LuaRocks - a declarative and extensible package management system for Lua

Hisham Muhammad[1] Fabio Mascarenhas[2] Roberto Ierusalimschy[1]

[1] Department of Computer Science – PUC-Rio – Rio de Janeiro – Brazil
{hisham,roberto}@inf.puc-rio.br
[2] Department of Computer Science – UFRJ – Rio de Janeiro – Brazil
fabiom@dcc.ufrj.br

Abstract. While sometimes dismissed as an operating systems issue, or even a matter of systems administration, module management is deeply linked to programming language design. The main issues are how to instruct the build and runtime environments to find modules and handle their dependencies; how to package modules into redistributable units; how to manage interaction of code written in different languages; and how to map modules to files. These issues are either handled by the language itself or delegated to external tools. Language-specific package managers have risen as a solution to these problems, as they can perform module management portably and in a manner suited to the overall design of the language. This paper presents LuaRocks, a package manager for Lua modules. LuaRocks adopts a declarative approach for specifications using Lua itself as a description language and features an extensible build system that copes with the heterogeneity of the Lua ecosystem.

Keywords: programming language environments, scripting languages, modules and libraries, package management

1 Introduction

While it is sometimes dismissed as an operating systems issue, or even a matter of systems administration, module management (and by extension package management) is deeply linked to programming language design. The questions of how modules are built, packaged, deployed, detected, and used are mostly dependent on decisions in the design and implementation of the languages in which they are written.

In languages that feature a separate compilation step, there's the issue of how to specify dependencies between modules, and how to instruct the compiler to find them. Some languages take care of this matter internally, such as the management of units in Pascal or classes in Java. Others, like C, relegate it to external tools — in the case of C, the preprocessor is used to forward-declare prototypes and tools like Make are used to handle dependencies between objects during build. In contrast, the Java compiler extracts the classes and interfaces a source file references, finds the files where they are defined, and compiles them

on demand. Still, building complex projects usually involves more than sources (including, for instance, generation and conversion of icons, interface description files and other assets, as well as inter-language dependencies), leading to the creation of external tools such as Apache Ant [13].

Packaging modules into redistributable units is another design issue. Some languages define packaging formats as part of their specification. Java has policies for the namespace hierarchy and defines the JAR format, with rules for the file format and its metadata. It also includes a library for reading and writing such archives in its standard library (`java.util.jar`). The .NET Common Language Infrastructure also defines package formats for module bundles, called assemblies, which contain compiled classes and metadata, as well as versioning information [16]. In the other extreme, languages such as C leave the definition of library formats entirely to the operating system and language implementors: support for modularization through dynamic libraries is implemented through OS-specific linkers and runtime support libraries. In all cases, the handling of modules requires some interaction with the operating system due to portability concerns, including varying installation directories and lookup paths.

Languages also employ different approaches when adding support for modules written in different languages. Extensible languages like Perl, Python, Ruby and Lua provide C APIs that allow dynamic libraries to interact with the runtime state of the language's virtual machine [21], as well as facilities to load those libraries into the runtime and register them as modules. Some languages also feature foreign-function interfaces, through which the mapping between functions of external libraries and the language environment are written in the host language itself; an example is the Racket FFI [5]. Those interfaces may be bundled into the language's standard libraries [14], or may be external modules themselves [22]. Loading those external libraries and modules again requires interaction with the operating system, and the extent to which this is performed internally or done by external tools is up to the language's design to define. In the case of modules written in different languages, this means one has two sets of design and implementation aspects to deal with (or even three, when C APIs are used as a bridge between two other languages, as is the case, for example, of LuaJava [18]).

Finally, there is the issue of deployment. While languages such as C and Pascal traditionally left the mapping between modules and files, the physical locations of those files, and the installation processes of the modules to be specified as implementation details, the desire for portability and increased code reuse has led the communities of many languages to attempt to standardize these definitions. From those efforts, a number of language-specific deployment tools have emerged: CPAN for Perl [8], RubyGems for Ruby [7], PIP for Python [29], Cabal for Haskell [17], and so on. While originally developed as external tools, many of these have in fact been integrated into the standard distribution of those languages, and are now considered to be part of their standard libraries, showing that deployment has grown from an OS issue into a core language concern.

These tools are essentially portable, language-specific package managers. Package management, however, is a task of the operating system in platforms such as Linux, and this overlap between OS and language concerns may put the necessity of these language-specific tools into question. This feeling is understandable, but comparing the numbers of packages provided by distributions 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 Ubuntu Linux distribution features 37,000 packages in total, Perl's CPAN alone contains over 23,000 packages, with the advantage that the language's repository is portable to various platforms. Besides, some platforms simply lack universal package management (Microsoft Windows being a notable case). The portability aspect and the great number of packages make a good case for having package managers for programming languages.

