paths via variables in addition to the system-wide defaults. Most applications can be installed in custom locations, with the installation prefix being adjustable at compile time. The `/opt` directory is a traditional location for filesystem-oriented organization of additional packages. Core system services are often harder to relocate.

To use the filesystem or a database is a frequent design dilemma beyond package management, especially on Unix systems, where "everything is a file" is a long-standing tradition. Database-oriented solutions often are considered un-Unix-like (GConf, for instance, raises comparisons to the Windows Registry [9]). It is remarkable that, in spite of the Unix philosophy, most Linux package managers are primarily database-oriented.

3 Language-specific vs. language-agnostic package managers

In the world of programming languages, there is a distinction between DSLs and general purpose languages. Categorizing languages in one camp or another is not always easy, but a working definition is that domain-specific languages are those designed with a specific application domain in mind, and general purpose languages are the complementary set, that is, those languages designed not with a particular domain in mind, but rather focusing on general areas such as "systems programming".

While we tend to see DSLs as smaller languages than their general-purpose counterparts (and in fact early literature used to term them "little languages" [10]), what defines a language as being a DSL is the *inclusion* of features tailored for a domain. This means that a domain-specific language may end up including all features normally understood as those defining a general purpose language. MATLAB [11], for instance, is a complete programming language, but its wealth of features for numerical computing it is often regarded as being domain-specific [12, 13].

In the world of package management, there is also a distinction between domain-specific and general purpose systems, but it is better defined. Language-specific managers are designed to be used in a particular *language ecosystem*. This ecosystem usually focuses around a single language (hence the name "language-specific"), but that is not necessarily the case: environments such as .NET and the JVM make this evident, but other languages also grow into families: for example, npm supports JavaScript, CoffeeScript, TypeScript and others. Besides, these VM-based ecosystems usually support loading native extensions, and therefore they must also support building and integrating libraries usually written in C or C++ (or, in the case of RubyGems with JRuby, Java). A language-specific package manager, therefore, is almost never specific to code written in a single language. Like domain-specific programming languages which are not necessarily much smaller than their general-purpose counterparts, the more sophisticated language-specific package managers are in effect much broader package managers with specific support for an ecosystem added. They need to build and deploy executables, native libraries and resource files written in different languages, keep track of installed files, check dependencies, perform network operations and manage remote repositories. Some of these tasks can be simplified due to ecosystem-specific assumptions, but many are equivalent in complexity to the tasks of a system-wide package manager.

This leads us to question why should we have language-specific managers at all, if they replicate so much of the work done by general-purpose package managers. Two arguments in defense of language-specific managers are scalability and portability. If we compare the number of packages provided by a typical Linux distribution versus the number of modules available in mature module repositories from scripting languages, it becomes clear that the approach of converting everything into native packages is untenable: for example, while the repository for the Debian Linux distribution features over 59,000 packages in total, the Maven Central repository for Java alone contains over 290,000 packages, with the advantage that the repository is portable to various platforms, some of which lack a built-in universal package manager (Microsoft Windows being a notable case). Still, this kind of effort duplication does happen: the Debian repository contains 1,196 Ruby packages; this is a far cry from the over 150,000 modules in the RubyGems repository.

Figure 2 contrasts language-specific and language-agnostic package managers, through a few examples. Language-specific package managers tend to be highly portable, even if the modules in their repositories are not. For example, while most packages for NuGet are Windows-specific, the manager itself has been ported to Unix systems via Mono; packages that do not depend on Windows APIs can be shared by various platforms. Language-agnostic managers are generally system-specific, and may present some degree of portability to other similar OSes. Note that the extent of portability of all language-agnostic managers in Figure 2 is limited to specific Unix variants. Those managers support packaging programs written in any language and for that reason do not expect particular file formats or subdirectory layouts. Language-specific

