Understanding package management by analogy to programming languages

1 Introduction

Package management is an area that lies somewhere in the border between programming languages, operating systems and system administration. For this reason, it seems to be overlooked by all three 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.

For example, when writing JavaScript web applications on a Mac environment, a developer may require using Bower [2], a package manager for client-side JavaScript components. Bower is installed using `npm` [6], a package manager for `node.js` [5], a JavaScript environment. On a Mac system, the typical way to install command-line tools such as `npm` is via either Homebrew [3] or MacPorts [4], the two most popular general-purpose package managers for Mac OSX. This is not a deliberately contrived example; it is the regular way to install development modules for a popular language in a modern platform.

In [8] we have another example of a typical software stack, where a deployment and management scenario for Ruby on Rails applications is described combining a number of tools. It uses Vagrant [11] for virtual machine management; Puppet [7] for editing system configuration files and driving the system-wide package manager on servers; Capistrano for deploying the Ruby on Rails application, including installing Ruby scripts and migrating database tables, driving RubyGems, the language-specific package manager for Ruby modules (with Bundler to mitigate module version conflicts); and RVM for managing conflicting versions of Ruby itself. It is interesting to note the number of different tools being used on top of each other to manage containment and compatibility issues on various layers; and again, this is a typical, realistic scenario.

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 pratical 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 as a solution as trying to make everyone agree on a single programming language, and this is the first of various analogies between package management and programming languages that we will make throughout this paper. 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-

	Filesystem-oriented	Database-oriented
Language-agnostic	GoboLinux Homebrew (Mac OS X) Nix Windows Installer	RPM (RedHat/Fedora/etc.) dpkg/apt (Debian/Ubuntu/etc.) Pacman (ArchLinux)
Language-speci c	npm (JavaScript) Bower (JavaScript) LuaRocks 1.x (Lua)	Cabal (Haskell) LuaRocks 2.x (Lua)

Figure 1: A package manager taxonomy

purpose languages. Most importantly, we know how to set boundaries for each language and how to make DSLs and general-purpose languages interact.

We argue that all of these observations can be made with regard to package management as well. In this paper, we discuss how these observations map from the world of programming languages to that of package managers. 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. By discussing how programming languages deal with these issues, we point to directions on how package manegers could follow their example, drawing on our experiences developing both a system-wide package manager [13, 12] and a language-speci c package manager [14].

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

An analogy between programming language paradigms and package management paradigms, generally advocating lesystem-oriented models.

- lesystem paradigm vs. database paradigm
- trade-offs: how runtime lookup of les happens
  - databases involved in runtime lookup: gtk icon cache, Haskell db updates, etc.
  - pkg-con g, how does it t?
  - lesystem paradigm and runtime lookup: environment variables, index directories with symlinks, symlinks in lib (libfoo.so, libfoo.so.1, etc.)
- versioning in db vs lesystem. libfoo.so vs. libfoo.so.1, versioning in gobo, /usr/include/python2.7/ etc.
- changes in lesystems to better support the lesystem paradigm? (union mounts, etc)
- related work outside of “classic” Unix: appdirs, Windows, Plan 9.
- lesystem paradigm in Unix:
  - /opt: early example of lesystem-oriented management
  - parts of the FS hierarchy which use the lesystem paradigm: /usr/share/appname
  - parts which don't: /usr/share/icons
    - ∗ effect this had on share/ in Gobo: launching helper indexing such as gtk-update-icon-cache upon each installation
- Being database-oriented does not imply an opaque, binary database format. LuaRocks 2.x uses a set of Lua tables (in `.lua` source format) as its database. ArchLinux keeps its database as a tree under `/var/lib/pacman/local`, with one subdirectory per package, containing text les that list which les belong to which package, as well as other metadata.
  - Make les with install and uninstall rules in ports: early example of database paradigm?
- other instances of lesystem vs database paradigm:
  - sysv and BSD init versus systemd
  - Mac OSX equivalent
  - /etc text-based con g les vs. gconf
  - Windows registry.
- Explain Figure 1.

2.1 GoboLinux

- Our extensive use of shell scripting for system management was a sort of attempt to show that the system could work with “just a few scripts”.
  - core ideas work (the system is still driven via symlinks) but management scripts became complex over time
- the build system, the tooling for generating packages, is the central piece of a package management system.
  - not clear at rst, but looking at package management systems they all integrate the build process. Why?

- Discuss the challenges of GoboLinux through the prism of its build system (?)
  - Tried to make it easy for users to build packages.
  - Took inspiration from systems such as Gentoo, which was itself inspired from BSD Ports.
  - aim was to make the simple cases super-simple (like a 3-line script) and the complex cases possible (leveraging the generality of shell scripts)
  - This worked up to a point. Eventually, started requiring more and more metadata, even for the so-called simple cases.
  - Further, this metadata had to be integrated with the deployed system.
  - The ArchLinux build system seems to be a modern-day successor of this style of build system in the Linux space.