This paper presents one such language-specific package manager: LuaRocks, for the Lua programming language. Lua was originally designed as an embeddable language, to be loaded as a library into other programs. As such, it features extensive facilities for inter-language interaction, through a complete and reentrant C API and a first-class type for boxed C pointers. However, features oriented towards the use of Lua as the host program language are more recent: Lua only gained a module system two major revisions ago, in version 5.0, ten years after the first release of the language [15]. With the module system, many of the concerns enumerated above naturally emerged: namespace issues, build methods, packaging formats, deployment and redistribution of modules. The focus of the language in being a portable language with a small footprint meant that Lua would not take the approach of dealing with these issues internally. Instead, it provides the minimal core of an extensible module system, concerning the integration with the language runtime (package loaders, namespace management), and all other tasks are left for external tools to perform. LuaRocks is an integrated solution for these tasks related to module management, providing a portable build system for both C and Lua code, package format specifications and a package management tool for remote deployment of modules.

2 Related work

This section provides background on package management, tracing its origins as operating system tools and the history of language-specific package managers. As systems grow in complexity, library dependencies become harder to track. Package management is the most common solution for this problem [33]; on environments without system-wide package management, these conflicts have to be tracked on a file-by-file basis [23], which is a more fragile approach [34].

2.1 Operating system package managers

The idea of having a unified system for building and installing packages can be traced back to open source operating systems in the 1990s. The growth of

both the free software movement and the commercial internet meant that a large number of independently developed projects were available in source form. However, much of this software could not be built unmodified in a variety of platforms, often requiring OS-specific patches to adapt them to the peculiarities of each system. In 1993, the Debian project introduced `dpkg` [19], a program for installing, removing and keeping track of installed *packages*, which are archives containing all files that compose a given compiled program. In 1994, FreeBSD introduced the Ports collection, a system of Makefiles that provided a unified interface for building software from third-party (*upstream*) developers while automatically applying compatibility patches [20]. Having a Makefile in the Ports collection means that a program can be easily installed into FreeBSD by using standardized commands.

Linux distributions soon adopted this concept. Red Hat Linux was the first distribution to gain popularity on the merits of its package management system, called RPM [3]. RPM combined both the facilities for creating binary packages found in `dpkg` with the unified method for building sources from Ports. Later, Debian introduced APT, a front-end tool to `dpkg` which included dependency resolution, recursively scanning for package dependencies, fetching necessary packages over the network and installing them in topologically-sorted order [19]. Over time, many other package management tools emerged, and these features have grown to become the essential expected feature set: fetching packages remotely; resolving dependency graphs; and installing, removing and listing packages. Current versions of FreeBSD Ports also allow the installation of precompiled packages, and RPM performs dependency management. Some of these features have also evolved in sophistication, for instance, with the distinction between *build dependencies* (packages that need to be installed in the system where the package is being compiled, such as a parser generator or a set of C header files) and *runtime dependencies* (packages that need to be installed in the system where the package will run, such as a shared library).

In recent years, deployment tools for centralized package management have been adopted in platforms for distribution and sale of binary packages as well: these are usually named "application stores". Some examples are the Apple App Store, Google Play and the Amazon Appstore.

2.2 Language-specific package managers

The history of language-specific package managers can be traced to online repositories of software. CPAN [8], the Comprehensive Perl Archive Network, was mainly influenced by CTAN, a repository for TEX class files. Created in 1995, CPAN is the oldest repository for language modules and over the years evolved into a fully-featured package manager. Figure 1 lists 15 of the most popular language specific package managers, along with their start years and number of available packages. Over the last 15 years, many languages, especially those associated with the notion of scripting [24], have gained package managers of their own. Some languages define a official package format as part of their specification, such as JAR for Java, and some include the package management tool along

Language	package manager / repository	packages	included in lang. distr.	official pkg. format	repository start year	direct publishing
Java	Maven/Central	56697	no	yes	2005	no*
Ruby	RubyGems	55035	yes	yes	2003	yes
Python	pip/PyPI	32180	no	yes	2003	yes
JavaScript	npm (node.js)	27688	yes	yes*	2009	yes
Perl	CPAN	24092	yes	yes	1995	no
C#/.NET	NuGet	11823	no	no	2011	yes
PHP	Composer/Packagist	9757	no	no	2011	yes
Clojure	Leiningen/Clojars	6004	no	yes	2009	yes
Haskell	Cabal/Hackage	5062	no**	yes	2007	yes
R	CRAN	4450	yes	yes	1997	no
Objective-C	CocoaPods	1391	no	no	2011	no
Common Lisp	Quicklisp	850	no	no	2010	no
Go	go	744	yes	no	2009	no***
Racket	PLaneT	510	yes	yes	2004	yes
Lua	LuaRocks	266	no	no	2007	no

* The Maven Central is a two-tier repository: it aggregates a number of approved repository, some of which may provide direct publishing functionality.
** Cabal is not included with Haskell implementations such as GHC and Hugs, but it is part of the Haskell Platform "batteries" package from haskell.org.
*** The Go repository is in fact just a wiki of links to projects which can be imported directly with the Go `import` statement; editing the list requires contributor access.

