A Scheme Towards Reproducibility

Felix Christ, Henning Stubbe[*]
[*]*Chair of Network Architectures and Services*
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: felix.christ@tum.de, stubbe@net.in.tum.de

***Abstract*—The result of compiling software depends on a myriad of factors. Describing the dependencies and the build environment in their entirety is difficult, but necessary to reach an output that is always the same for every compilation.**

With the functional package manager Guix, software packages are described in the functional programming language Guile Scheme, which makes identifying dependencies simple. Guix also provides ways to describe the build environment such that it can easily reproduced. These factors combine to allow wholly reproducible builds.

We packaged a piece of moderately complex software for Guix, and have found that this is a suitable way to achieve a reproducible build.

***Index Terms*—reproducibility, guix, functional package management**

1. Introduction

It is a difficult task to build and run software reproducibly, or even reliably. This is because both compilation and execution depend on many inputs. These inputs, commonly called “dependencies”, are required during build time (such as compilers and build systems) and during runtime (such as interpreters or shared libraries).

To add complexity, software also usually depends on exact versions of their dependencies. As an example, a program written to run with version 2.7 of the python interpreter may not work with version 2.6, if the program uses new features from 2.7. It may also not work with version 3.0, since python syntax between the versions is generally not compatible. Here, it is therefore not enough to state “python” as a dependency, but also the exact version.

The problem is made more difficult still because dependencies also have dependencies. To guarantee that a piece of software is built and run deterministically, it is necessary to have a system that defines software in such a way that it is possible to determine all recursive dependencies. Such a system would ideally also have a simple way of describing the entire environment, i.e. operating system, with all its software and specific versions, in which the build was performed.

Guix is a package manager and operating system that provides these features. In this paper, we set out to describe package management with Guix and how it can help make build processes reproducible in Section 2. We then package a concrete piece of moderately complex software with Guix in Section 3. In Section 4, we are then able to evaluate this packaging process in terms of complexity, and determine whether Guix is suitable for reproducibly building software. Section 5 briefly lists previous uses of Guix in academia, and a conclusion is reached in Section 6.

2. Guix

Guix is a package manager introduced in 2013 [1]. As of the time of writing, it includes 21436 packages of free software. It sets itself apart from other package managers such as `pacman` or `RPM` as a functional package manager. This difference lies in how packages are defined.

2.1. Functional Package Management

Packages, i.e. the building and installation process, are represented as pure functions, in the sense of functional programming. A pure function is a function with the following properties:

1) The function will always evaluate to the same value given the same input (i.e. it is deterministic).
2) It is free of side effects.

In the context of building software, the first property means the build processes do not depend on the state and available dependencies in the operating system,

The inputs (source code, all dependencies) for building software can be considered to be the function’s parameters. The finished build is considered to be the result of evaluating the function. In Guix, this is achieved by creating a container for each build that is separated from the host OS.

2.2. Building packages

A build daemon builds packages in a `chroot` environment. It is used because it provides a lightweight way to control which programs a process uses and has access to by changing the apparent root directory [2]. Inside this container, only the dependencies explicitly listed in the package definition are visible. This ensures that the evaluation stays pure, and does not depend on the state of the host system. It is the build daemons task to configure an environment that includes not only libraries and headers needed for compilation but also explicitly defined build systems and compilers.

Now it is clear how Guix differs from other, imperative package managers. Instead of packages modifying the state of the system unpredictably, as with privileged shell scripts, and depending on the packages installed in the system, installing and updating packages is instead transactional. This means that these processes can be recreated and rolled back easily. Also, because building is mediated through the daemon, and installation only requires linking to the built binaries, all actions are generally unprivileged. This is not the case for imperative managers, which need to move binaries into protected locations, such as `/bin`.

Instead of these traditional locations, all package build results are stored in a central location, the *store*. Each directory contained is prefixed by a cryptographic hash over the "function inputs". The *store* acts as a "cache of function results" [1] so that repeated evaluations of a package can be substituted by its result.

When a user wishes to install a package, they may do so by issuing an unprivileged call to the build daemon, which produces the packages inputs and dependencies, and evaluates the package function. It then stores the resulting binaries, libraries, and header files in the *store*. Links to these results are then added to the user's *profile*, a directory containing user-specific versions of the directories usually found in the root directory, such as `bin`, `etc`, or `include`.

2.3. Guile Scheme

Guix defines packages in the functional programming language Guile Scheme. While Guix's predecessor Nix shares the same mechanism for building and managing packages, Scheme is what separates them. Nix featured a different, less intuitive domain-specific language, called the "Nix expression language" [3]. The developers of Guix intended Scheme to offer simplicity so that developers could "help grow and maintain a large software distribution" [1].

