Functional Package Management with Guix

Ludovic Courtès

Functional Package Management with Guix

Ludovic Courtès
Bordeaux, France
ludo@gnu.org

ABSTRACT

We describe the design and implementation of GNU Guix, a purely functional package manager designed to support a complete GNU/Linux distribution. Guix supports transactional upgrades and roll-backs, unprivileged package management, per-user profiles, and garbage collection. It builds upon the low-level build and deployment layer of the Nix package manager. Guix uses Scheme as its programming interface. In particular, we devise an embedded domain-specific language (EDSL) to describe and compose packages. We demonstrate how it allows us to benefit from the host general-purpose programming language while not compromising on expressiveness. Second, we show the use of Scheme to write build programs, leading to a "two-tier" programming system.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.4.5 [**Operating Systems**]: System Programs and Utilities; D.1.1 [**Software**]: Applicative (Functional) Programming

General Terms

Languages, Management, Reliability

Keywords

Functional package management, Scheme, Embedded domain-specific language

European Lisp Symposium 2013, Madrid, Spain

1. INTRODUCTION

GNU Guix[1] is a *purely functional* package manager for the GNU system [20], and in particular GNU/Linux. Package management consists in all the activities that relate to building packages from source, honoring the build-time and run-time dependencies on packages, installing, removing, and upgrading packages in user environments. In addition to these standard features, Guix supports transactional upgrades and roll-backs, unprivileged package management, per-user profiles, and garbage collection. Guix comes with a distribution of user-land free software packages.

Guix seeks to empower users in several ways: by offering the uncommon features listed above, by providing the tools that allow users to formally correlate a binary package and the "recipes" and source code that led to it—furthering the spirit of the GNU General Public License—, by allowing them to customize the distribution, and by lowering the barrier to entry in distribution development.

The keys toward these goals are the implementation of a purely functional package management paradigm, and the use of both declarative and lower-level programming interfaces (APIs) embedded in Scheme. To that end, Guix reuses the package storage and deployment model implemented by the Nix functional package manager [8]. On top of that, it provides Scheme APIs, and in particular embedded domain-specific languages (EDSLs) to describe software packages and their build system. Guix also uses Scheme for programs and libraries that implement the actual package build processes, leading to a "two-tier" system.

This paper focuses on the programming techniques implemented by Guix. Our contribution is twofold: we demonstrate that use of Scheme and EDSLs achieves expressiveness comparable to that of Nix's DSL while providing a richer and extensible programming environment; we further show that Scheme is a profitable alternative to shell tools when it comes to package build programs. Section 2 first gives some background on functional package management and its implementation in Nix. Section 3 describes the design and implementation of Guix's programming and packaging interfaces. Section 4 provides an evaluation and discussion of the current status of Guix. Section 5 presents related work, and Section 6 concludes.

[1] `http://www.gnu.org/software/guix/`

2. BACKGROUND

This section describes the functional package management paradigm and its implementation in Nix. It then shows how Guix differs, and what the rationale is.

2.1 Functional Package Management

Functional package management is a paradigm whereby the build and installation process of a package is considered as a pure function, without any side effects. This is in contrast with widespread approaches to package build and installation where the build process usually has access to all the software installed on the machine, regardless of what its declared inputs are, and where installation modifies files in place.

Functional package management was pioneered by the Nix package manager [8], which has since matured to the point of managing a complete GNU/Linux distribution [9]. To allow build processes to be faithfully regarded as pure functions, Nix can run them in a `chroot` environment that only contains the inputs it explicitly declared; thus, it becomes impossible for a build process to use, say, Perl, if that package was not explicitly declared as an input of the build process. In addition, Nix maps the list of inputs of a build process to a statistically unique file system name; that file name is used to identify the output of the build process. For instance, a particular build of GNU Emacs may be installed in `/nix/store/-v9zic07iar8w90zcy398r745w78a7lqs-emacs-24.2`, based on a cryptographic hash of all the inputs to that build process; changing the compiler, configuration options, build scripts, or any other inputs to the build process of Emacs yields a different name. This is a form of *on-disk memoization*, with the `/nix/store` directory acting as a cache of "function results"—i.e., a cache of installed packages. Directories under `/nix/store` are immutable.

This direct mapping from build inputs to the result's directory name is basis of the most important properties of a functional package manager. It means that build processes are regarded as *referentially transparent*. To put it differently, instead of merely providing pre-built binaries and/or build recipes, functional package managers provide binaries, build recipes, and in effect a *guarantee* that a given binary matches a given build recipe.

2.2 Nix

The idea of *purely functional* package started by making an analogy between programming language paradigms and software deployment techniques [8]. The authors observed that, in essence, package management tools typically used on free operating systems, such as RPM and Debian's APT, implement an *imperative* software deployment paradigm. Package installation, removal, and upgrade are all done in-place, by mutating the operating system's state. Likewise, changes to the operating system's configuration are done in-place by changing configuration files.

This imperative approach has several drawbacks. First, it makes it hard to reproduce or otherwise describe the OS state. Knowing the list of installed packages and their version is not enough, because the installation procedure of packages may trigger hooks to change global system configuration files [4, 7], and of course users may have done additional modifications. Second, installation, removal, and upgrade are not transactional; interrupting them may leave the system in an undefined, or even unusable state, where some of the files have been altered. Third, rolling back to a previous system configuration is practically impossible, due to the absence of a mechanism to formally describe the system's configuration.

Nix attempts to address these shortcomings through the functional software deployment paradigm: installed packages are immutable, and build processes are regarded as pure functions, as explained before. Thanks to this property, it implements *transparent source/binary deployment*: the directory name of a build result encodes all the inputs of its build process, so if a trusted server provides that directory, then it can be directly downloaded from there, avoiding the need for a local build.

Each user has their own *profile*, which contains symbolic links to the `/nix/store` directories of installed packages. Thus, users can install packages independently, and the actual storage is shared when several users install the very same package in their profile. Nix comes with a *garbage collector*, which has two main functions: with conservative scanning, it can determine what packages a build output refers to; and upon user request, it can delete any packages not referenced *via* any user profile.