Fig. 1. Language-specific package managers, as of April 16, 2013

with the sources of the language reference implementation. These are identified in Figure 1 as well.

Following the steps of CPAN, CRAN [1] was started in 1997 as a repository and later package manager for R, a niche language in the field of statistics. In 2003, RubyGems was created for the Ruby language. Unlike its predecessors, RubyGems [7] allows any developer to publish modules directly in the public repository, without any curating process. By lowering the barrier of entry early on, RubyGems gained enormous popularity and became the largest module repository among scripting languages. In fact, the aspect that seems to affect most directly the number of available packages in a repository is whether the repository allows developers to publish packages directly or if it requires some kind of approval step. From the 15 languages listed in Figure 1, 8 allow direct publishing of modules; 7 of them are in the top 9 positions when ranked by number of available packages. The two exceptions in the top positions are Maven's Central, which is an aggregator of repositories, and CPAN, which has a large total of packages due to being much older than the other repositories. Maven [2] is a build and deployment tool for Java, which eventually evolved into a full-fledged package manager. Maven Central is currently the largest language-specific package repository in existence, with over 56000 packages [32].

Package management systems for Python have had an eventful evolution [35]. The package management tool has been added and then removed from the main Python distribution, and the original tool, `easy_install`, was eventually replaced by `pip`. Still, the package repository, PyPI (Python Package Index) has seen continuous growth [29]. PHP originally had two official package repositories, PEAR and PECL, respectively for PHP extensions and library bindings. These are not open for direct publishing of modules. Eventually, a new package manager, Composer, was created alongside a new open repository, called Packagist. Composer and Packagist quickly eclipsed the original repositories: while PEAR/PECL have less than 900 packages, Packagist features over 9700 [26].

The JavaScript world did not have a package manager until 2009, when `npm` was created. This tool has the peculiarity among package managers of being not only language-specific, but in fact framework-specific, being a tool created to be used with Node.js, an event-driven platform for server-side development [28].

Objective-C, like C and C++, does not define its own package format, but it has an unofficial package management system for class libraries. The CocoaPods project [9] was started in 2011 and hosts modules for iOS and Mac OS X platforms. It has the distinction of being the only one of the language-specific package management systems studied that is not implemented in the target language itself: instead, CocoaPods is written in Ruby, and it is in fact distributed as a Ruby gem. In the .NET platform, there is also no official package manager, but NuGet [25] is a popular tool, which integrates with the Visual Studio IDE.

The Go language adopts a very unusual approach towards module management. Go bundles the compiler, build and deployment tools, and instead of using a centralized repository, adds support for decentralized cross-reference of modules in the language itself: its `import` statement can refer to full URLs which point to source code repositories [4].

Cabal [17] is the package manager for Haskell. Due to the language's sophisticated type system, Haskell modules are known for their intricate dependency relations, as minor interface changes cause incompatibilities and there is no way for incompatible packages to coexist in an installed environment [31]. Other package managers worth mentioning are: PLaneT [27], for Racket; Quicklisp [6], for Common Lisp (which aims to be compatible with several implementations of the language standard); and Leiningen [12], for Clojure (a language that targets the Java Virtual Machine and therefore also uses the JAR format for packages).

Architecturally, all these tools are very similar to their OS-level counterparts, as they perform the same basic tasks: fetching modules; resolving dependencies; and building, installing, and removing modules. A common issue is avoiding conflicts with packages installed by the OS package manager, and how to inform the language runtime about newly installed modules. Old versions of Ruby, for example, required the user to write `require "rubygems"` to enable gem-installed modules, but more recently this support has been integrated by default. Modules that feature dependencies on external libraries, such as bindings to C libraries, are another point of concern. Each package manager specifies its own syntax for locating these libraries, and they often make OS-specific assumptions such as

filenames and paths. Integration between OS-level and language-level package managers is a problem that cannot be solved in the general case. For example, a module providing bindings to a JPEG library may be aware that the library is provided by an OS package called `jpeg-dev` in one platform, `libjpeg6-dev` in another, or even by a file called `JPEG.DLL` available somewhere in the library path, when the OS does not feature a standard package manager.

3 The design of LuaRocks

LuaRocks is written as a pure Lua application and does not assume the availability of any other Lua modules in the system. To perform operations not provided by stock Lua, such as manipulating directories or downloading remote files, it can either launch external programs (e.g. `wget`) or use additional modules such as LuaSocket, depending on what is available. On Windows, a set of helper binaries is included in the distribution that aids the bootstrapping process.