These Schemes explicitly state the inputs required to build and run the software. Things that are considered inputs in that sense include other packages providing dependencies, arguments for the build system, as well as source code (from git or tarballs) [4].

Because other packages are also inputs, and they too are defined as Schemes, Guix can easily traverse and describe all dependencies of a defined package. This is particularly useful when the build daemon needs to determine which packages need to be available in the `chroot` environment during build time. It constructs all inputs, and their inputs recursively. It can then determine which of these inputs are already present in the `store`, and build all those that are not.

2.4. Channels

Collections of Guix packages are organized into git repositories called channels [5]. These channels contain collections of Scheme (`.scm`) files with package definitions. Additionally, they may also define the URL of a CI server to download pre-built packages from, patches needed for certain packages to be built, channel dependencies (other channels that are required for this one), and keys of the channel authors to authenticate commits.

While Guix includes a default channel by default, users may add additional channels. Contributors may also choose to define and publish their own channel by simply making a git repository available online.

Channels also make it possible to easily describe the state of a system with its available packages. At a particular point in time, all information that is needed to recreate a system is the URLs of its channels, as well as the commit that represents the channels' current state.

3. Packaging LLVM-LC3

The goal of this section is to describe the process and assess the complexity of packaging software for Guix. We set out to build the code generator of the LLVM toolchain with the ability to generate code for the "Little Computer 3" (LC3) educational assembly language.

The LLVM toolchain is comprised of a frontend and a backend. The **frontend** translates code from a high-level language to an intermediate representation (IR). There exist frontends for languages such as C/C++ (with `clang` as frontend), Haskell (`kaleidoscope`), or Go (`llgo`). This IR may then be translated into the target instruction set by the **backend**. Our goal is to compile this backend, and include support for LC3 as a target instruction set.

3.1. Understanding the Build Process

The source code for the LLVM backend is hosted on GitHub as part of the LLVM Project. For it to support a target instruction set, a developer needs to define the procedures on how to translate the IR into that specific machine language. Several such implementations that are already included with the backend can be found in the directory `llvm/lib/Target`, such as x86, ARM, or RISCV, with each being represented by a directory with the target's name.

The implementation of the LC3 machine, hereafter referred to as `lc3-target`, is not included, but is fortunately made available by a user on GitHub. Unfortunately, little information on how to include his implementation into the LLVM backend is provided. The trivial approach of adding the implementation into `Target/LC3` is unsuccessful. Two important pieces of information are missing:

1) Where does the backend's code need modifications for the `lc3-target` to be supported?
2) What version of the backend is this `lc3-target` intended for?

We now proceed to identify the steps to build the backend and answer these questions through examining clues and repeatedly invoking the build process after slight modifications. Answering these questions together proves difficult. When a compilation error occurs, it is not immediately clear whether it stems from an incorrect backend version, or from some necessary modification that has not yet taken place.

3.1.1. Modifications in the Backend. The backend's source code needs to be modified in three places. In two places, this is related to adding some members to enums that are used by the `lc3-target`. A modification must also be made in the `CMakeLists.txt` in the `Target` directory, for the build system to pick up the directory in which `lc3-target` resides.

3.1.2. Identifying the Version. We identify the version of LLVM that `lc3-target` was developed for by narrowing down the window of possible versions. Since the last commit in LC3's history is from July 24, 2016, all versions greater than 3.8 are discarded. From there, we work backward to find the correct version. Version 3.8 is discarded because `lc3-target` uses a function that was removed in that version. The next lower, version 3.7, is found to be the correct one.

3.1.3. Finding the Compiler. Even after the correct version is found, compilation still yielded an error within `lc3-target`. An implicit conversion from a `unique_ptr` to `bool` is impossible. Using version 5 of the gcc compiler instead of the latest version fixes this error, which is a known bug with old versions of llvm[1].

3.1.4. Informal Process. After successfully building and verifying the functionality of the backend, we identify the following informal steps.

1) Fetch the source code of the LLVM Project (release 3.7)
2) Fetch the source code for `lc3-target` implementation into LLVM's `Target` directory
3) Register the `lc3-target` within the backend's source code.
4) create a build directory
5) call `cmake` with certain variables to generate build files
6) build llvm

We also identify that `python 2.7` is required to generate the build files. This informal process now must be formalized into a Scheme, defining it as a package.

3.2. Defining a Scheme

Guix features a high-level Scheme data type for representing a `package`. Documentation for it and the rest of this section can be found in the Guix cookbook [5]. This data type has some self-explanatory fields that are just strings, such as the `name`, `version`, or `description`.

