GNU Guix Reference Manual

Using the GNU Guix Functional Package Manager

The GNU Guix Developers

Edition 0.9.0
4 November 2015

Table of Contents

GNU Guix 1

1 Introduction 2

2 Installation 3

2.1 Binary Installation 3
2.2 Requirements 4
2.3 Running the Test Suite 5
2.4 Setting Up the Daemon 5
2.4.1 Build Environment Setup 5
2.4.2 Using the Offload Facility 6
2.5 Invoking `guix-daemon` 8
2.6 Application Setup 11
2.6.1 Locales 11
2.6.2 X11 Fonts 12

3 Package Management 13

3.1 Features 13
3.2 Invoking `guix package` 14
3.3 Substitutes 20
3.4 Packages with Multiple Outputs 21
3.5 Invoking `guix gc` 22
3.6 Invoking `guix pull` 24
3.7 Invoking `guix archive` 24

4 Emacs Interface 27

4.1 Initial Setup 27
4.2 Package Management 28
4.2.1 Commands 28
4.2.2 General information 29
4.2.3 “List” buffer 30
4.2.4 “Info” buffer 31
4.2.5 Configuration 31
4.2.5.1 Guile and Build Options 31
4.2.5.2 Buffer Names 31
4.2.5.3 Keymaps 32
4.2.5.4 Appearance 33
4.3 Popup Interface 33
4.4 Guix Prettify Mode 34
4.5 Build Log Mode 34
4.6 Shell Completions 35
4.7 Development 35

5 Programming Interface 37
5.1 Defining Packages 37
5.1.1 `package` Reference 39
5.1.2 `origin` Reference 41
5.2 Build Systems 42
5.3 The Store 46
5.4 Derivations 47
5.5 The Store Monad 50
5.6 G-Expressions 53

6 Utilities 60
6.1 Invoking `guix build` 60
6.2 Invoking `guix edit` 64
6.3 Invoking `guix download` 65
6.4 Invoking `guix hash` 65
6.5 Invoking `guix import` 66
6.6 Invoking `guix refresh` 68
6.7 Invoking `guix lint` 70
6.8 Invoking `guix size` 71
6.9 Invoking `guix graph` 72
6.10 Invoking `guix environment` 75
6.11 Invoking `guix publish` 78
6.12 Invoking `guix challenge` 79
6.13 Invoking `guix container` 80

7 GNU Distribution 82
7.1 System Installation 82
7.1.1 Limitations 82
7.1.2 USB Stick Installation 83
7.1.3 Preparing for Installation 84
7.1.4 Proceeding with the Installation 84
7.1.5 Building the Installation Image 85
7.2 System Configuration 85
7.2.1 Using the Configuration System 85
7.2.2 `operating-system` Reference 89
7.2.3 File Systems 91
7.2.4 Mapped Devices 93
7.2.5 User Accounts 94
7.2.6 Locales 96
7.2.6.1 Locale Data Compatibility Considerations 97
7.2.7 Services 97
7.2.7.1 Base Services 98
7.2.7.2 Networking Services 101
7.2.7.3 X Window 103
7.2.7.4 Desktop Services 104
7.2.7.5 Database Services 108
7.2.7.6 Web Services 108

7.2.7.7 Various Services ... 108
7.2.8 Setuid Programs ... 108
7.2.9 X.509 Certificates ... 109
7.2.10 Name Service Switch ... 110
7.2.11 Initial RAM Disk ... 112
7.2.12 GRUB Configuration ... 113
7.2.13 Invoking `guix system` ... 114
7.2.14 Defining Services ... 117
7.2.14.1 Service Composition ... 117
7.2.14.2 Service Types and Services ... 118
7.2.14.3 Service Reference ... 120
7.2.14.4 dmd Services ... 123
7.3 Installing Debugging Files ... 124
7.4 Security Updates ... 125
7.5 Package Modules ... 126
7.6 Packaging Guidelines ... 127
7.6.1 Software Freedom ... 128
7.6.2 Package Naming ... 128
7.6.3 Version Numbers ... 128
7.6.4 Synopses and Descriptions ... 129
7.6.5 Python Modules ... 130
7.6.6 Perl Modules ... 130
7.6.7 Fonts ... 130
7.7 Bootstrapping ... 131
Preparing to Use the Bootstrap Binaries ... 132
Building the Build Tools ... 133
Building the Bootstrap Binaries ... 133
7.8 Porting to a New Platform ... 134

8 Contributing ... 135
8.1 Building from Git ... 135
8.2 Running Guix Before It Is Installed ... 135
8.3 The Perfect Setup ... 136
8.4 Coding Style ... 137
8.4.1 Programming Paradigm ... 137
8.4.2 Modules ... 137
8.4.3 Data Types and Pattern Matching ... 137
8.4.4 Formatting Code ... 137
8.5 Submitting Patches ... 137

9 Acknowledgments ... 139

Appendix A GNU Free Documentation License ... 140

Concept Index ... 148

Programming Index **150**

GNU Guix

This document describes GNU Guix version 0.9.0, a functional package management tool written for the GNU system.

1 Introduction

GNU Guix[1] is a functional package management tool for the GNU system. Package management consists of all activities that relate to building packages from sources, honoring their build-time and run-time dependencies, installing packages in user environments, upgrading installed packages to new versions or rolling back to a previous set, removing unused software packages, etc.

The term *functional* refers to a specific package management discipline pioneered by Nix (see Chapter 9 [Acknowledgments], page 139). In Guix, the package build and installation process is seen as a function, in the mathematical sense. That function takes inputs, such as build scripts, a compiler, and libraries, and returns an installed package. As a pure function, its result depends solely on its inputs—for instance, it cannot refer to software or scripts that were not explicitly passed as inputs. A build function always produces the same result when passed a given set of inputs. It cannot alter the system's environment in any way; for instance, it cannot create, modify, or delete files outside of its build and installation directories. This is achieved by running build processes in isolated environments (or *containers*), where only their explicit inputs are visible.

The result of package build functions is *cached* in the file system, in a special directory called *the store* (see Section 5.3 [The Store], page 46). Each package is installed in a directory of its own, in the store—by default under `/gnu/store`. The directory name contains a hash of all the inputs used to build that package; thus, changing an input yields a different directory name.

This approach is the foundation of Guix's salient features: support for transactional package upgrade and rollback, per-user installation, and garbage collection of packages (see Section 3.1 [Features], page 13).

Guix has a command-line interface, which allows users to build, install, upgrade, and remove packages, as well as a Scheme programming interface.

Last but not least, Guix is used to build a distribution of the GNU system, with many GNU and non-GNU free software packages. The Guix System Distribution, or GNU GuixSD, takes advantage of the core properties of Guix at the system level. With GuixSD, users *declare* all aspects of the operating system configuration, and Guix takes care of instantiating that configuration in a reproducible, stateless fashion. See Chapter 7 [GNU Distribution], page 82.