On the surface, LuaRocks behaves like any other package manager. It provides two command-line tools: `luarocks`, the main interface; and `luarocks-admin`, for managing remote repositories. These tools support typical commands, such as `luarocks install` ⟨*package_name*⟩ and `luarocks remove` ⟨*package_name*⟩, respectively for installing and removing packages, while performing recursive dependency matching as expected. While all package managers perform essentially the same tasks, the specifics of each environment impose some design restrictions while opening up some possibilities. In this section, we discuss the novel aspects in the design of LuaRocks. They explore the potential of Lua as a data description language, its sandboxing facilities, and the extensible solutions LuaRocks uses to deal with the heterogeneity of operating systems and build tools that developers use. We also discuss the approach we take to versioning, which makes it easier to deal with package conflicts.

3.1 Declarative specifications

Package management tools usually define a file format through which packages are specified. Those files can be as simple as a Makefile, as is the case with FreeBSD Ports [20], or may contain various metadata and embedded build scripts, such as .spec files for the RPM package manager. For specifying LuaRocks packages, which we call "rocks", we devised a file format called "rockspec", which is actually a Lua file containing a series of assignments to predefined variable names such as `dependencies` and `description`, defining metadata and build rules for the package.

Rockspecs are loaded by LuaRocks as Lua scripts inside a sandbox that allows the use of Lua syntactical constructs, but no access to its standard libraries or external libraries. This ensures that the loading of the package specification is safe: loading a rockspec file (for example, for syntax verification with the `luarocks lint` command) can at most lock the command-line tool through an endless loop, but it is not able to access any system resources. Even the

```
%define luaver 5.1
%define lualibdir %{_libdir}/lua/%{luaver}
%define luapkgdir %{_datadir}/lua/%{luaver}
Name: luasocket
Version: 2.0.2
Release: 8%{?dist}
Summary: Network socket extension for Lua
# ...
Source0: http://.../luasocket-2.0.2.tar.gz
Patch0: lua-socket-unix-sockets.patch
# ...
%prep
%setup -q -n luasocket-%{version}
%patch0 -p1 -b .unix
%build
make %{?_smp_mflags} CFLAGS="%{optflags}
↪ -fPIC"
%install
rm -rf $RPM_BUILD_ROOT
make install
↪ INSTALL_TOP_LIB=$RPM_BUILD_ROOT%{lualibdir}
↪ INSTALL_TOP_SHARE=$RPM_BUILD_ROOT%{luapkgdir}
%clean
rm -rf $RPM_BUILD_ROOT

# ...
```

(a) RPM .spec file[3]

```
package = "LuaSocket"
version = "2.0.2-5"
source = {
   url = "http://.../luasocket-2.0.2.tar.gz",
}
description = {
   summary = "Network support for the Lua language",
   -- ...
}
build = {
   type = "make",
   build_variables = {
      CFLAGS = "$(CFLAGS) -I$(LUA_INCDIR)",
      LDFLAGS = "$(LIBFLAG) -O -fpic",
      LD = "$(CC)"
   },
   install_variables = {
      INSTALL_TOP_SHARE = "$(LUADIR)",
      INSTALL_TOP_LIB = "$(LIBDIR)"
   },
   -- ...
}
```

(b) LuaRocks rockspec[4]

Fig. 2. Excerpts from specification files for LuaSocket 2.0.2 using RPM and LuaRocks, including basic package identification, download URL and build instructions

possibility of an endless loop can be removed using hooks in the Lua virtuam machine, making the loading of rockspecs completely safe for use by servers that accept arbitrary rockspecs.

While the loading of a rockspec is imperative, it is not a "build script", but a declarative specification of the package and its build process. Imperative build scripts impose a strict order on operations. A rockspec does not list the sequence of build operations in order as a makefile or an RPM .spec would (Figure 2a), but rather contains definitions which describe the build method declaratively (Figure 2b). The use of declarative descriptions gives us more liberty as tool implementors to make changes to the way the build process is implemented from one version of LuaRocks to another.

Rockspecs allow developers to make higher-level descriptions of their build processes, as we will see in more detail in Section 3.2, and let the tool handle low-level details such as portability adaptations. As a simple example, the invocation of the `make` command is explicit in Figure 2a and implicit in Figure 2b, which allows LuaRocks to adjust the command name to `gmake` in some BSD environments. LuaRocks also provides a general method for conditionally replacing entries in a rockspec in a per-platform basis. For example, a field named `source.platforms.win32.url` will overwrite the `source.url` field on Windows platforms and will be ignored on other operating systems. Through `platforms` subtables, a developer can conditionally specify platform-specific build flags, module dependencies and external library requirements.