To describe and compose build processes, Nix implements its own domain-specific language (DSL), which provides a convenient interface to the build and storage mechanisms described above. The Nix language is purely functional, lazy, and dynamically typed; it is similar to that of the Vesta software configuration system [11]. It comes with a handful of built-in data types, and around 50 primitives. The primitive to describe a build process is `derivation`.

```
1:  derivation {
2:    name = "example-1.0";
3:    builder = "${./static-bash}";
4:    args = [ "-c" "echo hello > $out" ];
5:    system = "x86_64-linux";
6:  }
```

Figure 1: Call to the `derivation` primitive in the Nix language.

Figure 1 shows code that calls the `derivation` function with one argument, which is a dictionary. It expects at least the four key/value pairs shown above; together, they define the build process and its inputs. The result is a *derivation*, which is essentially the *promise* of a build. The derivation has a low-level on-disk representation independent of the Nix language—in other words, derivations are to the Nix language what assembly is to higher-level programming languages. When this derivation is instantiated—i.e., built—, it runs the command `static-bash -c "echo hello > $out"` in a chroot that contains nothing but the `static-bash` file; in addition, each key/value pair of the `derivation` argument is reified in the build process as an environment variable, and the `out` environment variable is defined to point to the output `/nix/store` file name.

Before the build starts, the file `static-bash` is imported under `/nix/store/...-static-bash`, and the value associated

with `builder` is substituted with that file name. This `${...}` form on line 3 for string interpolation makes it easy to insert Nix-language values, and in particular computed file names, in the contents of build scripts. The Nix-based GNU/Linux distribution, NixOS, has most of its build scripts written in Bash, and makes heavy use of string interpolation on the Nix-language side.

All the files referenced by derivations live under `/nix/store`, called *the store*. In a multi-user setup, users have read-only access to the store, and all other accesses to the store are mediated by a daemon running as `root`. Operations such as importing files in the store, computing a derivation, building a derivation, or running the garbage collector are all implemented as remote procedure calls (RPCs) to the daemon. This guarantees that the store is kept in a consistent state—e.g., that referenced files and directories are not garbage-collected, and that the contents of files and directories are genuine build results of the inputs hashed in their name.

The implementation of the Nix language is an interpreter written in C++. In terms of performance, it does not compete with typical general-purpose language implementations; that is often not a problem given its specific use case, but sometimes requires rewriting functions, such as list-processing tools, as language primitives in C++. The language itself is not extensible: it has no macros, a fixed set of data types, and no foreign function interface.

2.3 From Nix to Guix

Our main contribution with GNU Guix is the use of Scheme for both the composition and description of build processes, and the implementation of build scripts. In other words, Guix builds upon the build and deployment primitives of Nix, but replaces the Nix language by Scheme with embedded domain-specific languages (EDSLs), and promotes Scheme as a replacement for Bash in build scripts. Guix is implemented using GNU Guile 2.0[2], a rich implementation of Scheme based on a compiler and bytecode interpreter that supports the R5RS and R6RS standards. It reuses the build primitives of Nix by making remote procedure calls (RPCs) to the Nix build daemon.

We claim that using an *embedded* DSL has numerous practical benefits over an independent DSL: tooling (use of Guile's compiler, debugger, and REPL, Unicode support, etc.), libraries (SRFIs, internationalization support, etc.), and seamless integration in larger programs. To illustrate this last point, consider an application that traverses the list of available packages and processes it—for instance to filter packages whose name matches a pattern, or to render it as HTML. A Scheme program can readily and efficiently do it with Guix, where packages are first-class Scheme objects; conversely, writing such an implementation with an external DSL such as Nix requires either extending the language implementation with the necessary functionality, or interfacing with it *via* an external representation such as XML, which is often inefficient and lossy.

We show that use of Scheme in build scripts is natural, and can achieve conciseness comparable to that of shell scripts, but with improved expressivity and clearer semantics.

The next section describes the main programming interfaces of Guix, with a focus on its high-level package description language and "shell programming" substitutes provided to builder-side code.

[2] `http://www.gnu.org/software/guile/`

3. BUILD EXPRESSIONS AND PACKAGE DESCRIPTIONS

Our goal when designing Guix was to provide interfaces ranging from Nix's low-level primitives such as `derivation` to high-level package declarations. The declarative interface is a requirement to help grow and maintain a large software distribution. This section describes the three level of abstractions implemented in Guix, and illustrates how Scheme's homoiconicity and extensibility were instrumental.

3.1 Low-Level Store Operations

As seen above, *derivations* are the central concept in Nix. A derivation bundles together a *builder* and its execution environment: command-line arguments, environment variable definitions, as well as a list of input derivations whose result should be accessible to the builder. Builders are typically executed in a `chroot` environment where only those inputs explicitly listed are visible. Guix transposes Nix's `derivation` primitive literally to its Scheme interface.

```
 1: (let* ((store (open-connection))
 2:        (bash  (add-to-store store "static-bash"
 3:                             #t "sha256"
 4:                             "./static-bash")))
 5:   (derivation store "example-1.0"
 6:               "x86_64-linux"
 7:               bash
 8:               '("-c" "echo hello > $out")
 9:               '() '()))
10:
11: ⇒
12: "/nix/store/nsswy...-example-1.0.drv"
13: #<derivation "example-1.0" ...>
```

Figure 2: Using the `derivation` primitive in Scheme with Guix.

Figure 2 shows the example of Figure 1 rewritten to use Guix's low-level Scheme API. Notice how the former makes explicit several operations not visible in the latter. First, line 1 establishes a connection to the build daemon; line 2 explicitly asks the daemon to "intern" file `static-bash` into the store; finally, the `derivation` call instructs the daemon to compute the given derivation. The two arguments on line 9 are a set of environment variable definitions to be set in the build environment (here, it's just the empty list), and a set of *inputs*—other derivations depended on, and whose result must be available to the build process. Two values are returned (line 11): the file name of the on-disk representation of the derivation, and its in-memory representation as a Scheme record.

The build actions represented by this derivation can then be performed by passing it to the `build-derivations` RPC.

Again, its build result is a single file reading `hello`, and its build is performed in an environment where the only visible file is a copy of `static-bash` under `/nix/store`.