[1] "Guix" is pronounced like "geeks", or "iks" using the international phonetic alphabet (IPA).

2 Installation

GNU Guix is available for download from its website at http://www.gnu.org/software/guix/. This section describes the software requirements of Guix, as well as how to install it and get ready to use it.

Note that this section is concerned with the installation of the package manager, which can be done on top of a running GNU/Linux system. If, instead, you want to install the complete GNU operating system, see Section 7.1 [System Installation], page 82.

2.1 Binary Installation

This section describes how to install Guix on an arbitrary system from a self-contained tarball providing binaries for Guix and for all its dependencies. This is often quicker than installing from source, which is described in the next sections. The only requirement is to have GNU tar and Xz.

Installing goes along these lines:

1. Download the binary tarball from '`ftp://alpha.gnu.org/gnu/guix/guix-binary-0.9.0.`*system*`.tar.x` where *system* is `x86_64-linux` for an `x86_64` machine already running the kernel Linux, and so on.
2. As `root`, run:

   ```
   # cd /tmp
   # tar --warning=no-timestamp -xf \
       guix-binary-0.9.0.system.tar.xz
   # mv var/guix /var/ && mv gnu /
   ```

   This creates `/gnu/store` (see Section 5.3 [The Store], page 46) and `/var/guix`. The latter contains a ready-to-use profile for `root` (see next step.)

   Do *not* unpack the tarball on a working Guix system since that would overwrite its own essential files.

   The `--warning=no-timestamp` option makes sure GNU tar does not emit warnings about "implausibly old time stamps" (such warnings were triggered by GNU tar 1.26 and older; recent versions are fine.) They stem from the fact that all the files in the archive have their modification time set to zero (which means January 1st, 1970.) This is done on purpose to make sure the archive content is independent of its creation time, thus making it reproducible.
3. Make `root`'s profile available under `~/.guix-profile`:

   ```
   # ln -sf /var/guix/profiles/per-user/root/guix-profile \
            ~root/.guix-profile
   ```
4. Create the group and user accounts for build users as explained below (see Section 2.4.1 [Build Environment Setup], page 5).
5. Run the daemon:

   ```
   # ~root/.guix-profile/bin/guix-daemon --build-users-group=guixbuild
   ```

   On hosts using the systemd init system, drop `~root/.guix-profile/lib/systemd/system/guix-daemon.service` in `/etc/systemd/system`.

[1] As usual, make sure to download the associated `.sig` file and to verify the authenticity of the tarball against it!

6. Make the `guix` command available to other users on the machine, for instance with:

```
# mkdir -p /usr/local/bin
# cd /usr/local/bin
# ln -s /var/guix/profiles/per-user/root/guix-profile/bin/guix
```

7. To use substitutes from `hydra.gnu.org` (see Section 3.3 [Substitutes], page 20), authorize them:

```
# guix archive --authorize < ~root/.guix-profile/share/guix/hydra.gnu.org.pub
```

And that's it! For additional tips and tricks, see Section 2.6 [Application Setup], page 11.

The `guix` package must remain available in `root`'s profile, or it would become subject to garbage collection—in which case you would find yourself badly handicapped by the lack of the `guix` command.

The tarball in question can be (re)produced and verified simply by running the following command in the Guix source tree:

```
make guix-binary.system.tar.xz
```

2.2 Requirements

This section lists requirements when building Guix from source. The build procedure for Guix is the same as for other GNU software, and is not covered here. Please see the files `README` and `INSTALL` in the Guix source tree for additional details.

GNU Guix depends on the following packages:

- GNU Guile, version 2.0.7 or later;
- GNU libgcrypt;
- GNU Make.

The following dependencies are optional:

- Installing Guile-JSON will allow you to use the `guix import pypi` command (see Section 6.5 [Invoking guix import], page 66). It is of interest primarily for developers and not for casual users.
- Installing GnuTLS-Guile will allow you to access `https` URLs with the `guix download` command (see Section 6.3 [Invoking guix download], page 65), the `guix import pypi` command, and the `guix import cpan` command. This is primarily of interest to developers. See Section "Guile Preparations" in *GnuTLS-Guile*.

Unless `--disable-daemon` was passed to `configure`, the following packages are also needed:

- SQLite 3;
- libbz2;
- GCC's g++, with support for the C++11 standard.

When a working installation of the Nix package manager is available, you can instead configure Guix with `--disable-daemon`. In that case, Nix replaces the three dependencies above.

Guix is compatible with Nix, so it is possible to share the same store between both. To do so, you must pass `configure` not only the same `--with-store-dir` value, but also the

same `--localstatedir` value. The latter is essential because it specifies where the database that stores metadata about the store is located, among other things. The default values for Nix are `--with-store-dir=/nix/store` and `--localstatedir=/nix/var`. Note that `--disable-daemon` is not required if your goal is to share the store with Nix.

2.3 Running the Test Suite

After a successful `configure` and `make` run, it is a good idea to run the test suite. It can help catch issues with the setup or environment, or bugs in Guix itself—and really, reporting test failures is a good way to help improve the software. To run the test suite, type:

```
make check
```

Test cases can run in parallel: you can use the `-j` option of GNU make to speed things up. The first run may take a few minutes on a recent machine; subsequent runs will be faster because the store that is created for test purposes will already have various things in cache.

Upon failure, please email `bug-guix@gnu.org` and attach the `test-suite.log` file. When `tests/`*`something`*`.scm` fails, please also attach the *`something`*`.log` file available in the top-level build directory. Please specify the Guix version being used as well as version numbers of the dependencies (see Section 2.2 [Requirements], page 4) in your message.

2.4 Setting Up the Daemon

Operations such as building a package or running the garbage collector are all performed by a specialized process, the *build daemon*, on behalf of clients. Only the daemon may access the store and its associated database. Thus, any operation that manipulates the store goes through the daemon. For instance, command-line tools such as `guix package` and `guix build` communicate with the daemon (*via* remote procedure calls) to instruct it what to do.

The following sections explain how to prepare the build daemon's environment. Also Section 3.3 [Substitutes], page 20, for information on how to allow the daemon to download pre-built binaries.

2.4.1 Build Environment Setup

In a standard multi-user setup, Guix and its daemon—the `guix-daemon` program—are installed by the system administrator; `/gnu/store` is owned by `root` and `guix-daemon` runs as `root`. Unprivileged users may use Guix tools to build packages or otherwise access the store, and the daemon will do it on their behalf, ensuring that the store is kept in a consistent state, and allowing built packages to be shared among users.

When `guix-daemon` runs as `root`, you may not want package build processes themselves to run as `root` too, for obvious security reasons. To avoid that, a special pool of *build users* should be created for use by build processes started by the daemon. These build users need not have a shell and a home directory: they will just be used when the daemon drops `root` privileges in build processes. Having several such users allows the daemon to launch distinct build processes under separate UIDs, which guarantees that they do not interfere with each other—an essential feature since builds are regarded as pure functions (see Chapter 1 [Introduction], page 2).