3.2.1. Fetching source code. Origins of source code, such as remote git repositories, tar-balls, or local files, are represented as the `origin` data type. A `package` has only one `source` field, which can be considered to be the primary `origin` of the package. In our case, however, two origins of source code are needed. Fortunately, additional `origins` can be added to a package in the `inputs` field, which contains a list of dependencies of the package.

3.2.2. Modifying source code. `origin` objects also include a `patches` field, which contains a list of patch files to be applied to the code. This is useful in our case, as it allows us to modify the code from the official llvm repository to register `lc3-target`.

1. https://github.com/digego/extempore/issues/318

3.2.3. Build systems. Guix offers many standard build systems as pre-defined `build-system` objects. They represent common ways to build software. The `build-system` field of the package contains the system to be used for the package. Depending on its value, a different sequence of commands will be executed inside the build environment. In our case, we use the `cmake-build-system`.

Sometimes modifications must be made to that idealized build process. For this purpose, it is divided into build phases, which may be edited in the Scheme. This is necessary in our case as well. In particular, the source code of `lc3-target` needs to be moved to the `Target` directory. This can be done by modifying the `configure` phase of the build system. In this phase, all the `origins` have already been fetched into the build environment. Before the phase is executed, we insert a hook that copies the `lc3-target` code to the correct location.

3.3. Adding to a Channel

Adding a package to a channel only involves adding the Scheme file to the git repository. It is also necessary to adjust the module definition at the top of the file to reference the channel.

If, like in our case, the build process involves patches from `.patch` files, they must also be added to the repository. Patches are found during building using `search-pathes` from the channels root, so the path of the patch files referenced in the Scheme must also be adjusted to be relative from there.

Finally, for Guix to recognize this channel, it is added to the user's channels, and `guix pull` needs to be executed to make packages from the channel available to build.

4. Evaluation

After the process of making a software available as a package in Guix, we are now able to discuss whether this has brought us closer to the goal of reproducibility.

4.1. Reproducibility

When trying to get the compilation to finish successfully during the first, exploratory stage (3.1.3), we encounter a roadblock that has an implication for whether the package builds deterministically. With the most recent version of gcc available in Guix at the time of writing, version 10.3.0, compilation fails due to an implicit cast. With an earlier version such as 5.5.0, the compilation succeeds.

In the Scheme defining the package, it is not required to specify the version that should be used. The following line implicitly includes the most recent version of the `commencement` package, which puts the most recent version of the gcc toolchain into the build environment.

```
#:use-module (gnu packages commencement)
```

It is therefore possible to define a package in a Scheme for Guix, that builds and installs correctly when the most recent version of gcc accepts the implicit cast. However, when a newer version of gcc treats this as an error, the build fails.

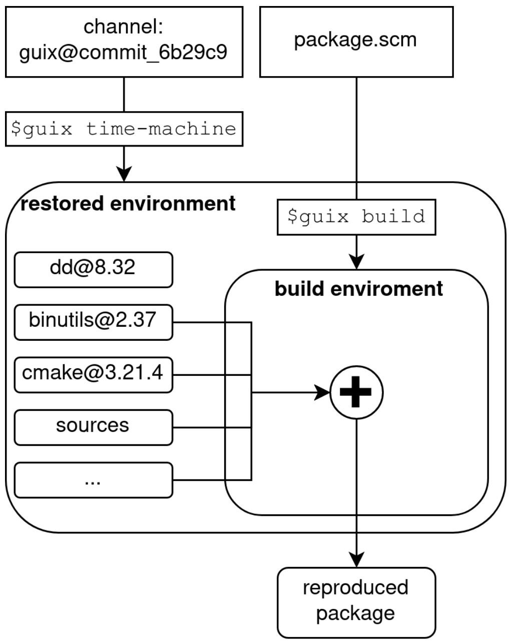

Figure 1: The same built package is reproduced from just the channel state and the package Scheme.

It is clear from this example that simply defining a package in a Scheme is not enough to make it reproducible. To actually compile with reproducibly, the packager's current Guix environment needs to be described. This includes the versions of all input packages.

For reproducing sofware, the environment must be recreated, and the package built within it. Thankfully, Guix includes a utility to do just this, called `guix time-machine` [4]. A conceptual overview of its use for this purpose is illustrated in figure 1.

To summarize, reaching the same build result each time is the result of two of Guix's features. First, with help of the Scheme package format, `guix build` is able to determine all transitive inputs of the software, down to the operating system. Second, the state of the Guix environment, including the packages at the version the software was built at, can be recreated with the `guix time-machine`. The time machine only requires the description of Guix's channels at that time for this, i.e. a git commit (`6b29c9` in this example), since channels are git repositories.