3.2 Build Expressions

The Nix language heavily relies on string interpolation to allow users to insert references to build results, while hiding the underlying `add-to-store` or `build-derivations` operations that appear explicitly in Figure 2. Scheme has no support for string interpolation; adding it to the underlying Scheme implementation is certainly feasible, but it's also unnatural.

The obvious strategy here is to instead leverage Scheme's homoiconicity. This leads us to the definition of `build-expression->derivation`, which works similarly to `derivation`, except that it expects a *build expression* as an S-expression instead of a builder. Figure 3 shows the same derivation as before but rewritten to use this new interface.

```
 1: (let ((store   (open-connection))
 2:       (builder '(call-with-output-file %output
 3:                   (lambda ()
 4:                     (display "hello")))))
 5:   (build-expression->derivation store
 6:                                 "example-1.0"
 7:                                 "x86_64-linux"
 8:                                 builder '()))
 9:
10: ⇒
11: "/nix/store/zv3b3...-example-1.0.drv"
12: #<derivation "example-1.0" ...>
```

Figure 3: Build expression written in Scheme.

This time the builder on line 2 is purely a Scheme expression. That expression will be evaluated when the derivation is built, in the specified build environment with no inputs. The environment implicitly includes a copy of Guile, which is used to evaluate the `builder` expression. By default this is a stand-alone, statically-linked Guile, but users can also specify a derivation denoting a different Guile variant.

Remember that this expression is run by a separate Guile process than the one that calls `build-expression->derivation`: it is run by a Guile process launched by the build daemon, in a `chroot`. So, while there is a single language for both the "host" and the "build" side, there are really two *strata* of code, or *tiers*: the host-side, and the build-side code[3].

Notice how the output file name is reified *via* the `%output` variable automatically added to `builder`'s scope. Input file names are similarly reified through the `%build-inputs` variable (not shown here). Both variables are non-hygienically introduced in the build expression by `build-expression->derivation`.

Sometimes the build expression needs to use functionality from other modules. For modules that come with Guile, the expression just needs to be augmented with the needed `(use-modules ...)` clause. Conversely, external modules first need to be imported into the derivation's build environment so the build expression can use them. To that end, the `build-expression->derivation` procedure has an optional `#:modules` keyword parameter, allowing additional modules to be imported into the expression's environment.

When `#:modules` specifies a non-empty module list, an auxiliary derivation is created and added as an input to the initial derivation. That auxiliary derivation copies the module source and compiled files in the store. This mechanism allows build expressions to easily use helper modules, as described in Section 3.4.

3.3 Package Declarations

The interfaces described above remain fairly low-level. In particular, they explicitly manipulate the store, pass around the system type, and are very distant from the abstract notion of a software package that we want to focus on. To address this, Guix provides a high-level package definition interface. It is designed to be *purely declarative* in common cases, while allowing users to customize the underlying build process. That way, it should be intelligible and directly usable by packagers will little or no experience with Scheme. As an additional constraint, this extra layer should be efficient in space and time: package management tools need to be able to load and traverse a distribution consisting of thousands of packages.

Figure 4 shows the definition of the GNU Hello package, a typical GNU package written in C and using the GNU build system—i.e., a `configure` script that generates a makefile supporting standardized targets such as `check` and `install`. It is a direct mapping of the abstract notion of a software package and should be rather self-descriptive.

The `inputs` field specifies additional dependencies of the package. Here line 16 means that Hello has a dependency labeled `"gawk"` on GNU Awk, whose value is that of the `gawk` global variable; `gawk` is bound to a similar `package` declaration, omitted for conciseness.

The `arguments` field specifies arguments to be passed to the build system. Here `#:configure-flags`, unsurprisingly, specifies flags for the `configure` script. Its value is quoted because it will be evaluated in the build stratum—i.e., in the build process, when the derivation is built. It refers to the `%build-inputs` global variable introduced in the build stratum by `build-expression->derivation`, as seen before. That variable is bound to an association list that maps input names, like `"gawk"`, to their actual directory name on disk, like `/nix/store/...-gawk-4.0.2`.

The code in Figure 4 demonstrates Guix's use of embedded domain-specific languages (EDSLs). The `package` form, the `origin` form (line 5), and the `base32` form (line 9) are expanded at macro-expansion time. The `package` and `origin` forms expand to a call to Guile's `make-struct` primitive, which instantiates a record of the given type and with the given field values[4]; these macros look up the mapping of

[3]The term "stratum" is this context was coined by Manuel Serrano et al. for their work on Hop where a similar situation arises [17].

[4]The `make-struct` instantiates SRFI-9-style flat records,

```
 1: (define hello
 2:   (package
 3:     (name "hello")
 4:     (version "2.8")
 5:     (source (origin
 6:               (method url-fetch)
 7:               (uri (string-append "mirror://gnu/hello/hello-"
 8:                                   version ".tar.gz"))
 9:               (sha256 (base32 "0wqd8...")))))
10:     (build-system gnu-build-system)
11:     (arguments
12:       '(#:configure-flags
13:         `("-disable-color"
14:           ,(string-append "-with-gawk="
15:                           (assoc-ref %build-inputs "gawk")))))
16:     (inputs `(("gawk" ,gawk)))
17:     (synopsis "GNU Hello")
18:     (description "An illustration of GNU's engineering practices.")
19:     (home-page "http://www.gnu.org/software/hello/")
20:     (license gpl3+)))
```

Figure 4: A package definition using the high-level interface.

field names to field indexes, such that that mapping incurs no run-time overhead, in a way similar to SRFI-35 records [14]. They also bind fields as per `letrec*`, allowing them to refer to one another, as on line 8 of Figure 4. The `base32` macro simply converts a literal string containing a base-32 representation into a bytevector literal, again allowing the conversion and error-checking to be done at expansion time rather than at run-time.

```
 1: (define-record-type* <package>
 2:   package make-package
 3:   package?
 4:
 5:   (name package-name)
 6:   (version package-version)
 7:   (source package-source)
 8:   (build-system package-build-system)
 9:   (arguments package-arguments
10:              (default '()) (thunked))
11:
12:   (inputs package-inputs
13:           (default '()) (thunked))
14:   (propagated-inputs package-propagated-inputs
15:                      (default '()))
16:
17:   (synopsis package-synopsis)
18:   (description package-description)
19:   (license package-license)
20:   (home-page package-home-page)
21:
22:   (location package-location
23:      (default (current-source-location))))
```

Figure 5: Definition of the `package` record type.

which are essentially vectors of a disjoint type. In Guile they are lightweight compared to CLOS-style objects, both in terms of run time and memory footprint. Furthermore, `make-struct` is subject to inlining.