	Filesystem-oriented	Database-oriented
Language-agnostic	Homebrew (macOS), GNU Stow, Nix, Encap, PBI 8 (PC-BSD), GoboLinux	RPM (RedHat/Fedora/etc.), dpkg/apt (Debian/Ubuntu/etc.) PBI 9 (PC-BSD), Pacman (ArchLinux)
Language-specific	npm (server-side JavaScript), Bower (client-side JavaScript) RubyGems (Ruby), Cargo (Rust), LuaRocks 1.x (Lua)	Cabal (Haskell), pip (Python), LuaRocks 2.x (Lua)

Figure 1. A package manager taxonomy, with representative examples

	Language-specific managers			
Package managers	**npm**	**RubyGems**	**NuGet**	**LuaRocks**
Portability	OS-independent (all Unix, Windows)			
Installs code written in	JS family, C/C++	Ruby, C/C++, JVM family	any .NET, C++	Lua family, C/C++
Files managed	JS scripts, JS modules	Ruby scripts, Ruby modules	.NET and native packages	Lua scripts, Lua modules
Supports per-user install	yes			

	Language-agnostic managers			
Package managers	**Nix**	**Homebrew**	**RPM**	**GoboLinux**
Portability	Linux/macOS	macOS/Linux	Linux/AIX	Linux/Cygwin/OSX
Installs code written in	any language			
Files managed	all kinds			
Supports per-user install	yes	no*	no	yes

* different installation prefixes are supported but `/usr/local` is strongly recommended.

Figure 2. Contrasting language-specific and language-agnostic package managers

managers make more assumptions in that regard, and also support customizing the installation directory prefix, which is a necessity for running as a non-privileged user. Some system-wide managers, like Nix and GoboLinux support per-user installations, but that often requires patching packages for removing hardcoded pathnames. Homebrew supports this feature as a tool, but their packages are not adapted for that, so per-user installations are discouraged.

4 Integration between languages vs. integration between package managers

Programming languages, both general-purpose and domain-specific, frequently have points of integration between each other, in the form of foreign function interfaces (FFIs). Code written in one language can frequently call into code written in another language, sometimes with some adapter code in between. Domain-specific languages are in fact frequently embedded in general-purpose languages and in programs. Programming languages can also integrate between each other through common calling and linking conventions.

The same is not true of general-purpose and domain-specific package managers. Integration between two package managers is almost unheard of, even when they may be found on the same system. Instead, a subset of packages distributed through a domain-specific package manager are repackaged in the format of the general-purpose system. These packages are fully integrated with the broader system and fully detached from the domain-specific manager. Packages that were not repackaged are still available by using the domain-specific system, but others are available twice, potentially in different versions and with different configurations. Debian experimented with a `rubygems-integration` package that provided a limited connection between APT and RubyGems, allowing Debian packages of individual RubyGems to satisfy dependencies in the gem tool, but encountered nontrivial complications in doing so [14, 15]. Debian has not yet pursued even this level of integration for other widely-used domain-specific package managers, and the integration it has for RubyGems is ad-hoc and highly specialized. In Section 5.3 we discuss our attempt at deeper integration in GoboLinux, but we are aware of no other such integrations beyond what Debian performs.

A weaker form of one-way integration between package managers occurs when the system-wide manager uses the language-specific package manager merely as a build system. An example is the use of LuaRocks by Buildroot [16]. Buildroot is a system for compiling full-system images for embedded environments, which has its own package specification format. It uses LuaRocks as a build tool: the Buildroot specification scripts launch LuaRocks to generate Lua modules and then collect and integrate them to the system.

5 Experiences with package management

In the following section, we share some of our experiences, each case study dealing with one the three aspects of package management design outlined above.

5.1 GoboLinux

GoboLinux [6] is a Linux distribution based on the concept of installing each package in a separate installation prefix. Introduced in 2002, it was the first Linux distribution to be entirely based on a filesystem-oriented approach to package management[2]. Each program is installed under its own versioned directory, such as `/Programs/Bash/4.3.28` and `/Programs/GTK+/3.16.0`. This direct mapping of the package structure to the directory layout allows one to inspect the system using standard Unix commands. For example, to get a list of installed packages, one only needs to issue `ls /Programs`.