The mostly declarative style of GoboLinux recipes proved to be a success among users and gave us greater freedom when modifying the build process. We made major changes to the system's directory layout between releases 014 and 015 of the distribution, and the use of high-level descriptions and our cautious avoidance of hardcoded paths allowed us to reuse the majority of recipes with no changes.

2.2 Other lesystem-oriented approaches

- Short history of alternative approaches:
  - earliest related work: GNU stow, encap. (Our paper from Workshop em Software Livre mentions that, but we didn't really know about them when making Gobo)
  - Zero Install (still active), Autopackage (dormant, “packages must be relocatable”),
- The Nix project has been around almost as long as GoboLinux. NixOS is nowadays a serious contender in the world of server-oriented operating systems.
  - virtual machines: minimalism is making its way back in OS layout design. (There have always been minimalistic Linux distributions, back from the “rescue” distros such as Damn Small Linux and tomsrtbt (which would t in a oppy!).
    - CoreOS, based on Gentoo, is the current representative of the minimalistic server-oriented distro world.
    - Ubuntu Core was recently announced, and this might be a beginning of a general trend of “core” OSes.
      - Ubuntu Core “snappy” packages strongly resemble GoboLinux!
  - Homebrew, a package manager for Mac OS X, is a successful realization of this idea. One of its original design criteria was to do package management “the GoboLinux way” [in the git history of homebrew's README.md we nd them citing gobolinux] (so I guess that's the most widespread legacy of our work)
- What do we mean by lesystem facilities: a more powerful fs would make things better?
  - Back in 200x we mentioned [the “clueless” whitepaper] how we needed more low-level tooling from the underlying operating system in order to be able to realize some of the ideas of GoboLinux cleanly. We were asking for more abstraction and isolation in userspace: essentially we wanted union lesystems and possibly some sort of containers for ner-grained isolation. Lacking those, we had to make do with chroot.
  - For a while we hoped that as underlying technology matured, these ideas could come to fruition.
  - Docker seems to be proof of that; a container-based system that greatly simpli ed application deployment.
  - However, Glauber Costa, one of the developers of the Linux Containers system, described the limitations of that approach exposed the hackery involved [did he write about it somewhere?]. Costa himself moved away from containers and joined the efforts of the OSv project, a minimalistic operating system targeting hypervisor-based architectures.

3 Language-oriented vs. language-agnostic package managers

An analogy between PL space with DSLs and general purpose languages and pkgmgr space with language-oriented and language-agnostic 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

	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/OSX	OSX	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

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” [1]), 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, for instance, is a complete programming language, but its wealth of features for numerical computing it is often regarded as being domain-specific [10, 9].

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, LuaRocks was written for Lua but also supports MoonScript and Typed Lua; 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 general 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 43,000 packages in total, the Maven Central repository for Java alone contains over 103,000 packages, with the advantage that the repository is portable to various platforms, some of which lacking a built-in universal package manager (Microsoft Windows being a notable case). Still, this kind of effort duplication does happen: the Debian repository contains 715 packages of Ruby modules; this is a far cry from the 100,000 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 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.

3.1 LuaRocks

LuaRocks [14] is a package manager for the Lua ecosystem. It was developed building on our previous experience writing package management tools for GoboLinux and adapting them to the realities of a language-specific manager.

The package specification format (rockspec) was largely influenced by GoboLinux, and LuaRocks pushes the use of declarative specifications even further. While the `.rockspec` files themselves are actually Lua scripts, LuaRocks loads them using a restricted execution environment that disables the Lua standard libraries; they essentially consist of variable declarations, describing the package via Lua tables (Lua's single data structure, which combines numeric and associative arrays).

Another design feature inherited from GoboLinux was the support for multiple build back-ends. In most language ecosystems, at any point in time a single build tool is well-established as a standard (such as `easy_install` and later `pip` for Python, Rake for Ruby, ExtUtils::MakeMaker and later Module::Build for Perl). In the Lua world, by the time LuaRocks was conceived, in spite of several contenders, there was no clear winner as a standard build tool. To deal with the variety of build systems, OS-level package managers typically let developers call their preferred build tools explicitly in imperative scripts. LuaRocks attempted to solve this in a more controlled manner, with a system of plugins for the various tools, akin to the one used in GoboLinux. In LuaRocks, each plugin is implemented as a Lua module, selected through an entry in the rockspec. For example, using `build.type="cmake"` causes LuaRocks to invoke CMake with the appropriate arguments. We also included a simple built-in build system, invoked via `build.type="builtin"`, which is able to install Lua files and portably compile C code into Lua modules. This proved a success among developers: at the time of this writing, 75% of all packages in the LuaRocks repository use the built-in build system.