[3] Full file at `http://pkgs.fedoraproject.org/cgit/lua-socket.git/tree/lua-socket.spec?h=f18`

[4] Full file at `http://luarocks.org/repositories/rocks/luasocket-2.0.2-5.rockspec`

After LuaRocks compiles and installs a rockspec, the rockspec maintainer can package it as a .rock file, which is a .zip archive containing all modules, the rockspec and a manifest file. Manifest files are essentially plain-text databases for package management, implemented as Lua tables which are loaded in the same sandbox used for rockspecs and saved using a simple serialization procedure.

While each rock has its own manifest in a `rock_manifest` file (containing also the MD5 checksum for each deployed file), LuaRocks also caches a global manifest for all packages in a system `manifest` file for quicker initialization. This global manifest has indexes for efficiently finding dependencies between packages, which package owns a module, and which modules a package owns. This same style of global manifest is used in remote repositories as a directory of available packages. In short, LuaRocks stores all of its metadata as Lua source files, making heavy use of Lua facilities for sandboxes and data description.

3.2 Extensible build system

One aspect in which the design of LuaRocks resembles OS-level package managers more than typical language-specific package managers is in its handling of build tools. Often, language-specific repositories are built around one specific tool: for example, `easy_install` and later `pip` for Python, Rake for Ruby, ExtUtils::MakeMaker and later Module::Build for Perl. The package manager then delegates the build process to the build tool and focuses on tracking installed files and dependencies. Lua, however, does not have a standard build system. By the time LuaRocks was created, a number of Lua-based build tools had been proposed, but none have gained traction in the developer community. Most Lua modules were distributed by upstream authors along with Unix Makefiles only, or with no build scripts at all, and the user is expected to build and deploy the modules by hand. Some developers also use other tools, such as CMake.

To support these uses, LuaRocks supports several build tools, like OS-level package managers normally do. OS-level package managers typically let developers call their preferred build tools explicitly in imperative specification scripts, as seen in the call to `make` in the .spec file from Figure 2a. This is an open-ended approach that allows the use of any build tool, at the cost of having hard-coded references and a low abstraction level, akin to a shell script or a batch file.

LuaRocks, however, does this in a more controlled manner, with a system of plugins for the different build tools. Each plugin is implemented as a Lua module, and selected through the `build.type` field in the rockspec. For example, using `build.type="make"` (as in Figure 2b) causes LuaRocks to load the module `luarocks.build.make`, which is then responsible for providing the necessary plumbing that connects LuaRocks with the build tool.

The `build` field has additional entries specific for the build type, which are passed to the plugin. These entries, and a set of context variables describing the system where LuaRocks is installed, can be used to parameterize the build. For example, in Figure 2b, the plugin responsible for the "`make`" build type interprets the `build.build_variables` and `build.install_variables` entries, passing the appropriate variables to the build tool. Other customizations are possible: for

```
package = "midialsa"
version = "1.17-1"
source = {
   url = "http://www.pjb.com.au/comp/lua/midialsa-1.17.tar.gz",
   md5 = "0482df57c2262ff75f09cec5568352a7"
}
description = {
   summary = "Provides access to the ALSA sequencer", detailed =
[[ ... ]],
   homepage = "http://www.pjb.com.au/comp/lua/midialsa.html",
license = "MIT/X11"
}
dependencies = { "lua >= 5.1" }
external_dependencies = {
   ALSA = { header = "alsa/asoundlib.h", library = "asound" }
}
build = {
   type = "builtin",
   modules = {
      ['C-midialsa'] = {
         incdirs = { "$(ALSA_INCDIR)" },
         libdirs = { "$(ALSA_LIBDIR)" },
         libraries = { "asound" },
         sources = { "C-midialsa.c" }
      },
      midialsa = "midialsa.lua"
   },
   copy_directories = { "doc", "test" }

}
```

Fig. 3. Rockspec for a module using the `builtin` build type.

instance, the default `make` target for installation is "`install`", but one can override that using `build.install_target`. Any of these fields can be specified in a platform-specific manner. For example, a rockspec may specify build variables specific to the Windows platform in a `build.platforms.win32.install_target` field.

The first release of LuaRocks shipped with support for three build types: `make`, `cmake` and `command`. The `command` type is a catch-all backend for unsupported build tools: it allows writing a pair of operating system commands in the rockspec (`build.build_command` and `build.install_command`) which LuaRocks then calls. Early on in its history, however, a fourth standard build type was added, called `builtin`.

The `builtin` build type, as the name suggests, is a lightweight built-in build tool integrated with LuaRocks. It was designed to cover the common cases when a module is either written in pure Lua, or contains C code that can be compiled without sophisticated pre-configuration.