As well as the individual program trees, a tree of symbolic links called `/System/Index` collects references to the files from every program in the system. A single directory contains symlinks matching the structure of the "`lib`" directory of every program, paralleling the contents of `/usr/lib` in a conventional layout. Figure 3 illustrates this structure. In this way only a single entry in PATH is needed to find every executable and libraries can be loaded using the ordinary linker mechanisms without further configuration. An additional layer of fixed symlinks provides backwards compatibility with the conventional Filesystem Hierarchy Standard[17].

In its original design, packages compiled for GoboLinux targeted their versioned directory during compilation. That made them aware of the modified filesystem structure, and often required configuration contortions and also workarounds to handle the management of files that were designed to be shared between packages. In a later revision, GoboLinux switched to compilation targeting `/usr` and installation to the per-program location. Through this structure, even though packages are organized in self-contained directories under `/Programs`, applications find their files through the traditional Unix hierarchy, as `/usr` is a symbolic link to `/System/Index`.

5.2 LuaRocks

LuaRocks [8] is a package manager for the Lua ecosystem. It was developed building on our previous experience writing package management tools for GoboLinux and adapting it to the realities of a language-specific manager.

The design changes that LuaRocks underwent were due to lessons learned on the specificities of language-specific package management. The original design of LuaRocks was

```
/System/
├ /Index/
│ ├ /sbin -> bin
│ ├ /bin/
│ ├ /include/
│ ├ /lib/
│ │ ├ audit  -> /Programs/Glibc/2.18/lib/audit
│ │ │ awk    -> /Programs/Awk/4.1.0/lib/awk
│ │ │ cairo  -> /Programs/Cairo/1.12.16/lib/cairo
│ │ ├ /cmake/
│ │ │ ├ Evas -> /Programs/EFL/1.11.0/lib/cmake/Evas
│ │ │ ├ qjson-> /Programs/QJSON/0.8.1/lib/cmake/qjson
│ │ │ │ ...
│ │ │ ...
```

(a) The filesystem is indexed with the use of directories and symbolic links.

```
/Programs/
├ /ALSA-Lib/
├ /ALSA-Utils/
├ /BeeCrypt/
│ ├ /4.1.2/
│ ├ /4.2.1/
│ │ ├ /include/
│ │ ├ /lib/
│ │ └ /Resources/
│ │   ├ Architecture
│ │   ├ Dependencies
│ │   ├ Description
│ │   ├ FileHash
│ │   ├ MetaData
│ │   └ UseFlags
│ ├ Current -> 4.2.1
│ └ /Settings/
│   └ beecrypt.conf
│ ...
```

(b) Versioned directory tree.

Figure 3. GoboLinux file system hierarchy

filesystem-oriented, like GoboLinux. LuaRocks included then a custom wrapper for Lua's `require()` function, much like RubyGems. However, many Lua users perceived the wrapper as tampering with a standard library function, and disliked having to perform an initial setup in their scripts for using modules installed via LuaRocks [3]. For LuaRocks 2.0, the design was changed to be database-oriented, so that Lua modules could be installed into a typical Unix-like layout that matched the default configuration of the Lua interpreter's package loader. With all packages installed under a single directory, a database had to be put in place matching files to packages. Supporting multiple versions of the same package installed at the same time is still possible, but requires the now-optional custom package loader, which produces versioned filenames when the dependency graph requires an old version of a module. This language-specific runtime adjustment allows avoiding the issue with filename conflicts, so common with database-oriented designs — a luxury that operating system package managers cannot afford so easily.

Having a high-level declarative specification allowed us to make such radical changes to the installation layout easily.