On a GNU/Linux system, a build user pool may be created like this (using Bash syntax and the `shadow` commands):

```
# groupadd --system guixbuild
# for i in `seq -w 1 10`;
  do
    useradd -g guixbuild -G guixbuild           \
            -d /var/empty -s `which nologin`    \
            -c "Guix build user $i" --system    \
            guixbuilder$i;
  done
```

The number of build users determines how many build jobs may run in parallel, as specified by the `--max-jobs` option (see Section 2.5 [Invoking guix-daemon], page 8). The `guix-daemon` program may then be run as `root` with the following command[2]:

```
# guix-daemon --build-users-group=guixbuild
```

This way, the daemon starts build processes in a chroot, under one of the `guixbuilder` users. On GNU/Linux, by default, the chroot environment contains nothing but:

- a minimal `/dev` directory, created mostly independently from the host `/dev`[3];
- the `/proc` directory; it only shows the container's processes since a separate PID name space is used;
- `/etc/passwd` with an entry for the current user and an entry for user `nobody`;
- `/etc/group` with an entry for the user's group;
- `/etc/hosts` with an entry that maps `localhost` to `127.0.0.1`;
- a writable `/tmp` directory.

If you are installing Guix as an unprivileged user, it is still possible to run `guix-daemon` provided you pass `--disable-chroot`. However, build processes will not be isolated from one another, and not from the rest of the system. Thus, build processes may interfere with each other, and may access programs, libraries, and other files available on the system—making it much harder to view them as *pure* functions.

2.4.2 Using the Offload Facility

When desired, the build daemon can *offload* derivation builds to other machines running Guix, using the `offload` *build hook*. When that feature is enabled, a list of user-specified build machines is read from `/etc/guix/machines.scm`; anytime a build is requested, for instance via `guix build`, the daemon attempts to offload it to one of the machines that satisfies the derivation's constraints, in particular its system type—e.g., `x86_64-linux`. Missing prerequisites for the build are copied over SSH to the target machine, which then proceeds with the build; upon success the output(s) of the build are copied back to the initial machine.

The `/etc/guix/machines.scm` file typically looks like this:

---

[2] If your machine uses the systemd init system, dropping the *prefix*`/lib/systemd/system/guix-daemon.service` file in `/etc/systemd/system` will ensure that `guix-daemon` is automatically started.

[3] "Mostly", because while the set of files that appear in the chroot's `/dev` is fixed, most of these files can only be created if the host has them.

```
(list (build-machine
        (name "eightysix.example.org")
        (system "x86_64-linux")
        (user "bob")
        (speed 2.))    ; incredibly fast!

      (build-machine
        (name "meeps.example.org")
        (system "mips64el-linux")
        (user "alice")
        (private-key
         (string-append (getenv "HOME")
                        "/.lsh/identity-for-guix"))))
```

In the example above we specify a list of two build machines, one for the `x86_64` architecture and one for the `mips64el` architecture.

In fact, this file is—not surprisingly!—a Scheme file that is evaluated when the `offload` hook is started. Its return value must be a list of `build-machine` objects. While this example shows a fixed list of build machines, one could imagine, say, using DNS-SD to return a list of potential build machines discovered in the local network (see Section "Introduction" in *Using Avahi in Guile Scheme Programs*). The `build-machine` data type is detailed below.

`build-machine` [Data Type]

This data type represents build machines the daemon may offload builds to. The important fields are:

`name` The remote machine's host name.

`system` The remote machine's system type—e.g., `"x86_64-linux"`.

`user` The user account to use when connecting to the remote machine over SSH. Note that the SSH key pair must *not* be passphrase-protected, to allow non-interactive logins.

A number of optional fields may be specified:

`port` Port number of the machine's SSH server (default: 22).

`private-key`
The SSH private key file to use when connecting to the machine.

Currently offloading uses GNU lsh as its SSH client (see Section "Invoking lsh" in `GNU lsh Manual`). Thus, the key file here must be an lsh key file. This may change in the future, though.

`parallel-builds`
The number of builds that may run in parallel on the machine (1 by default.)

`speed` A "relative speed factor". The offload scheduler will tend to prefer machines with a higher speed factor.

`features` A list of strings denoting specific features supported by the machine. An example is `"kvm"` for machines that have the KVM Linux modules and

corresponding hardware support. Derivations can request features by name, and they will be scheduled on matching build machines.

The `guix` command must be in the search path on the build machines, since offloading works by invoking the `guix archive` and `guix build` commands. In addition, the Guix modules must be in `$GUILE_LOAD_PATH` on the build machine—you can check whether this is the case by running:

```
lsh build-machine guile -c '(use-modules (guix config))'
```

There's one last thing to do once `machines.scm` is in place. As explained above, when offloading, files are transferred back and forth between the machine stores. For this to work, you first need to generate a key pair on each machine to allow the daemon to export signed archives of files from the store (see Section 3.7 [Invoking guix archive], page 24):

```
# guix archive --generate-key
```

Each build machine must authorize the key of the master machine so that it accepts store items it receives from the master:

```
# guix archive --authorize < master-public-key.txt
```

Likewise, the master machine must authorize the key of each build machine.

All the fuss with keys is here to express pairwise mutual trust relations between the master and the build machines. Concretely, when the master receives files from a build machine (and *vice versa*), its build daemon can make sure they are genuine, have not been tampered with, and that they are signed by an authorized key.

2.5 Invoking `guix-daemon`

The `guix-daemon` program implements all the functionality to access the store. This includes launching build processes, running the garbage collector, querying the availability of a build result, etc. It is normally run as `root` like this:

```
# guix-daemon --build-users-group=guixbuild
```

For details on how to set it up, see Section 2.4 [Setting Up the Daemon], page 5.

By default, `guix-daemon` launches build processes under different UIDs, taken from the build group specified with `--build-users-group`. In addition, each build process is run in a chroot environment that only contains the subset of the store that the build process depends on, as specified by its derivation (see Chapter 5 [Programming Interface], page 37), plus a set of specific system directories. By default, the latter contains `/dev` and `/dev/pts`. Furthermore, on GNU/Linux, the build environment is a *container*: in addition to having its own file system tree, it has a separate mount name space, its own PID name space, network name space, etc. This helps achieve reproducible builds (see Section 3.1 [Features], page 13).

When the daemon performs a build on behalf of the user, it creates a build directory under `/tmp` or under the directory specified by its `TMPDIR` environment variable; this directory is shared with the container for the duration of the build. Be aware that using a directory other than `/tmp` can affect build results—for example, with a longer directory name, a build process that uses Unix-domain sockets might hit the name length limitation for `sun_path`, which it would otherwise not hit.

The build directory is automatically deleted upon completion, unless the build failed and the client specified `--keep-failed` (see Section 6.1 [Invoking guix build], page 60).