Figure 3 depicts a complete rockspec that uses the `builtin` build type. This is the rockspec for midialsa, Lua bindings for the MIDI features of the Advanced Linux Sound Architecture (ALSA). This package installs a module written in Lua, `midialsa`, which provides a high-level Lua API, and a module written in C, `C-midialsa`, which links to the ALSA library and provides core functions for the Lua module. This is a fairly typical setup for library bindings.

The `builtin` build type expects a `modules` map. In the simpler cases, such as pure Lua modules, it associates the name of each module to the source file that implements it. For modules written in C, one can specify more metadata. These are typically paths where to find headers and libraries needed to build the module, names of libraries the module depends on, and the source files for the module. In the example of Figure 3, `C-midialsa` specifies that it needs to be

linked to the `asound` library and has its implementation in the `C-midialsa.c` file.

The example also references two context variables that LuaRocks provides, `ALSA_LIBDIR` and `ALSA_INCDIR`. LuaRocks defines these variables after detecting the location of the `alsa/asoundlib.h` and `asound` library the rockspec specifies in the `external_dependencies` section of the rockspec.

To deal with variations between operating systems, external dependencies to libraries are given as abstractly as possible: based on `libraries = "asound"`, LuaRocks will look for files matching one of various possibilities: `libasound.so`, `libasound.so.*`, `libasound.dylib`, `ASOUND.DLL`, `ASOUND.LIB` and so on, depending on the running platform. LuaRocks searches for these files in a series of OS-specific directories. This flexible approach for dependency verification has proven to be a good compromise solution that limits the amount of OS-specific information in the specification file and keeps platform-specific metadata to a minimum. The locations of system header and library directories can, if necessary, be adjusted permanently by the user in a configuration file, or on a case-by-case basis with command-line arguments.

The `builtin` plugin launches the C compiler and linker, passing proper flags for the system it is running on. It has internal support for the GCC and Visual Studio toolchains by default, but it is also largely configurable. All programs and flags used can be overridden using familiar variables such as `CC`, `LD` and `CFLAGS` in the LuaRocks configuration file, in particular making it easy to use alternative compiler toolchains, including cross-compilers.

The `builtin` build type provides LuaRocks with a build tool that is declarative, like the rest of the rockspec format. Platform-specific details are abstracted as much as possible, and can be added only when needed. This gives an easy way for developers who often shipped Unix-only makefiles to support Windows builds with little effort. Still, developers wishing to continue using other tools such as make, CMake, and GNU Autotools can easily do so. LuaRocks integrates well with these tools, as all configuration variables it provides are available for all build plugins, including those by detected external dependencies such as `ALSA_LIBDIR` in the above example, and others such as `PREFIX` and `CFLAGS`.

For all build types, LuaRocks executes the build stage targeting a temporary sandbox directory in its `PREFIX` variable, later moving the generated files to their final locations. This forces relocatability: there is no way for a module compiled through LuaRocks to hard-code its install location. In other words, any module built with LuaRocks can be packed into a .rock file with `luarocks pack` ⟨*rock*⟩ and then installed in a different directory. This is important when deploying binaries, particularly on Windows environments.

3.3 Versioning

Another feature that sets LuaRocks apart from other package managers is the fact that it supports multiple simultaneous versions of the same package in a single installed tree, so that one can, for example, install two modules A and B, where A depends on C version < 2 and B depends on C version ≥ 2. LuaRocks

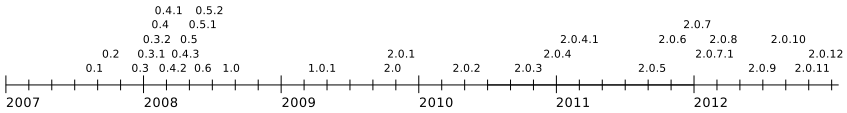

Fig. 4. Timeline of LuaRocks releases

allows all four modules to remain installed in the same directory simultaneously, and provides runtime support so that the correct version of C is used for either A or B.

When LuaRocks installs a new version of a module, it renames the old version so they can coexist in the same directory (adding the rock name and version as a prefix). The idea is that Lua will always find the latest installed version of each module, as that file will have a standard pathname such as `/usr/local/lib/lua/5.1/socket.so`.

Users who need support for loading versions other than the latest one can use a custom module loader that LuaRocks provides. Module loaders are the extensibility mechanism for the Lua module system. Whenever a module is requested, Lua tries to load it using a series of loader functions registered in a list. The LuaRocks module loader keeps in memory a "context", which is the list of previously loaded modules, the rocks they belong to, and their dependencies, so that when Lua needs to load a new module, the LuaRocks module loader can choose a version based on dependencies from the current context.