[2]While GoboLinux remained a research distribution with a niche community, its design proved influential, as its filesystem-oriented approach was used as a basis for the design of Homebrew, the most popular package manager in macOS today, as noted in its original documentation: https://github.com/Homebrew/legacy-homebrew/tree/89283db693e9380ccc2e4abc4fa0ad14b4790202

[3]Since Ruby 1.9, the interpreter preloads the `rubygems` module automatically; in prior versions uses had to add `require 'rubygems'` explicitly. This was never an option for Lua due to the language's minimalistic design, rendering LuaRocks as a strictly optional component.

Since LuaRocks produces relocatable packages, it does not provide to specification files (rockspecs) any knowledge of the final directory structure. This allowed all existing rockspecs to be used in the new directory layout without any changes. This level of information hiding was only possible because we were dealing with a language-specific manager, where we knew what was in the files (Lua source code and binary dynamic libraries) and how they would be used (as command-line scripts or loaded by Lua through its package loader system).

5.3 GoboLinux Aliens

In GoboLinux we researched the idea of building a foreign function interface (FFI) of sorts into our general-purpose package manager, which we called Aliens[7]. Aliens provides an API to write shims that connect the general-purpose system package manager with domain-specific package managers.

With Aliens, packages in the general-purpose manager may express a dependency on a package provided by a supported domain-specific manager: for example, a package that requires the Perl `XML::Parser` module, available from CPAN, can express a dependency "`CPAN:XML::Parser >= 2.34`". The Aliens system directs such a dependency to a translating shim, which uses the CPAN tool to confirm whether it is satisfied, to install the package (and its dependencies) if required, and to upgrade it, communicating any necessary information back to the general-purpose manager. The shim can then make symbolic links for any binaries or native libraries that have been installed. Any package in one of the supported domain-specific managers is automatically available in this way, without creating wrapper packages.

The domain-specific managers themselves are not modified in this process. Each system is given complete control of a directory tree, and the relevant languages configured to search in that tree. This protects against changes in the functioning of the third-party systems, and allows users to access them directly as well. A drawback, however, is that the domain-specific managers do not have reciprocal access to the wider system: installing a RubyGem that depends on a native library will not innately result in the native dependency being satisfied. The cross-platform nature of these systems makes even specifying such information in a machine-readable way difficult, although some, notably LuaRocks, make the attempt.

Not all domain-specific package managers lend themselves to this integration. Some are resistant to placing their files within a restricted directory tree, preferring to install into the global filesystem hierarchy where they may interfere with each other and the system, while others do not mechanize well. This limited coverage is an additional drawback of the Aliens approach, but one that is limited to failing to solve an existing problem, rather than creating a new one. As with programming languages, a consensus implementation platform would inevitably be simpler, but social and technical factors make it impractical. FFIs, and Aliens, attempt to bridge the gap, with reasonable success.

6 Conclusion

Package management is an area that is notably neglected in academic studies, but is one of practical impact in the design of modern operating systems and module systems for programming languages. In the realm of programming languages, we have useful ways to categorize languages. Package management even lacks common terminology, and each new system faces the same design issues time and again, even as we move to containers and orchestration systems [18].

As we categorize package management systems, we conclude that filesystem-oriented designs are preferable as they tend to be less susceptible to conflicts, but they require some level of intervention to enable files to be found at run time. We observe that this control exists when language-specific package managers are bundled with the language environment, as is the case with npm for node.js and RubyGems for Ruby. These managers were free to adopt filesystem-oriented designs since they adjusted their module loaders accordingly. The other way to exert this control over run-time lookup is to employ a system-wide lower-level solution as we did in GoboLinux with the `/System/Index` tree of symbolic links or in the stowfs filesystem virtualization proposed for GNU Hurd [19]. Database-oriented designs, on the other hand, are more generally applicable, but are more opaque to their users and are more prone to package conflict and file-to-package synchronization issues. For these reasons, we advocate filesystem-oriented systems in general, but we also recognize that there are situations where a database-oriented solution works best to preserve compatibility with the ecosystem at play, as was the case with LuaRocks.