The following command-line options are supported:

`--build-users-group=`*group*
: Take users from *group* to run build processes (see Section 2.4 [Setting Up the Daemon], page 5).

`--no-substitutes`
: Do not use substitutes for build products. That is, always build things locally instead of allowing downloads of pre-built binaries (see Section 3.3 [Substitutes], page 20).

  By default substitutes are used, unless the client—such as the `guix package` command—is explicitly invoked with `--no-substitutes`.

  When the daemon runs with `--no-substitutes`, clients can still explicitly enable substitution *via* the `set-build-options` remote procedure call (see Section 5.3 [The Store], page 46).

`--substitute-urls=`*urls*
: Consider *urls* the default whitespace-separated list of substitute source URLs. When this option is omitted, '`http://hydra.gnu.org`' is used.

  This means that substitutes may be downloaded from *urls*, as long as they are signed by a trusted signature (see Section 3.3 [Substitutes], page 20).

`--no-build-hook`
: Do not use the *build hook*.

  The build hook is a helper program that the daemon can start and to which it submits build requests. This mechanism is used to offload builds to other machines (see Section 2.4.2 [Daemon Offload Setup], page 6).

`--cache-failures`
: Cache build failures. By default, only successful builds are cached.

  When this option is used, `guix gc --list-failures` can be used to query the set of store items marked as failed; `guix gc --clear-failures` removes store items from the set of cached failures. See Section 3.5 [Invoking guix gc], page 22.

`--cores=`*n*
`-c` *n*
: Use *n* CPU cores to build each derivation; `0` means as many as available.

  The default value is `0`, but it may be overridden by clients, such as the `--cores` option of `guix build` (see Section 6.1 [Invoking guix build], page 60).

  The effect is to define the `NIX_BUILD_CORES` environment variable in the build process, which can then use it to exploit internal parallelism—for instance, by running `make -j$NIX_BUILD_CORES`.

`--max-jobs=`*n*
`-M` *n*
: Allow at most *n* build jobs in parallel. The default value is `1`. Setting it to `0` means that no builds will be performed locally; instead, the daemon will offload builds (see Section 2.4.2 [Daemon Offload Setup], page 6), or simply fail.

`--debug` Produce debugging output.

This is useful to debug daemon start-up issues, but then it may be overridden by clients, for example the `--verbosity` option of `guix build` (see Section 6.1 [Invoking guix build], page 60).

`--chroot-directory=`*dir*
Add *dir* to the build chroot.

Doing this may change the result of build processes—for instance if they use optional dependencies found in *dir* when it is available, and not otherwise. For that reason, it is not recommended to do so. Instead, make sure that each derivation declares all the inputs that it needs.

`--disable-chroot`
Disable chroot builds.

Using this option is not recommended since, again, it would allow build processes to gain access to undeclared dependencies. It is necessary, though, when `guix-daemon` is running under an unprivileged user account.

`--disable-log-compression`
Disable compression of the build logs.

Unless `--lose-logs` is used, all the build logs are kept in the *localstatedir*. To save space, the daemon automatically compresses them with bzip2 by default. This option disables that.

`--disable-deduplication`
Disable automatic file "deduplication" in the store.

By default, files added to the store are automatically "deduplicated": if a newly added file is identical to another one found in the store, the daemon makes the new file a hard link to the other file. This can noticeably reduce disk usage, at the expense of slightly increased input/output load at the end of a build process. This option disables this optimization.

`--gc-keep-outputs[=yes|no]`
Tell whether the garbage collector (GC) must keep outputs of live derivations.

When set to "yes", the GC will keep the outputs of any live derivation available in the store—the `.drv` files. The default is "no", meaning that derivation outputs are kept only if they are GC roots.

`--gc-keep-derivations[=yes|no]`
Tell whether the garbage collector (GC) must keep derivations corresponding to live outputs.

When set to "yes", as is the case by default, the GC keeps derivations—i.e., `.drv` files—as long as at least one of their outputs is live. This allows users to keep track of the origins of items in their store. Setting it to "no" saves a bit of disk space.

Note that when both `--gc-keep-derivations` and `--gc-keep-outputs` are used, the effect is to keep all the build prerequisites (the sources, compiler, libraries, and other build-time tools) of live objects in the store, regardless of

whether these prerequisites are live. This is convenient for developers since it saves rebuilds or downloads.

`--impersonate-linux-2.6`
: On Linux-based systems, impersonate Linux 2.6. This means that the kernel's `uname` system call will report 2.6 as the release number.

  This might be helpful to build programs that (usually wrongfully) depend on the kernel version number.

`--lose-logs`
: Do not keep build logs. By default they are kept under *`localstatedir`*`/guix/log`.

`--system=`*`system`*
: Assume *system* as the current system type. By default it is the architecture/kernel pair found at configure time, such as `x86_64-linux`.

`--listen=`*`socket`*
: Listen for connections on *socket*, the file name of a Unix-domain socket. The default socket is *`localstatedir`*`/daemon-socket/socket`. This option is only useful in exceptional circumstances, such as if you need to run several daemons on the same machine.

2.6 Application Setup

When using Guix on top of GNU/Linux distribution other than GuixSD—a so-called *foreign distro*—a few additional steps are needed to get everything in place. Here are some of them.

2.6.1 Locales

Packages installed *via* Guix will not use the host system's locale data. Instead, you must first install one of the locale packages available with Guix and then define the `GUIX_LOCPATH` environment variable:

```
$ guix package -i glibc-locales
$ export GUIX_LOCPATH=$HOME/.guix-profile/lib/locale
```

Note that the `glibc-locales` package contains data for all the locales supported by the GNU libc and weighs in at around 110 MiB. Alternately, the `glibc-utf8-locales` is smaller but limited to a few UTF-8 locales.

The `GUIX_LOCPATH` variable plays a role similar to `LOCPATH` (see Section "Locale Names" in *The GNU C Library Reference Manual*). There are two important differences though:

1. `GUIX_LOCPATH` is honored only by Guix's libc, and not by the libc provided by foreign distros. Thus, using `GUIX_LOCPATH` allows you to make sure the the foreign distro's programs will not end up loading incompatible locale data.
2. libc suffixes each entry of `GUIX_LOCPATH` with `/X.Y`, where `X.Y` is the libc version—e.g., `2.22`. This means that, should your Guix profile contain a mixture of programs linked against different libc version, each libc version will only try to load locale data in the right format.

This is important because the locale data format used by different libc versions may be incompatible.

2.6.2 X11 Fonts

The majority of graphical applications use Fontconfig to locate and load fonts and perform X11-client-side rendering. Guix's `fontconfig` package looks for fonts in `$HOME/.guix-profile` by default. Thus, to allow graphical applications installed with Guix to display fonts, you will have to install fonts with Guix as well. Essential font packages include `gs-fonts`, `font-dejavu`, and `font-gnu-freefont-ttf`.