This approach to versioning alleviates the so called "dependency hell" experienced in many other package managers. If the user wants to write a script using a module that happens to depend on a version of another module that is different than what is already installed on their system, they are free to install that additional dependency without worrying that other modules that depend on the previously installed version will break. When using package managers that lack this feature, the workaround is to create separate local module trees in different directories and configure the runtime environment accordingly whenever each script is run. Some languages even feature tools that encapsulate this usage pattern, creating replicated environments to avoid conflicting dependencies: RVM [30] for Ruby and Virtualenv [11] for Python are two examples.

4 Development history

The initial release of LuaRocks was published in August 2007. After thirteen 0.x releases, LuaRocks 1.0 was released in September 2008 (See Figure 4), and the rockspec format specification has been frozen ever since. LuaRocks 2.0 was released in October 2009, introducing the custom module loader.

Given that Lua has a history as a language for embedding into applications and games, where the only additional modules are those specific to the underlying program, the developer community for reusable modules is small compared to languages with a focus on areas such as, for example, web development. Still, the LuaRocks repository has shown a steady growth. Figure 5, generated from

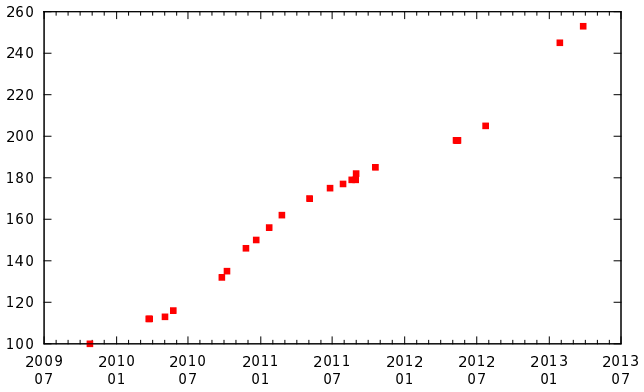

Fig. 5. Number of packages in the LuaRocks repository during the 2.0 series, from October 2009 to March 2013

archive snapshots of the repository index, shows the growth of the collection, from October 2009 when LuaRocks 2.0 was released and the repository contained exactly 100 packages, up to March 2013, when we just surpassed 250 packages. During this time, the 2.0 series had a number of point releases. Save for bugfixes, these releases are essentially compatible. They were mainly driven by feedback and contributions from users, and were focused on improving portability, adding new commands to the `luarocks` and `luarocks-admin` command-line tools, and improving user experience with better platform detection.

The `builtin` build plugin proved to be quite popular. As of this writing, of the 258 projects in the LuaRocks repositories, 195 of them use the `builtin` build type, and only 26 use `make`[5]. In particular, from those 195 rocks, 29 of them originally used the `make` build type and later switched to `builtin`, suggesting that it was a good strategy to allow developers to warm up to the idea of using LuaRocks by letting them start to use it along with their existing build systems. The `make` build type often exposed shortcomings in the developers' makefiles, such as poor support for specifying custom install paths and linker flags. This was often noticed when Mac users attempted to install rocks written by Linux developers and vice versa, and also as developers transitioned from x86 to x86-64. The `builtin` type handles those issues transparently.

5 Conclusion

In recent years, language-specific package managers have become an essential part of programming language ecosystems, as the internet allows large communities of developers to build upon each other's work by reusing modules. The exact role and scope of language-specific package managers vary from language

[5] From the 37 remaining projects, 10 use `command`, mostly for invoking GNU Autotools, and 27 use `none`, which is a blank build type for merely copying .lua files (a predecessor of `builtin`).

to language, as the definitions of what is handled by the language and what is delegated to the package manager are language design decision themselves. Still, these developments have been underrepresented in academic literature so far.

This paper presented LuaRocks, a package manager for the Lua programming language. LuaRocks brings some novel concepts to language-specific package manager design, such as a completely declarative integrated build system, thorough use of the language itself as its data description language (which allows the tool to bootstrap itself without any external dependencies) and support for coexisting versions of modules in local repositories, with runtime support for dependency resolution based on the extensibility mechanisms of the Lua language.

LuaRocks is used in production systems around the world and is included in repositories of several Linux distributions. As of this writing, the rocks repository features 750 rockspecs for 258 different projects. LuaRocks users have reported success using it in a number of platforms, such as Windows (either natively, with Cygwin, or with Mingw32), Linux, Mac OS X, FreeBSD, OpenBSD, NetBSD and Solaris. The directions of development nowadays are essentially dictated by the needs of the community, while trying to balance concerns of compatibility, portability and ease of use. For many developers, especially those used to other languages that already have similar ecosystems in place, LuaRocks is their introduction to writing and sharing Lua modules.

The declarative rockspec format proved to be a success among developers, and its specification remains largely frozen since LuaRocks 1.0. Still, we have identified some possibilities for improvement of the format over the years, and the next major release may include a revision of the specification, while keeping backward compatibility. LuaRocks is prepared to recognize incompatibilities through the `rockspec_format` field, so the transition shouldn't be traumatic.