The `package` and `origin` macros are generated by a `syntax-case` hygienic macro [19], `define-record-type*`, which is layered above SRFI-9's syntactic record layer [13]. Figure 5 shows the definition of the <`package`> record type (the <`origin`> record type, not shown here, is defined similarly.) In addition to the name of a procedural constructor, `make-package`, as with SRFI-9, the name of a *syntactic* constructor, `package`, is given (likewise, `origin` is the syntactic constructor of <`origin`>.) Fields may have a default value, introduced with the `default` keyword. An interesting use of default values is the `location` field: its default value is the result of `current-source-location`, which is itself a built-in macro that expands to the source file location of the `package` form. Thus, records defined with the `package` macro automatically have a `location` field denoting their source file location. This allows the user interface to report source file location in error messages and in package search results, thereby making it easier for users to "jump into" the distribution's source, which is one of our goals.

```
1: (package (inherit hello)
2:   (version "2.7")
3:   (source
4:     (origin
5:       (method url-fetch)
6:       (uri
7:        "mirror://gnu/hello/hello-2.7.tar.gz")
8:       (sha256
9:         (base32 "7dqw3...")))))
```

Figure 6: Creating a variant of the `hello` package.

The syntactic constructors generated by `define-record-type*` additionally support a form of *functional setters* (sometimes referred to as "lenses" [15]), *via* the `inherit` keyword. It allows programmers to create new instances that differ from an existing instance by one or more field values. A typical use case is shown in Figure 6: the expression shown evaluates to a new <`package`> instance whose fields all have

the same value as the `hello` variable of Figure 4, except for the `version` and `source` fields. Under the hood, again, this expands to a single `make-struct` call with `struct-ref` calls for fields whose value is reused.

The `inherit` feature supports a very useful idiom. It allows new package variants to be created programmatically, concisely, and in a purely functional way. It is notably used to bootstrap the software distribution, where bootstrap variants of packages such as GCC or the GNU libc are built with different inputs and configuration flags than the final versions. Users can similarly define customized variants of the packages found in the distribution. This feature also allows high-level transformations to be implemented as pure functions. For instance, the `static-package` procedure takes a <`package`> instance, and returns a variant of that package that is statically linked. It operates by just adding the relevant `configure` flags, and recursively applying itself to the package's inputs.

Another application is the *on-line auto-updater*: when installing a GNU package defined in the distribution, the `guix package` command automatically checks whether a newer version is available upstream from `ftp.gnu.org`, and offers the option to substitute the package's source with a fresh download of the new upstream version—all at run time.This kind of feature is hardly accessible to an external DSL implementation. Among other things, this feature requires networking primitives (for the FTP client), which are typically unavailable in an external DSL such as the Nix language. The feature could be implemented in a language other than the DSL—for instance, Nix can export its abstract syntax tree as XML to external programs. However, this approach is often inefficient, due to the format conversion, and lossy: the exported representation may be either be too distant from the source code, or too distant from the preferred abstraction level. The author's own experience writing an off-line auto-updater for Nix revealed other specific issues; for instance, the Nix language is lazily evaluated, but to make use of its XML output, one has to force strict evaluation, which in turn may generate more data than needed. In Guix, <`package`> instances have the expected level of abstraction, and they are readily accessible as first-class Scheme objects.

Sometimes it is desirable for the value of a field to depend on the system type targeted. For instance, for bootstrapping purposes, MIT/GNU Scheme's build system depends on pre-compiled binaries, which are architecture-dependent; its `input` field must be able to select the right binaries depending on the architecture. To allow field values to refer to the target system type, we resort to *thunked* fields, as shown on line 13 of Figure 5. These fields have their value automatically wrapped in a thunk (a zero-argument procedure); when accessing them with the associated accessor, the thunk is transparently invoked. Thus, the values of thunked fields are computed lazily; more to the point, they can refer to *dynamic state* in place at their invocation point. In particular, the `package-derivation` procedure (shortly introduced) sets up a `current-system` dynamically-scoped parameter, which allows field values to know what the target system is.

Finally, both <`package`> and <`origin`> records have an associated "compiler" that turns them into a derivation. `origin-derivation` takes an <`origin`> instance and returns a derivation that downloads it, according to its `method` field. Likewise, `package-derivation` takes a package and returns a derivation that builds it, according to its `build-system` and associated `arguments` (more on that in Section 3.4). As we have seen on Figure 4, the `inputs` field lists dependencies of a package, which are themselves <`package`> objects; the `package-derivation` procedure recursively applies to those inputs, such that their derivation is computed and passed as the inputs argument of the lower-level `build-expression->derivation`.

Guix essentially implements *deep embedding* of DSLs, where the semantics of the packaging DSL is interpreted by a dedicated compiler [12]. Of course the DSLs defined here are simple, but they illustrate how Scheme's primitive mechanisms, in particular macros, make it easy to implement such DSLs without requiring any special support from the Scheme implementation.

3.4 Build Programs

The value of the `build-system` field, as shown on Figure 4, must be a `build-system` object, which is essentially a wrapper around two procedure: one procedure to do a native build, and one to do a cross-build. When the aforementioned `package-derivation` (or `package-cross-derivation`, when cross-building) is called, it invokes the build system's build procedure, passing it a connection to the build daemon, the system type, derivation name, and inputs. It is the build system's responsibility to return a derivation that actually builds the software.

```
(define* (gnu-build #:key (phases %standard-phases)
                    #:allow-other-keys
                    #:rest args)
  ;; Run all the PHASES in order, passing them ARGS.
  ;; Return true on success.
  (every (match-lambda
          ((name . proc)
           (format #t "starting phase `~a'~%" name)
           (let ((result (apply proc args)))
             (format #t "phase `~a' done~%" name)
             result)))
         phases))
```

Figure 7: Entry point of the builder side code of `gnu-build-system`.