3 Package Management

The purpose of GNU Guix is to allow users to easily install, upgrade, and remove software packages, without having to know about their build procedure or dependencies. Guix also goes beyond this obvious set of features.

This chapter describes the main features of Guix, as well as the package management tools it provides. Two user interfaces are provided for routine package management tasks: A command-line interface described below (see Section 3.2 [Invoking guix package], page 14), as well as a visual user interface in Emacs described in a subsequent chapter (see Chapter 4 [Emacs Interface], page 27).

3.1 Features

When using Guix, each package ends up in the *package store*, in its own directory—something that resembles `/gnu/store/xxx-package-1.2`, where `xxx` is a base32 string (note that Guix comes with an Emacs extension to shorten those file names, see Section 4.4 [Emacs Prettify], page 34.)

Instead of referring to these directories, users have their own *profile*, which points to the packages that they actually want to use. These profiles are stored within each user's home directory, at `$HOME/.guix-profile`.

For example, `alice` installs GCC 4.7.2. As a result, `/home/alice/.guix-profile/bin/gcc` points to `/gnu/store/...-gcc-4.7.2/bin/gcc`. Now, on the same machine, `bob` had already installed GCC 4.8.0. The profile of `bob` simply continues to point to `/gnu/store/...-gcc-4.8.0/bin/gcc`—i.e., both versions of GCC coexist on the same system without any interference.

The `guix package` command is the central tool to manage packages (see Section 3.2 [Invoking guix package], page 14). It operates on those per-user profiles, and can be used *with normal user privileges*.

The command provides the obvious install, remove, and upgrade operations. Each invocation is actually a *transaction*: either the specified operation succeeds, or nothing happens. Thus, if the `guix package` process is terminated during the transaction, or if a power outage occurs during the transaction, then the user's profile remains in its previous state, and remains usable.

In addition, any package transaction may be *rolled back*. So, if, for example, an upgrade installs a new version of a package that turns out to have a serious bug, users may roll back to the previous instance of their profile, which was known to work well. Similarly, the global system configuration is subject to transactional upgrades and roll-back (see Section 7.2.1 [Using the Configuration System], page 85).

All those packages in the package store may be *garbage-collected*. Guix can determine which packages are still referenced by the user profiles, and remove those that are provably no longer referenced (see Section 3.5 [Invoking guix gc], page 22). Users may also explicitly remove old generations of their profile so that the packages they refer to can be collected.

Finally, Guix takes a *purely functional* approach to package management, as described in the introduction (see Chapter 1 [Introduction], page 2). Each `/gnu/store` package directory name contains a hash of all the inputs that were used to build that package—compiler,

libraries, build scripts, etc. This direct correspondence allows users to make sure a given package installation matches the current state of their distribution. It also helps maximize *build reproducibility*: thanks to the isolated build environments that are used, a given build is likely to yield bit-identical files when performed on different machines (see Section 2.5 [Invoking guix-daemon], page 8).

This foundation allows Guix to support *transparent binary/source deployment*. When a pre-built binary for a `/gnu/store` item is available from an external source—a *substitute*, Guix just downloads it and unpacks it; otherwise, it builds the package from source, locally (see Section 3.3 [Substitutes], page 20).

Control over the build environment is a feature that is also useful for developers. The `guix environment` command allows developers of a package to quickly set up the right development environment for their package, without having to manually install the package's dependencies in their profile (see Section 6.10 [Invoking guix environment], page 75).

3.2 Invoking `guix package`

The `guix package` command is the tool that allows users to install, upgrade, and remove packages, as well as rolling back to previous configurations. It operates only on the user's own profile, and works with normal user privileges (see Section 3.1 [Features], page 13). Its syntax is:

```
guix package options
```

Primarily, *options* specifies the operations to be performed during the transaction. Upon completion, a new profile is created, but previous *generations* of the profile remain available, should the user want to roll back.

For example, to remove `lua` and install `guile` and `guile-cairo` in a single transaction:

```
guix package -r lua -i guile guile-cairo
```

`guix package` also supports a *declarative approach* whereby the user specifies the exact set of packages to be available and passes it *via* the `--manifest` option (see [profile-manifest], page 16).

For each user, a symlink to the user's default profile is automatically created in `$HOME/.guix-profile`. This symlink always points to the current generation of the user's default profile. Thus, users can add `$HOME/.guix-profile/bin` to their `PATH` environment variable, and so on. If you are not using the Guix System Distribution, consider adding the following lines to your `~/.bash_profile` (see Section "Bash Startup Files" in *The GNU Bash Reference Manual*) so that newly-spawned shells get all the right environment variable definitions:

```
GUIX_PROFILE="$HOME/.guix-profile" \
source "$HOME/.guix-profile/etc/profile"
```

In a multi-user setup, user profiles are stored in a place registered as a *garbage-collector root*, which `$HOME/.guix-profile` points to (see Section 3.5 [Invoking guix gc], page 22). That directory is normally *`localstatedir`*`/profiles/per-user/`*`user`*, where *localstatedir* is the value passed to `configure` as `--localstatedir`, and *user* is the user name. The `per-user` directory is created when `guix-daemon` is started, and the *user* sub-directory is created by `guix package`.

The *options* can be among the following:

`--install=`*package* ...
`-i` *package* ...
: Install the specified *package*s.

  Each *package* may specify either a simple package name, such as `guile`, or a package name followed by a hyphen and version number, such as `guile-1.8.8` or simply `guile-1.8` (in the latter case, the newest version prefixed by `1.8` is selected.)

  If no version number is specified, the newest available version will be selected. In addition, *package* may contain a colon, followed by the name of one of the outputs of the package, as in `gcc:doc` or `binutils-2.22:lib` (see Section 3.4 [Packages with Multiple Outputs], page 21). Packages with a corresponding name (and optionally version) are searched for among the GNU distribution modules (see Section 7.5 [Package Modules], page 126).

  Sometimes packages have *propagated inputs*: these are dependencies that automatically get installed along with the required package (see [package-propagated-inputs], page 40, for information about propagated inputs in package definitions).

  An example is the GNU MPC library: its C header files refer to those of the GNU MPFR library, which in turn refer to those of the GMP library. Thus, when installing MPC, the MPFR and GMP libraries also get installed in the profile; removing MPC also removes MPFR and GMP—unless they had also been explicitly installed independently.

  Besides, packages sometimes rely on the definition of environment variables for their search paths (see explanation of `--search-paths` below). Any missing or possibly incorrect environment variable definitions are reported here.

  Finally, when installing a GNU package, the tool reports the availability of a newer upstream version. In the future, it may provide the option of installing directly from the upstream version, even if that version is not yet in the distribution.

`--install-from-expression=`*exp*
`-e` *exp*
: Install the package *exp* evaluates to.

  exp must be a Scheme expression that evaluates to a `<package>` object. This option is notably useful to disambiguate between same-named variants of a package, with expressions such as `(@ (gnu packages base) guile-final)`.