Our classification of package managers as language-agnostic and language-specific highlighted the complementary qualities of these two classes of managers. The existence of language-specific package managers distributes integration efforts, as upstream module developers are often the package integrators themselves. This allows scaling repositories way beyond what is possible through the work of OS distribution maintainers, but also generates some tension between the language and distribution communities as perceived duplicate work and incompatibilites happen. Through our experience in both ends of the spectrum of package management — from low-level distribution management in GoboLinux to high-level language modules in LuaRocks — we observed a necessity for these different levels of system organization to recognize each other and aim for cooperation. Package managers do not exist on their own, but are part of an ecosystem in which other package managers often take part. We shared our experience in progressing on this direction with the GoboLinux Aliens project, and we plan to further pursue FFI-style package management interoperability.

REFERENCES

[1] Bower - a package manager for the web. http://bower.io, accessed August 9, 2019 2019. URL http://bower.io.

[2] npm - Node Package Manager. http://npmjs.com, accessed August 9, 2019 2019. URL http://bower.io.

[3] node.js. https://nodejs.org/, accessed August 9, 2019 2019.

[4] Homebrew - the missing package manager for macOS (or Linux). http://brew.sh, accessed August 9, 2019 2019.

[5] The MacPorts project. http://www.macports.org, accessed August 9, 2019 2019.

[6] Muhammad H, Detsch A. An alternative for the Unix directory structure. *III Workshop Software Livre*, Porto Alegre, Brazil, 2002.

[7] Homer M, Muhammad H, Karlsson J. An updated directory structure for Unix. *linux.conf.au 2010*, Wellington, New Zealand, 2010.

[8] Muhammad H, Mascarenhas F, Ierusalimschy R. LuaRocks - a declarative and extensible package management system for Lua. *Lecture Notes in Computer Science* 2013; **8129**:16–30, doi:10.1007/978-3-642-40922-6_2.

[9] Wallen J. GConf makes Linux administration a little more like Windows. TechRepublic Feb 2003. URL http://www.techrepublic.com/article/gconf-makes-linux-administration-a-little-more-like-windows/, http://is.gd/V5fBLB, accessed August 9, 2019.

[10] Bentley J. Programming pearls: Little languages. *Commun. ACM* Aug 1986; **29**(8):711–721, doi:10.1145/6424.315691. URL http://doi.acm.org/10.1145/6424.315691.

[11] MATLAB. https://www.mathworks.com/products/matlab.html, accessed August 09, 2019 2019. URL https://www.mathworks.com/products/matlab.html.

[12] Gill A. Domain-specific languages and code synthesis using haskell. *Queue* Apr 2014; **12**(4):30:30–30:43, doi:10.1145/2611429.2617811. URL http://doi.acm.org/10.1145/2611429.2617811.

[13] Fowler M. Language workbenches: The killer-app for domain specific languages? jun 2005. URL http://martinfowler.com/articles/languageWorkbench.html, http://martinfowler.com/articles/languageWorkbench.html, accessed August 9, 2019.

[14] Nussbaum L. Re: Ruby packaging in Wheezy: gem2deb, new policy, etc. debian-ruby mailing list post, 18 January 2011. Available from https://lists.debian.org/debian-ruby/2011/01/msg00050.html, accessed August 9, 2019 2011. Mailing list post.

[15] Debian Ruby Team. Teams/Ruby/Packaging - Debian wiki. https://wiki.debian.org/Teams/Ruby/Packaging, accessed August 9, 2019.

[16] Buildroot. http://www.buildroot.org, accessed August 9, 2019 Apr 2019. URL http://www.buildroot.org/.

[17] Russel R, Quinlan D, Yeoh C ((eds.)). *Filesystem Hierarchy Standard*. 2004. Available from http://www.pathname.com/fhs/pub/fhs-2.3.pdf, accessed August 9, 2019.

[18] Helm - the package manager for Kubernetes. http://helm.sh, accessed August 09, 2019 2019. URL http://helm.sh.

[19] Stow 2.2.0. http://www.gnu.org/software/stow/manual/stow.pdf, accessed August 9, 2019 2012.