The `gnu-build-system` object (line 10 of Figure 4) provides procedures to build and cross-build software that uses the GNU build system or similar. In a nutshell, it runs the following phases by default:

1. unpack the source tarball, and change the current directory to the resulting directory;
2. patch shebangs on installed files—e.g., replace `#!/bin/sh` by `#!/nix/store/...-bash-4.2/bin/sh`; this is required to allow scripts to work with our unusual file system layout;
3. run `./configure --prefix=/nix/store/...`, followed by `make` and `make check`

4. run `make install` and patch shebangs in installed files.

Of course, that is all implemented in Scheme, *via* `build-expression->derivation`. Supporting code is available as a build-side module that `gnu-build-system` automatically adds as an input to its build scripts. The default build programs just call the procedure of that module that runs the above phases.

The `(guix build gnu-build-system)` module contains the implementation of the above phases; it is imported on the builder side. The phases are modeled as follows: each phase is a procedure accepting several keyword arguments, and ignoring any keyword arguments it does not recognize[5]. For instance the `configure` procedure is in charge of running the package's `./configure` script; that procedure honors the `#:configure-flags` keyword parameter seen on Figure 4. Similarly, the `build`, `check`, and `install` procedures run the `make` command, and all honor the `#:make-flags` keyword parameter.

All the procedures implementing the standard phases of the GNU build system are listed in the `%standard-phases` builder-side variable, in the form of a list of phase name-/procedure pairs. The entry point of the builder-side code of `gnu-build-system` is shown on Figure 7. It calls all the phase procedures in order, by default those listed in the `%standard-phases` association list, passing them all the arguments it got; its return value is true when every procedure's return value is true.

```
(define howdy
  (package (inherit hello)
    (arguments
      '(#:phases
        (alist-cons-after
          'configure 'change-hello
          (lambda* (#:key system #:allow-other-keys)
            (substitute* "src/hello.c"
              (("Hello, world!")
               (string-append "Howdy! Running on "
                              system "."))))
          %standard-phases)))))
```

Figure 8: Package specification with custom build phases.

The `arguments` field, shown on Figure 4, allows users to pass keyword arguments to the builder-side code. In addition to the `#:configure-flags` argument shown on the figure, users may use the `#:phases` argument to specify a different set of phases. The value of the `#:phases` must be a list of phase name/procedure pairs, as discussed above. This allows users to arbitrarily extend or modify the behavior of the build system. Figure 8 shows a variant of the definition in Figure 4 that adds a custom build phase. The `alist-cons-after` procedure is used to add a pair with `change-hello` as its first item and the `lambda*` as its second item right after the pair in `%standard-phases` whose first item is `configure`; in other words, it reuses the standard build phases, but with an additional `change-hello` phase right after the `configure` phase. The whole `alist-cons-after` expression is evaluated on the builder side.

This approach was inspired by that of NixOS, which uses Bash for its build scripts. Even with "advanced" Bash features such as functions, arrays, and associative arrays, the phases mechanism in NixOS remains limited and fragile, often leading to string escaping issues and obscure error reports due to the use of `eval`. Again, using Scheme instead of Bash unsurprisingly allows for better code structuring, and improves flexibility.

Other build systems are provided. For instance, the standard build procedure for Perl packages is slightly different: mainly, the configuration phase consists in running `perl Makefile.PL`, and test suites are run with `make test` instead of `make check`. To accommodate that, Guix provides `perl-build-system`. Its companion build-side module essentially calls out to that of `gnu-build-system`, only with appropriate `configure` and `check` phases. This mechanism is similarly used for other build systems such as CMake and Python's build system.

```
(substitute* (find-files "gcc/config"
                         "^gnu-user(64)?\\.h$")
  (("#define LIB_SPEC (.*)$" _ suffix)
   (string-append "#define LIB_SPEC \"-L" libc
                  "/lib \" " suffix "\n"))
  (("#define STARTFILE_SPEC.*$" line)
   (string-append "#define STARTFILE_PREFIX_1 \""
                  libc "/lib\"\n" line)))
```

Figure 9: The `substitute*` macro for `sed`-like substitutions.

Build programs often need to traverse file trees, modify files according to a given pattern, etc. One example is the "patch shebang" phase mentioned above: all the source files must be traversed, and those starting with `#!` are candidate to patching. This kind of task is usually associated with "shell programming"—as is the case with the build scripts found in NixOS, which are written in Bash, and resort to `sed`, `find`, etc. In Guix, a build-side Scheme module provides the necessary tools, built on top of Guile's operating system interface. For instance, `find-files` returns a list of files whose names matches a given pattern; `patch-shebang` performs the `#!` adjustment described above; `copy-recursively` and `delete-recursively` are the equivalent, respectively, of the shell `cp -r` and `rm -rf` commands; etc.

An interesting example is the `substitute*` macro, which does `sed`-style substitution on files. Figure 9 illustrates its use to patch a series of files returned by `find-files`. There are two clauses, each with a pattern in the form of a POSIX regular expression; each clause's body returns a string, which is the substitution for any matching line in the given files. In the first clause's body, `suffix` is bound to the submatch corresponding to `(.*)` in the regexp; in the second clause, `line` is bound to the whole match for that regexp. This snippet is nearly as concise than equivalent shell code using

[5]Like many Scheme implementations, Guile supports *named* or *keyword* arguments as an extension to the R5 and R6RS. In addition, procedure definitions whose formal argument list contains the `#:allow-other-keys` keyword ignore any unrecognized keyword arguments that they are passed.

`find` and `sed`, and it is much easier to work with.

Build-side modules also include support for fetching files over HTTP (using Guile's web client module) and FTP, as needed to realize the derivation of `origin`s (line 5 of Figure 4). TLS support is available when needed through the Guile bindings of the GnuTLS library.

4. EVALUATION AND DISCUSSION

This section discusses the current status of Guix and its associated GNU/Linux distribution, and outlines key aspects of their development.