  Note that this option installs the first output of the specified package, which may be insufficient when needing a specific output of a multiple-output package.

`--install-from-file=`*file*
`-f` *file*
: Install the package that the code within *file* evaluates to.

  As an example, *file* might contain a definition like this (see Section 5.1 [Defining Packages], page 37):

```
(use-modules (guix)
             (guix build-system gnu)
             (guix licenses))
```

```
(package
  (name "hello")
  (version "2.10")
  (source (origin
            (method url-fetch)
            (uri (string-append "mirror://gnu/hello/hello-" version
                                ".tar.gz"))
            (sha256
             (base32
              "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
  (build-system gnu-build-system)
  (synopsis "Hello, GNU world:  An example GNU package")
  (description "Guess what GNU Hello prints!")
  (home-page "http://www.gnu.org/software/hello/")
  (license gpl3+))
```

Developers may find it useful to include such a `package.scm` file in the root of their project's source tree that can be used to test development snapshots and create reproducible development environments (see Section 6.10 [Invoking guix environment], page 75).

`--remove=`*package* `...`
`-r` *package* `...`

Remove the specified *package*s.

As for `--install`, each *package* may specify a version number and/or output name in addition to the package name. For instance, `-r glibc:debug` would remove the `debug` output of `glibc`.

`--upgrade[=`*regexp* `...]`
`-u [`*regexp* `...]`

Upgrade all the installed packages. If one or more *regexp*s are specified, upgrade only installed packages whose name matches a *regexp*. Also see the `--do-not-upgrade` option below.

Note that this upgrades package to the latest version of packages found in the distribution currently installed. To update your distribution, you should regularly run `guix pull` (see Section 3.6 [Invoking guix pull], page 24).

`--do-not-upgrade[=`*regexp* `...]`

When used together with the `--upgrade` option, do *not* upgrade any packages whose name matches a *regexp*. For example, to upgrade all packages in the current profile except those containing the substring "emacs":

```
$ guix package --upgrade . --do-not-upgrade emacs
```

`--manifest=`*file*
`-m` *file*

Create a new generation of the profile from the manifest object returned by the Scheme code in *file*.

This allows you to *declare* the profile's contents rather than constructing it through a sequence of `--install` and similar commands. The advantage is that *file* can be put under version control, copied to different machines to reproduce the same profile, and so on.

file must return a *manifest* object, which is roughly a list of packages:

```
(use-package-modules guile emacs)

(packages->manifest
 (list emacs
       guile-2.0
       ;; Use a specific package output.
       (list guile-2.0 "debug")))
```

`--roll-back`

Roll back to the previous *generation* of the profile—i.e., undo the last transaction.

When combined with options such as `--install`, roll back occurs before any other actions.

When rolling back from the first generation that actually contains installed packages, the profile is made to point to the *zeroth generation*, which contains no files apart from its own meta-data.

Installing, removing, or upgrading packages from a generation that has been rolled back to overwrites previous future generations. Thus, the history of a profile's generations is always linear.

`--switch-generation=`*pattern*
`-S` *pattern*

Switch to a particular generation defined by *pattern*.

pattern may be either a generation number or a number prefixed with "+" or "-". The latter means: move forward/backward by a specified number of generations. For example, if you want to return to the latest generation after `--roll-back`, use `--switch-generation=+1`.

The difference between `--roll-back` and `--switch-generation=-1` is that `--switch-generation` will not make a zeroth generation, so if a specified generation does not exist, the current generation will not be changed.

`--search-paths[=`*kind*`]`

Report environment variable definitions, in Bash syntax, that may be needed in order to use the set of installed packages. These environment variables are used to specify *search paths* for files used by some of the installed packages.

For example, GCC needs the `CPATH` and `LIBRARY_PATH` environment variables to be defined so it can look for headers and libraries in the user's profile (see Section "Environment Variables" in *Using the GNU Compiler Collection (GCC)*). If GCC and, say, the C library are installed in the profile, then `--search-paths` will suggest setting these variables to *`profile`*`/include` and *`profile`*`/lib`, respectively.

The typical use case is to define these environment variables in the shell:

```
$ eval `guix package --search-paths`
```

kind may be one of `exact`, `prefix`, or `suffix`, meaning that the returned environment variable definitions will either be exact settings, or prefixes or

suffixes of the current value of these variables. When omitted, *kind* defaults to `exact`.

`--profile=`*profile*
`-p` *profile*
: Use *profile* instead of the user's default profile.

`--verbose`
: Produce verbose output. In particular, emit the environment's build log on the standard error port.

`--bootstrap`
: Use the bootstrap Guile to build the profile. This option is only useful to distribution developers.

In addition to these actions `guix package` supports the following options to query the current state of a profile, or the availability of packages:

`--search=`*regexp*
`-s` *regexp*
: List the available packages whose name, synopsis, or description matches *regexp*. Print all the meta-data of matching packages in `recutils` format (see *GNU recutils manual*).

  This allows specific fields to be extracted using the `recsel` command, for instance:

```
$ guix package -s malloc | recsel -p name,version
name: glibc
version: 2.17

name: libgc
version: 7.2alpha6
```

  Similarly, to show the name of all the packages available under the terms of the GNU LGPL version 3:

```
$ guix package -s "" | recsel -p name -e 'license ~ "LGPL 3"'
name: elfutils

name: gmp
...
```

`--show=`*package*
: Show details about *package*, taken from the list of available packages, in `recutils` format (see *GNU recutils manual*).

```
$ guix package --show=python | recsel -p name,version
name: python
version: 2.7.6

name: python
version: 3.3.5
```

  You may also specify the full name of a package to only get details about a specific version of it:

```
$ guix package --show=python-3.3.5 | recsel -p name,version
name: python
version: 3.3.5
```

`--list-installed[=`*regexp*`]`
`-I [`*regexp*`]`

List the currently installed packages in the specified profile, with the most recently installed packages shown last. When *regexp* is specified, list only installed packages whose name matches *regexp*.

For each installed package, print the following items, separated by tabs: the package name, its version string, the part of the package that is installed (for instance, `out` for the default output, `include` for its headers, etc.), and the path of this package in the store.

`--list-available[=`*regexp*`]`
`-A [`*regexp*`]`

List packages currently available in the distribution for this system (see Chapter 7 [GNU Distribution], page 82). When *regexp* is specified, list only installed packages whose name matches *regexp*.

For each package, print the following items separated by tabs: its name, its version string, the parts of the package (see Section 3.4 [Packages with Multiple Outputs], page 21), and the source location of its definition.

`--list-generations[=`*pattern*`]`
`-l [`*pattern*`]`

Return a list of generations along with their creation dates; for each generation, show the installed packages, with the most recently installed packages shown last. Note that the zeroth generation is never shown.

For each installed package, print the following items, separated by tabs: the name of a package, its version string, the part of the package that is installed (see Section 3.4 [Packages with Multiple Outputs], page 21), and the location of this package in the store.