Another frequent request is implementing support for LuaRocks to upgrade itself. The tool already has experimental support for that, but it is not enabled by default, since the interaction with installations made through OS-level package managers still has to be assessed.

Another plan is to eventually allow direct publishing of modules by developers. This requires development of server-side infrastructure, but the LuaRocks community has already started efforts in this direction, with an alternative repository called MoonRocks which allows direct publishing [10].

References

1. Adler, Joseph. "R in a Nutshell". pp 37-47. O'Reilly Media. October 2012. ISBN 144931208X
2. Apache Software Foundation. *Apache Maven Project.* `https://maven.apache.org/`
3. Bailey, Edward. "Maximum RPM", 450pp. Sams. August 1997. ISBN 0672311054.
4. Balbaert, Ivo. "The Way To Go: A Thorough Introduction to the Go Programming Language" pp. 203-223. iUniverse. March 2012. ISBN 1469769166
5. Barzilay, Eli; Orlovsky, Dmitry. Foreign Interface for PLT Scheme. *Fifth Workshop on Scheme and Functional Programming.* September 22, 2004, Snowbird Utah, USA.

6. Beane, Zach. *Quicklisp*. http://www.quicklisp.org/
7. Berube, David. "Practical Ruby Gems". Apress. April 2007. ISBN 1590598113
8. Christiansen, Tom; Foy, Brian D.; Wall, Larry; Orwant, John. "Programming Perl" p. 629-644. 4th edition. O'Reilly Media. ISBN 0596004923
9. *CocoaPods*. http://www.cocoapods.org
10. Corcoran, Leaf. *MoonRocks*. http://rocks.moonscript.org/
11. Gift, Noah; Jones, Jeremy. "Python for Unix and Linux System Administration". pp. 279-283. O'Reilly Media. August 2008. ISBN 0596515820
12. Hagelberg, Phil, et al. *Leiningen*. http://leiningen.org/
13. Hatcher, Erik; Loughran, Steve. "Java Development with Ant", 672pp. Manning Publications, August 2002. ISBN 1930110588
14. Heller, Thomas et al. *Python CTypes*. http://docs.python.org/3/library/ctypes.html
15. Ierusalimschy, Roberto; Figueiredo, Luiz Henrique; Celes, Waldemar. The evolution of Lua. *History of Programming Languages III*, June 2007, San Diego, USA.
16. ISO/IEC 23271:2012. Information Technology — Common Language Infrastructure (CLI)
17. Jones, Isaac; Peyton Jones, Simon; Marlow, Simon; Wallace, Malcolm; Patterson, Ross. The Haskell Cabal: A Common Architecture for Building Applications and Tools. *Haskell Workshop 2005*.
18. Kepler Project. *LuaJava — A script tool for Java* http://keplerproject.org/luajava/
19. Krafft, Martin. "The Debian System: Concepts and Techniques", 608pp. No Starch Press. September 2005. ISBN 1593270690
20. Lehey, Greg. "The Complete FreeBSD: Documentation from the Source". 4th edition. p.167-180. O'Reilly Media. April 2003. ISBN 0596005164.
21. Muhammad, Hisham; Ierusalimschy, Roberto. C APIs in extension and extensible languages. *Journal of Universal Computer Science*, Vol. 13, No. 6.
22. Mascarenhas, Fabio. *Alien — Pure Lua extensions*. http://mascarenhas.github.com/alien/
23. Microsoft, MSDN Library, "Side-by-side Assemblies (Windows)", August 2010. http://msdn.microsoft.com/en-us/library/aa376307.aspx
24. Osterhout, John. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, Vol. 31, No. 3, March 1998, pp. 23-30.
25. Outercurve Foundation. *NuGet Gallery*. https://nuget.org/
26. *Packagist*. http://www.packagist.org
27. *PLaneT Package Repository*. http://planet.racket-lang.org/
28. Powers, Shelley. "Learning Node" pp. 63-79. O'Reilly Media. October 2012. ISBN 1449323073
29. *Python Package Index*. https://pypi.python.org/pypi
30. *Ruby Version Manager*. https://rvm.io/
31. Snoyman, Michael. "Solving Cabal Hell." http://www.yesodweb.com/blog/2012/11/solving-cabal-hell
32. Sonatype Inc. *The Central Repository*. https://search.maven.org
33. Spinellis, Diomidis. Package Management Systems. *IEEE Software*, 29(2):84–86, March/April 2012. (doi:10.1109/MS.2012.38)
34. Worthmuller, Stefan. "No End to DLL Hell!", Dr. Dobb's Journal, September 2010. http://www.drdobbs.com/windows/no-end-to-dll-hell/227300037
35. Ziadé, Tarek. "Chronology of Packaging". http://ziade.org/2012/11/17/chronology-of-packaging/