4.1 Status

Guix is still a young project. Its main features as a package manager are already available. This includes the APIs discussed in Section 3, as well as command-line interfaces. The development of Guix's interfaces was facilitated by the reuse of Nix's build daemon as the storage and deployment layer.

The `guix package` command is the main user interface: it allows packages to be browsed, installed, removed, and upgraded. The command takes care of maintaining meta-data about installed packages, as well as a per-user tree of symlinks pointing to the actual package files in `/nix/store`, called the *user profile*. It has a simple interface. For instance, the following command installs Guile and removes Bigloo from the user's profile, as a single transaction:

```
$ guix package --install guile --remove bigloo
```

The transaction can be rolled back with the following command:

```
$ guix package --roll-back
```

The following command upgrades all the installed packages whose name starts with a 'g':

```
$ guix package --upgrade '^g.*'
```

The `--list-installed` and `--list-available` options can be used to list the installed or available packages.

As of this writing, Guix comes with a user-land distribution of GNU/Linux. That is, it allows users to install packages on top of a running GNU/Linux system. The distribution is self-contained, as explained in Section 4.3, and available on `x86_64` and `i686`. It provides more than 400 packages, including core GNU packages such as the GNU C Library, GCC, Binutils, and Coreutils, as well as the Xorg software stack and applications such as Emacs, TeX Live, and several Scheme implementations. This is roughly a tenth of the number of packages found in mature free software distributions such as Debian. Experience with NixOS suggests that the functional model, coupled with continuous integration, allows the distribution to grow relatively quickly, because it is always possible to precisely monitor the status of the whole distribution and the effect of a change—unlike with imperative distributions, where the upgrade of a single package can affect many applications in many unpredictable ways [7].

From a programming point of view, packages are exposed as first-class global variables. For instance, the `(gnu packages guile)` module exports two variables, `guile-1.8` and `guile-2.0`, each bound to a <package> variable corresponding to the legacy and current stable series of Guile. In turn, this module imports `(gnu packages multiprecision)`, which exports a `gmp` global variable, among other things; that `gmp` variable is listed in the `inputs` field of `guile` and `guile-2.0`. The package manager *and* the distribution are just a set of "normal" modules that any program or library can use.

Packages carry meta-data, as shown in Figure 4. Synopses and descriptions are internationalized using GNU Gettext—that is, they can be translated in the user's native language, a feature that comes for free when embedding the DSL in a mature environment like Guile. We are in the process of implementing mechanisms to synchronize part of that meta-data, such as synopses, with other databases of the GNU Project.

While the distribution is not bootable yet, it already includes a set of tools to build bootable GNU/Linux images for the QEMU emulator. This includes a package for the kernel itself, as well as procedures to build QEMU images, and Linux "initrd"—the "initial RAM disk" used by Linux when booting, and which is responsible for loading essential kernel modules and mounting the root file system, among other things. For example, we provide the `expression->derivation-in-linux-vm`: it works in a way similar to `build-expression->derivation`, except that the given expression is evaluated in a virtual machine that mounts the host's store over CIFS. As a demonstration, we implemented a derivation that builds a "boot-to-Guile" QEMU image, where the initrd contains a statically-linked Guile that directly runs a boot program written in Scheme [5].

The performance-critical parts are the derivation primitives discussed in Section 3. For instance, the computation of Emacs's derivation involves that of 292 other derivations—that is, 292 invocations of the `derivation` primitive—corresponding to 582 RPCs[6]. The wall time of evaluating that derivation is 1.1 second on average on a 2.6 GHz `x86_64` machine. This is acceptable as a user, but 5 times slower than Nix's clients for a similar derivation written in the Nix language. Profiling shows that Guix spends most of its time in its derivation serialization code and RPCs. We interpret this as a consequence of Guix's unoptimized code, as well as the difference between native C++ code and our interpreted bytecode.

4.2 Purity

Providing pure build environments that do not honor the "standard" file system layout turned out not to be a problem, as already evidenced in NixOS [8]. This is largely thanks to the ubiquity of the GNU build system, which strives to provide users with ways to customize the layout of installed packages and to adjust to the user's file locations.

The only directories visible in the build `chroot` environment are `/dev`, `/proc`, and the subset of `/nix/store` that is ex-

[6]The number of `derivation` calls and `add-to-store` RPCs is reduced thanks to the use of client-side memoization.

plicitly declared in the derivation being built. NixOS makes one exception: it relies on the availability of `/bin/sh` in the `chroot` [9]. We remove that exception, and instead automatically patch script "shebangs" in the package's source, as noted in Section 3.4. This turned out to be more than just a theoretical quest for "purity". First, some GNU/Linux distributions use Dash as the implementation of `/bin/sh`, while others use Bash; these are two variants of the Bourne shell, with different extensions, and in general different behavior. Second, `/bin/sh` is typically a dynamically-linked executable. So adding `/bin` to the `chroot` is not enough; one typically needs to also add `/lib*` and `/lib/*-linux-gnu` to the chroot. At that point, there are many impurities, and a great potential for non-reproducibility—which defeats the purpose of the `chroot`.

Several packages had to be adjusted for proper function in the absence of `/bin/sh` [6]. In particular, libc's `system` and `popen` functions had to be changed to refer to "our" Bash instance. Likewise, GNU Make, GNU Awk, GNU Guile, and Python needed adjustment. Occasionally, occurrences of `/bin/sh` are not be handled automatically, for instance in test suites; these have to be patched manually in the package's recipe.

4.3 Bootstrapping

Bootstrapping in our context refers to how the distribution gets built "from nothing". Remember that the build environment of a derivation contains nothing but its declared inputs. So there's an obvious chicken-and-egg problem: how does the first package get built? How does the first compiler get compiled?

The GNU system we are building is primarily made of C code, with libc at its core. The GNU build system itself assumes the availability of a Bourne shell, traditional Unix tools provided by GNU Coreutils, Awk, Findutils, sed, and grep. Furthermore, our build programs are written in Guile Scheme. Consequently, we rely on pre-built statically-linked binaries of GCC, Binutils, libc, and the other packages mentioned above to get started.