When *pattern* is used, the command returns only matching generations. Valid patterns include:

- *Integers and comma-separated integers.* Both patterns denote generation numbers. For instance, `--list-generations=1` returns the first one.

  And `--list-generations=1,8,2` outputs three generations in the specified order. Neither spaces nor trailing commas are allowed.

- *Ranges.* `--list-generations=2..9` prints the specified generations and everything in between. Note that the start of a range must be lesser than its end.

  It is also possible to omit the endpoint. For example, `--list-generations=2..`, returns all generations starting from the second one.

- *Durations.* You can also get the last *N* days, weeks, or months by passing an integer along with the first letter of the duration. For example, `--list-generations=20d` lists generations that are up to 20 days old.

`--delete-generations[=`*pattern*`]`
`-d [`*pattern*`]`

When *pattern* is omitted, delete all generations except the current one.

This command accepts the same patterns as `--list-generations`. When *pattern* is specified, delete the matching generations. When *pattern* specifies a duration, generations *older* than the specified duration match. For instance, `--delete-generations=1m` deletes generations that are more than one month old.

If the current generation matches, it is *not* deleted. Also, the zeroth generation is never deleted.

Note that deleting generations prevents roll-back to them. Consequently, this command must be used with care.

Finally, since `guix package` may actually start build processes, it supports all the common build options that `guix build` supports (see Section 6.1 [Invoking guix build], page 60).

3.3 Substitutes

Guix supports transparent source/binary deployment, which means that it can either build things locally, or download pre-built items from a server. We call these pre-built items *substitutes*—they are substitutes for local build results. In many cases, downloading a substitute is much faster than building things locally.

Substitutes can be anything resulting from a derivation build (see Section 5.4 [Derivations], page 47). Of course, in the common case, they are pre-built package binaries, but source tarballs, for instance, which also result from derivation builds, can be available as substitutes.

The `hydra.gnu.org` server is a front-end to a build farm that builds packages from the GNU distribution continuously for some architectures, and makes them available as substitutes. This is the default source of substitutes; it can be overridden by passing the `--substitute-urls` option either to `guix-daemon` (see [`guix-daemon --substitute-urls`], page 9) or to client tools such as `guix package` (see [client `--substitute-urls` option], page 63).

To allow Guix to download substitutes from `hydra.gnu.org`, you must add its public key to the access control list (ACL) of archive imports, using the `guix archive` command (see Section 3.7 [Invoking guix archive], page 24). Doing so implies that you trust `hydra.gnu.org` to not be compromised and to serve genuine substitutes.

This public key is installed along with Guix, in *prefix*`/share/guix/hydra.gnu.org.pub`, where *prefix* is the installation prefix of Guix. If you installed Guix from source, make sure you checked the GPG signature of `guix-0.9.0.tar.gz`, which contains this public key file. Then, you can run something like this:

```
# guix archive --authorize < hydra.gnu.org.pub
```

Once this is in place, the output of a command like `guix build` should change from something like:

```
$ guix build emacs --dry-run
The following derivations would be built:
   /gnu/store/yr7bnx8xwcayd6j95r2clmkdl1qh688w-emacs-24.3.drv
```

```
   /gnu/store/x8qsh1hlhgjx6cwsjyvybnfv2i37z23w-dbus-1.6.4.tar.gz.drv
   /gnu/store/1ixwp12fl950d15h2cj11c73733jay0z-alsa-lib-1.0.27.1.tar.bz2.drv
   /gnu/store/nlma1pw0p603fpfiqy7kn4zm105r5dmw-util-linux-2.21.drv
...
```

to something like:

```
$ guix build emacs --dry-run
The following files would be downloaded:
   /gnu/store/pk3n22lbq6ydamyymqkkz7i69wiwjiwi-emacs-24.3
   /gnu/store/2ygn4ncnhrpr61rssa6z0d9x22si0va3-libjpeg-8d
   /gnu/store/71yz6lgx4dazma9dwn2mcjxaah9w77jq-cairo-1.12.16
   /gnu/store/7zdhgp0n1518lvfn8mb96sxqfmvqrl7v-libxrender-0.9.7
...
```

This indicates that substitutes from `hydra.gnu.org` are usable and will be downloaded, when possible, for future builds.

Guix ignores substitutes that are not signed, or that are not signed by one of the keys listed in the ACL. It also detects and raises an error when attempting to use a substitute that has been tampered with.

The substitute mechanism can be disabled globally by running `guix-daemon` with `--no-substitutes` (see Section 2.5 [Invoking guix-daemon], page 8). It can also be disabled temporarily by passing the `--no-substitutes` option to `guix package`, `guix build`, and other command-line tools.

Today, each individual's control over their own computing is at the mercy of institutions, corporations, and groups with enough power and determination to subvert the computing infrastructure and exploit its weaknesses. While using `hydra.gnu.org` substitutes can be convenient, we encourage users to also build on their own, or even run their own build farm, such that `hydra.gnu.org` is less of an interesting target. One way to help is by publishing the software you build using `guix publish` so that others have one more choice of server to download substitutes from (see Section 6.11 [Invoking guix publish], page 78).

Guix has the foundations to maximize build reproducibility (see Section 3.1 [Features], page 13). In most cases, independent builds of a given package or derivation should yield bit-identical results. Thus, through a diverse set of independent package builds, we can strengthen the integrity of our systems. The `guix challenge` command aims to help users assess substitute servers, and to assist developers in finding out about non-deterministic package builds (see Section 6.12 [Invoking guix challenge], page 79).

In the future, we want Guix to have support to publish and retrieve binaries to/from other users, in a peer-to-peer fashion. If you would like to discuss this project, join us on `guix-devel@gnu.org`.

3.4 Packages with Multiple Outputs

Often, packages defined in Guix have a single *output*—i.e., the source package leads exactly one directory in the store. When running `guix package -i glibc`, one installs the default output of the GNU libc package; the default output is called `out`, but its name can be omitted as shown in this command. In this particular case, the default output of `glibc` contains all the C header files, shared libraries, static libraries, Info documentation, and other supporting files.

Sometimes it is more appropriate to separate the various types of files produced from a single source package into separate outputs. For instance, the GLib C library (used by GTK+ and related packages) installs more than 20 MiB of reference documentation as HTML pages. To save space for users who do not need it, the documentation goes to a separate output, called `doc`. To install the main GLib output, which contains everything but the documentation, one would run:

```
guix package -i glib
```

The command to install its documentation is:

```
guix package -i glib:doc
```

Some packages install programs with different "dependency footprints". For instance, the WordNet package install both command-line tools and graphical user interfaces (GUIs). The former depend solely on the C library, whereas the latter depend on Tcl/Tk and the underlying X libraries. In this case, we leave the command-line tools in the default output, whereas the GUIs are in a separate output. This allows users who do not need the GUIs to save space. The `guix size` command can help find out about such situations (see Section 6.8 [Invoking guix size], page 71). `guix graph` can also be helpful (see Section 6.9 [Invoking guix graph], page 72).