LuaRocks installs all packages into a sandbox directory and later moves them to their final destinations. The actual installation prefix is never informed to the build system under execution. This prevents a package from hardcoding its own installation directory in its source code. While the ability to do this is a feature sometimes requested by developers (as it would make it easier to load asset files, for example), disallowing hardcoded paths ensures that every package built is relocatable. Having fully relocatable packages is rare on Unix, but is an expected feature on Windows. Having to cater to such conflicting requirements is a constant in writing portable software; to alleviate the issue of finding asset files, one of the authors developed a Lua module called `datafile` [1], resolves directory locations portably at runtime.

As it also happened with GoboLinux, having a high-level description format allowed us to make radical changes to the installation layout once that proved necessary. Since LuaRocks does not provide to specification files any knowledge of the final directory structure, this change was even less impacting for users than the change on GoboLinux: all existing rockspecs could be used in the new directory layout without any changes. This level of information hiding, extending not only the installation prefix, but to all subdirectories, was only possible because we were dealing with a language-specific manager, where we could make assumptions about the contents of files (Lua source code and binary dynamic libraries) and how they would be later used (as command-line scripts or loaded by Lua through its package loader system).

The design changes that LuaRocks underwent from version 1.0 to version 2.0 were due to lessons learned on the specificities of language-specific package management. The original design of LuaRocks was filesystem-oriented, like GoboLinux. Each version of an installed package was given its own directory and each category of files got its own subdirectory. For example, for module `lpeg 0.12-1` one would store Lua files in `$PREFIX/lib/luarocks/rocks/lpeg/0.12-1/lua`. LuaRocks included then a custom wrapper for Lua's `require()` function in module `luarocks.require`. Once this module was loaded (either via plain `require()` or via a command-line flag), subsequent calls to `require()` would match module names to file locations, taking into account the dependency trees of previously loaded modules. This allowed multiple versions of a module to coexist in an installation, and let modules load the correct versions of their dependencies transparently. The approach of loading a custom `require()` function is the same as adopted by 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[2]. For LuaRocks 2.0, the design was

[1] `http://github.com/hishamhm/datafile`, also available via `luarocks install datafile`

[2] 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.

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. Since now Lua modules from all packages were installed into a single directory such as `$PREFIX/share/lua/5.3`, a database had to be put in place matching files to packages. We kept to Lua's minimalistic approach, using a plain-text manifest file (loadable as Lua code) as a database. Supporting multiple versions of the same package installed at the same time is still possible, but requires the now-optional custom package loader, which produces versioned filenames such as `$PREFIX/lib/5.3/lpeg-0.11-1.so` when the dependency graph requires an old version of a module.

4 Integration between languages vs. integration between package managers

An analogy between PL interaction (APIs and FFIs) and pkgmgr interaction (virtually none??)

- Ad-hoc integrations between general-purpose and domain-specific package managers proved traumatic to some projects [debian's troubles with rubygems].
- Still, they have been successfully realized in others, such as the use of LuaRocks by Buildroot.

4.1 GoboLinux Aliens

- In GoboLinux we researched the idea of building an "FFI" of sorts into our general-purpose package manager, called Aliens [cite Michael's presentation in linux.conf.au].
  - The idea was that we could provide a general API for writing shims that interacted with domain-specific package managers (that is, language-specific ones) in an clean way.

5 Conclusion

To-do.

References

[1]

[2] Bower - a package manager for the web. `http://bower.io`, 2015.

[3] Homebrew - the missing package manager for OSX. `http://brew.sh`, 2015.

[4] The MacPorts project. `http://www.macports.org`, 2015.

[5] node.js. `https://nodejs.org/`, 2015.

[6] npm - Node Package Manager. `http://npmjs.com`, 2015.

[7] John Arrundel. *Puppet 3 Cookbook*. Packt Publishing, August 2013.

[8] Anthony Burns and Tom Copeland. *Deploying Rails*. Pragmatic Bookshelf, July 2012.

[9] Martin Fowler. Language workbenches: The killer-app for domain specific languages? http://martinfowler.com/articles/languageWorkbench.html, jun 2005.

[10] Andy Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12(4):30:30–30:43, April 2014.

[11] Mitchell Hashimoto. *Vagrant: Up and Running*. O'Reilly Media, 1 edition, June 2013.

[12] Michael Homer, Hisham Muhammad, and Jonas Karlsson. An updated directory structure for Unix. In *linux.conf.au 2010*, Wellington, New Zealand, 2010.

[13] Hisham Muhammad and André Detsch. An alternative for the Unix directory structure. In *III Workshop Software Livre*, Porto Alegre, Brazil, 2002.

[14] Hisham Muhammad, Fábio Mascarenhas, and Roberto Ierusalimschy. LuaRocks - a declarative and extensible package management system for Lua. *Lecture Notes in Computer Science*, 8129:16–30, 2013.