Figure 10 shows the very beginning of the dependency graph of our distribution. At this level of detail, things are slightly more complex. First, Guile itself consists of an ELF executable, along with many source and compiled Scheme files that are dynamically loaded when it runs. This gets stored in the `guile-2.0.7.tar.xz` tarball shown in this graph. This tarball is part of Guix's "source" distribution, and gets inserted into the store with `add-to-store`.

But how do we write a derivation that unpacks this tarball and adds it to the store? To solve this problem, the `guile-bootstrap-2.0.drv` derivation—the first one that gets built—uses `bash` as its builder, which runs `build-bootstrap-guile.sh`, which in turn calls `tar` to unpack the tarball. Thus, `bash`, `tar`, `xz`, and `mkdir` are statically-linked binaries, also part of the Guix source distribution, whose sole purpose is to allow the Guile tarball to be unpacked.

Once `guile-bootstrap-2.0.drv` is built, we have a functioning Guile that can be used to run subsequent build programs. Its first task is to download tarballs containing the other pre-built binaries—this is what the `.tar.xz.drv` derivations do. Guix modules such as `ftp-client.scm` are used for this purpose. The `module-import.drv` derivations import those modules in a directory in the store, using the original layout[7]. The `module-import-compiled.drv` derivations compile those modules, and write them in an output directory with the right layout. This corresponds to the `#:module` argument of `build-expression->derivation` mentioned in Section 3.2.

Finally, the various tarballs are unpacked by the derivations `gcc-bootstrap-0.drv`, `glibc-bootstrap-0.drv`, etc., at which point we have a working C GNU tool chain. The first tool that gets built with these tools (not shown here) is GNU Make, which is a prerequisite for all the following packages.

Bootstrapping is complete when we have a full tool chain that does not depend on the pre-built bootstrap tools shown in Figure 10. Ways to achieve this are known, and notably documented by the *Linux From Scratch* project [1]. We can formally verify this no-dependency requirement by checking whether the files of the final tool chain contain references to the `/nix/store` directories of the bootstrap inputs.

Obviously, Guix contains `package` declarations to build the bootstrap binaries shown in Figure 10. Because the final tool chain does not depend on those tools, they rarely need to be updated. Having a way to do that automatically proves to be useful, though. Coupled with Guix's nascent support for cross-compilation, porting to a new architecture will boil down to cross-building all these bootstrap tools.

5. RELATED WORK

Numerous package managers for Scheme programs and libraries have been developed, including Racket's PLaneT, Dorodango for R6RS implementations, Chicken Scheme's "Eggs", Guildhall for Guile, and ScmPkg [16]. Unlike GNU Guix, they are typically limited to Scheme-only code, and take the core operating system software for granted. To our knowledge, they implement the *imperative* package management paradigm, and do not attempt to support features such as transactional upgrades and rollbacks. Unsurprisingly, these tools rely on package descriptions that more or less resemble those described in Section 3.3; however, in the case of at least ScmPkg, Dorodango, and Guildhall, package descriptions are written in an *external* DSL, which happens to use s-expression syntax.

In [21], the authors illustrate how the *units* mechanism of MzScheme modules could be leveraged to improve operating system packaging systems. The examples therein focus on OS services, and multiple instantiation thereof, rather than on package builds and composition.

The Nix package manager is the primary source of inspiration for Guix [8, 9]. As noted in Section 2.3, Guix reuses the low-level build and deployment mechanisms of Nix, but differs in its programming interface and preferred implementation language for build scripts. While the Nix language relies on

[7]In Guile, module names are a list of symbols, such as `(guix ftp-client)`, which map directly to file names, such as `guix/ftp-client.scm`.

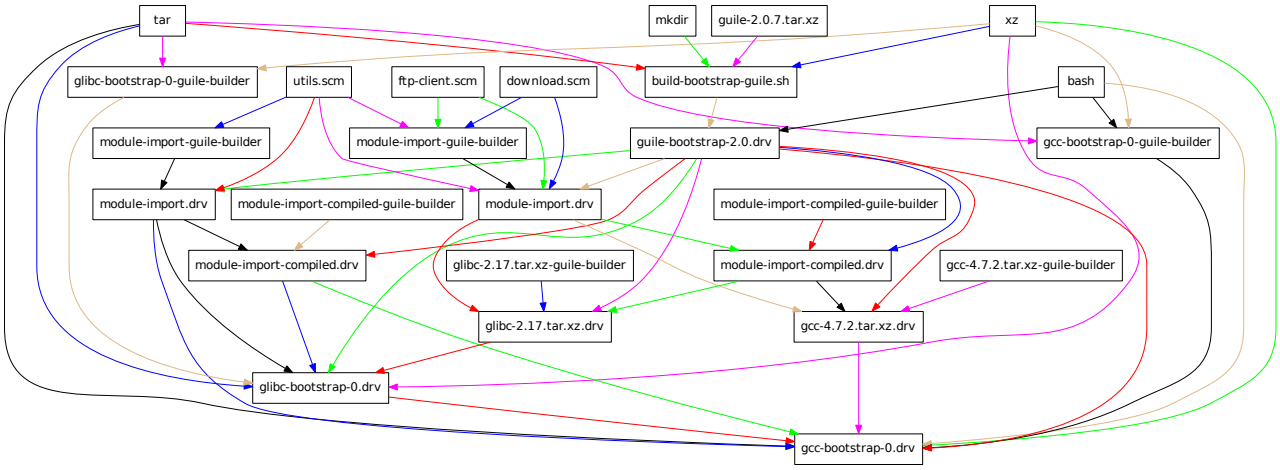

Figure 10: Dependency graph of the software distribution bootstrap.

laziness to ensure that only packages needed are built [9], we instead support *ad hoc* laziness with the `package` form. Nix and Guix have the same application: packaging of a complete GNU/Linux distribution.

Before Nix, the idea of installing each package in a directory of its own and then managing symlinks pointing to those was already present in a number of systems. In particular, the Depot [3], Store [2], and then GNU Stow [10] have long supported this approach. GNU's now defunct package management project called 'stut', ca. 2005, used that approach, with Stow as a back-end. A "Stow file system", or `stowfs`, has been available in the GNU Hurd operating system core to offer a dynamic and more elegant approach to user profiles, compared to symlink trees. The storage model of Nix/Guix can be thought of as a formalization of Stow's idea.