4.2. Complexity

It is a goal of the Scheme format for packages to be "purely declarative in common cases", so as to be "directly usable by packagers with little or no experience with Scheme." [1]. For such common cases, a Scheme can intuitively be constructed with help from the examples in the Guix cookbook. However, for non-obvious cases, such as ours, where a git repository needs to be fetched and copied to a specific location, the documentation does not provide an easy recipe. In cases such as these, it is very useful to browse the default channel's other package definitions. It is likely that the special case has been dealt with by another packager in the past. Guix's package still appear less complex than Nix's though, as can be seen by comparing the definitons for the same packages, like `llvm`.

To aid with debugging during the definition of the Scheme, the command `guix build` with the `--keep-failed` switch is useful. It allows building a package without a channel and also keeps the build environment to examine, which helps determine the cause of failures.

The most challenging part of our example is to determine the informal building steps and dependencies before they are translated into a Scheme. Because the developer of `lc3-target` has not provided any information on the llvm version or modifications to llvm's source code, a process of trial-and-error is necessary.

5. Related work

Guix has been used previously in scientific work. In their paper from 2015, Courtés and Wurmus explore the use of Guix in high-performance computing [6]. They show that its structure, with unprivileged users being able to build packages through calls to the user daemon, benefits environments where many users share a cluster. It is also useful because Guix packages do not depend on parts of the host system, making moving them to new clusters a less involved process. With Guix, it is also more straightforward for them to define variants of software for different use cases. In a specific example, they define different variants for a package for developers and users.

There is also use for Guix in bioinformatics. In 2018, Wurmus et al. introduce PiGx, a tool for developing pipelines that transform raw experimental data into reports [7]. PiGx is packaged using Guix, which makes it not only reproducible but also transparent. The specific dependencies and packages used can be easily described and analyzed, which would more complex if a virtual machine is used to recreate the software environment.

Recently, in 2022, work has also been done by Batten et al. as part of the CARRV workshop to use Guix for packaging RISC-V software, as well as hardware simulators like gem5 [8]. RISC-V software is mostly cross-compiled as of yet, posing the problem that toolchains for it need to be defined. This again has many complex inputs, making packaging with Guix suitable.

6. Conclusion

In this paper, we have discussed the internals and mechanisms of the Guix package manager, and what makes its approach useful for reproducibility in contrast to other package managers. Furthermore, we have shown in a concrete example the process of packaging software, and described in detail the process of learning the dependencies of largely undocumented software. We aim to make this package available in the channel `guix-past`, which is managed by France's National Institute for Research in Digital Science and Technology (INRIA). Furthermore, we have provided a recipe for using the `guix time-machine` to recreate the package environment to build a piece of software with the same inputs each time, making it reproducible.

References

[1] L. Courtès, “Functional Package Management with Guix,” *CoRR*, vol. abs/1305.4584, 2013. [Online]. Available: http://arxiv.org/abs/1305.4584

[2] *chroot(2) - Linux man page*, Free Software Foundation. [Online]. Available: https://linux.die.net/man/2/chroot

[3] E. Dolstra and A. Löh, “NixOS: A Purely Functional Linux Distribution,” in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’08. New York, NY, USA: Association for Computing Machinery, 2008, p. 367–378. [Online]. Available: https://doi.org/10.1145/1411204.1411255

[4] K. Hinsen, “Reproducible computations with Guix,” 2020. [Online]. Available: https://guix.gnu.org/en/blog/2020/reproducible-computations-with-guix/

[5] R. Wurmus, E. Flashner, P. Neidhardt, O. Pykhalov, M. Brooks, M. Karpezo, B. Waegeneire, A. Batista, C. Lemmer-Webber, J. Branson, M. Couroyer, and L. Courté, “GNU Guix Cookbook,” 2019. [Online]. Available: https://guix.gnu.org/en/cookbook/en/

[6] L. Courtès and R. Wurmus, “Reproducible and User-Controlled Software Environments in HPC with Guix,” in *2nd International Workshop on Reproducibility in Parallel Computing (RepPar)*, Vienne, Austria, Aug. 2015. [Online]. Available: https://hal.inria.fr/hal-01161771

[7] R. Wurmus, B. Uyar, B. Osberg, V. Franke, A. Gosdschan, K. Wreczycka, J. Ronen, and A. Akalin, “PiGx: reproducible genomics analysis pipelines with GNU Guix,” *GigaScience*, vol. 7, no. 12, 10 2018, giy123. [Online]. Available: https://doi.org/10.1093/gigascience/giy123

[8] C. Batten, P. Prins, E. Flashner, A. Isaac, J. Nieuwenhuizen, E. Zarraga, T. Ta, A. Rovinski, and E. Garrison, “The Case for Using Guix to Enable Reproducible RISC-V Software & Hardware,” 2022.