Like Guix and Nix, Vesta is a purely functional build system [11]. It uses an external DSL close to the Nix language. However, the primary application of Vesta is fine-grain software build operations, such as compiling a single C file. It is a developer tool, and does not address deployment to end-user machines. Unlike Guix and Nix, Vesta tries hard to support the standard Unix file system layout, relying on a virtual file system to "map" files to their right location in the build environment.

Hop defines a *multi-tier* extension of Scheme to program client/server web applications [17]. It allows client code to be introduced ("quoted") in server code, and server code to be invoked from client code. There's a parallel between the former and Guix's use of Scheme in two different strata, depicted in Section 3.2.

Scsh provides a complete interface to substitute Scheme in "shell programming" tasks [18]. Since it spans a wide range of applications, it goes beyond the tools discussed in Section 3.4 some ways, notably by providing a concise *process notation* similar to that of typical Unix shells, and S-expression regular expressions (SREs). However, we chose not to use it as its port to Guile had been unmaintained for some time, and Guile has since grown a rich operating system interface on top of which it was easy to build the few additional tools we needed.

6. CONCLUSION

GNU Guix is a contribution to package management of free operating systems. It builds on the functional paradigm pioneered by the Nix package manager [8], and benefits from its unprecedented feature set—transactional upgrades and roll-back, per-user unprivileged package management, garbage collection, and referentially-transparent build processes, among others.

We presented Guix's two main contributions from a programming point of view. First, Guix *embeds* a declarative domain-specific language in Scheme, allowing it to benefit from its associated tool set. Embedding in a general-purpose language has allowed us to easily support internationalization of package descriptions, and to write a fast keyword search mechanism; it has also permitted novel features, such as an on-line auto-updater. Second, its build programs and libraries are also written in Scheme, leading to a unified programming environment made of two strata of code.

We hope to make Guix a good vehicle for an innovative free software distribution. The GNU system distribution we envision will give Scheme an important role just above the operating system interface.

Acknowledgments

The author would like to thank the Guix contributors for their work growing the system: Andreas Enge, Nikita Karetnikov, Cyril Roelandt, and Mark H. Weaver. We are also grateful to the Nix community, and in particular to Eelco Dolstra for his inspiring PhD work that led to Nix. Lastly, thanks to the anonymous reviewer whose insight has helped improve this document.

7. REFERENCES

[1] G. Beekmans, M. Burgess, B. Dubbs. Linux From Scratch. 2013. *http://www.linuxfromscratch.org/lfs/*.

[2] A. Christensen, T. Egge. Store—a system for handling third-party applications in a heterogeneous computer environment. Springer Berlin Heidelberg, 1995, pp. 263–276.

[3] S. N. Clark, W. R. Nist. The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries. In *In Proceedings of the Fourth Large Installation Systems Administrator's Conference (LISA '90)*, pp. 37–46, 1990.

[4] R. D. Cosmo, D. D. Ruscio, P. Pelliccione, A. Pierantonio, S. Zacchiroli. Supporting software evolution in component-based FOSS systems. In *Sci. Comput. Program.*, 76(12) , Amsterdam, The Netherlands, December 2011, pp. 1144–1160.

[5] L. Courtès. Boot-to-Guile!. February 2013. *http://lists.gnu.org/archive/html/bug-guix/2013-02/msg00173.html*.

[6] L. Courtès. Down with /bin/sh!. January 2013. *https://lists.gnu.org/archive/html/bug-guix/2013-01/msg00041.html*.

[7] O. Crameri, R. Bianchini, W. Zwaenepoel, D. Kostić. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *In Proceedings of the Symposium on Operating Systems Principles*, 2007.

[8] E. Dolstra, M. d. Jonge, E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th Large Installation System Administration Conference (LISA '04)*, pp. 79–92, USENIX, November 2004.

[9] E. Dolstra, A. Löh, N. Pierron. NixOS: A Purely Functional Linux Distribution. In *Journal of Functional Programming*, (5-6) , New York, NY, USA, November 2010, pp. 577–615.

[10] B. Glickstein, K. Hodgson. Stow—Managing the Installation of Software Packages. 2012. *http://www.gnu.org/software/stow/*.

[11] A. Heydon, R. Levin, Y. Yu. Caching Function Calls Using Precise Dependencies. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, PLDI '00, pp. 311–320, ACM, 2000.

[12] P. Hudak. Building domain-specific embedded languages. In *ACM Computing Surveys*, 28(4es) , New York, NY, USA, December 1996, .

[13] R. Kelsey. Defining Record Types. 1999. *http://srfi.schemers.org/srfi-9/srfi-9.html*.

[14] R. Kelsey, M. Sperber. Conditions. 2002. *http://srfi.schemers.org/srfi-35/srfi-35.html*.

[15] T. Morris. Asymmetric Lenses in Scala. 2012. *http://days2012.scala-lang.org/*.

[16] M. Serrano, É. Gallesio. An Adaptive Package Management System for Scheme. In *Proceedings of the 2007 Symposium on Dynamic languages*, DLS '07, pp. 65–76, ACM, 2007.

[17] M. Serrano, G. Berry. Multitier Programming in Hop. In *Queue*, 10(7) , New York, NY, USA, July 2012, pp. 10:10–10:22.

[18] O. Shivers, B. D. Carlstrom, M. Gasbichler, M. Sperber. Scsh Reference Manual. 2006. *http://www.scsh.net/*.

[19] M. Sperber, R. K. Dybvig, M. Flatt, A. V. Straaten, R. B. Findler, J. Matthews. Revised6 Report on the Algorithmic Language Scheme. In *Journal of Functional Programming*, 19, 7 2009, pp. 1–301.

[20] R. M. Stallman. The GNU Manifesto. 1983. *http://www.gnu.org/gnu/manifesto.html*.

[21] D. B. Tucker, S. Krishnamurthi. Applying Module System Research to Package Management. In *Proceedings of the Tenth International Workshop on Software Configuration Management*, 2001.