There are several such multiple-output packages in the GNU distribution. Other conventional output names include `lib` for libraries and possibly header files, `bin` for stand-alone programs, and `debug` for debugging information (see Section 7.3 [Installing Debugging Files], page 124). The outputs of a packages are listed in the third column of the output of `guix package --list-available` (see Section 3.2 [Invoking guix package], page 14).

3.5 Invoking `guix gc`

Packages that are installed but not used may be *garbage-collected*. The `guix gc` command allows users to explicitly run the garbage collector to reclaim space from the `/gnu/store` directory. It is the *only* way to remove files from `/gnu/store`—removing files or directories manually may break it beyond repair!

The garbage collector has a set of known *roots*: any file under `/gnu/store` reachable from a root is considered *live* and cannot be deleted; any other file is considered *dead* and may be deleted. The set of garbage collector roots includes default user profiles, and may be augmented with `guix build --root`, for example (see Section 6.1 [Invoking guix build], page 60).

Prior to running `guix gc --collect-garbage` to make space, it is often useful to remove old generations from user profiles; that way, old package builds referenced by those generations can be reclaimed. This is achieved by running `guix package --delete-generations` (see Section 3.2 [Invoking guix package], page 14).

The `guix gc` command has three modes of operation: it can be used to garbage-collect any dead files (the default), to delete specific files (the `--delete` option), to print garbage-collector information, or for more advanced queries. The garbage collection options are as follows:

`--collect-garbage[=`*min*`]`
`-C [`*min*`]` Collect garbage—i.e., unreachable `/gnu/store` files and sub-directories. This is the default operation when no option is specified.

When *min* is given, stop once *min* bytes have been collected. *min* may be a number of bytes, or it may include a unit as a suffix, such as `MiB` for mebibytes and `GB` for gigabytes (see Section "Block size" in *GNU Coreutils*).

When *min* is omitted, collect all the garbage.

`--delete`
`-d` Attempt to delete all the store files and directories specified as arguments. This fails if some of the files are not in the store, or if they are still live.

`--list-failures`
List store items corresponding to cached build failures.

This prints nothing unless the daemon was started with `--cache-failures` (see Section 2.5 [Invoking guix-daemon], page 8).

`--clear-failures`
Remove the specified store items from the failed-build cache.

Again, this option only makes sense when the daemon is started with `--cache-failures`. Otherwise, it does nothing.

`--list-dead`
Show the list of dead files and directories still present in the store—i.e., files and directories no longer reachable from any root.

`--list-live`
Show the list of live store files and directories.

In addition, the references among existing store files can be queried:

`--references`
`--referrers`
List the references (respectively, the referrers) of store files given as arguments.

`--requisites`
`-R` List the requisites of the store files passed as arguments. Requisites include the store files themselves, their references, and the references of these, recursively. In other words, the returned list is the *transitive closure* of the store files.

See Section 6.8 [Invoking guix size], page 71, for a tool to profile the size of an element's closure. See Section 6.9 [Invoking guix graph], page 72, for a tool to visualize the graph of references.

Lastly, the following options allow you to check the integrity of the store and to control disk usage.

`--verify[=`*options*`]`
Verify the integrity of the store.

By default, make sure that all the store items marked as valid in the daemon's database actually exist in `/gnu/store`.

When provided, *options* must a comma-separated list containing one or more of `contents` and `repair`.

When passing `--verify=contents`, the daemon will compute the content hash of each store item and compare it against its hash in the database. Hash

mismatches are reported as data corruptions. Because it traverses *all the files in the store*, this command can take a long time, especially on systems with a slow disk drive.

Using `--verify=repair` or `--verify=contents,repair` causes the daemon to try to repair corrupt store items by fetching substitutes for them (see Section 3.3 [Substitutes], page 20). Because repairing is not atomic, and thus potentially dangerous, it is available only to the system administrator.

`--optimize`

Optimize the store by hard-linking identical files—this is *deduplication*.

The daemon performs deduplication after each successful build or archive import, unless it was started with `--disable-deduplication` (see Section 2.5 [Invoking guix-daemon], page 8). Thus, this option is primarily useful when the daemon was running with `--disable-deduplication`.

3.6 Invoking `guix pull`

Packages are installed or upgraded to the latest version available in the distribution currently available on your local machine. To update that distribution, along with the Guix tools, you must run `guix pull`: the command downloads the latest Guix source code and package descriptions, and deploys it.

On completion, `guix package` will use packages and package versions from this just-retrieved copy of Guix. Not only that, but all the Guix commands and Scheme modules will also be taken from that latest version. New `guix` sub-commands added by the update also become available[1].

The `guix pull` command is usually invoked with no arguments, but it supports the following options:

`--verbose`

Produce verbose output, writing build logs to the standard error output.

`--url=`*`url`*

Download the source tarball of Guix from *url*.

By default, the tarball is taken from its canonical address at `gnu.org`, for the stable branch of Guix.

`--bootstrap`

Use the bootstrap Guile to build the latest Guix. This option is only useful to Guix developers.

3.7 Invoking `guix archive`

The `guix archive` command allows users to *export* files from the store into a single archive, and to later *import* them. In particular, it allows store files to be transferred from one machine to another machine's store. For example, to transfer the `emacs` package to a machine connected over SSH, one would run:

[1] Under the hood, `guix pull` updates the `~/.config/guix/latest` symbolic link to point to the latest Guix, and the `guix` command loads code from there.

```
guix archive --export -r emacs | ssh the-machine guix archive --import
```

Similarly, a complete user profile may be transferred from one machine to another like this:

```
guix archive --export -r $(readlink -f ~/.guix-profile) | \
  ssh the-machine guix-archive --import
```

However, note that, in both examples, all of `emacs` and the profile as well as all of their dependencies are transferred (due to `-r`), regardless of what is already available in the target machine's store. The `--missing` option can help figure out which items are missing from the target's store.

Archives are stored in the "Nix archive" or "Nar" format, which is comparable in spirit to 'tar', but with a few noteworthy differences that make it more appropriate for our purposes. First, rather than recording all Unix meta-data for each file, the Nar format only mentions the file type (regular, directory, or symbolic link); Unix permissions and owner/group are dismissed. Second, the order in which directory entries are stored always follows the order of file names according to the C locale collation order. This makes archive production fully deterministic.

When exporting, the daemon digitally signs the contents of the archive, and that digital signature is appended. When importing, the daemon verifies the signature and rejects the import in case of an invalid signature or if the signing key is not authorized.

The main options are:

`--export` Export the specified store files or packages (see below.) Write the resulting archive to the standard output.

Dependencies are *not* included in the output, unless `--recursive` is passed.

`-r`
`--recursive`
When combined with `--export`, this instructs `guix archive` to include dependencies of the given items in the archive. Thus, the resulting archive is self-contained: it contains the closure of the exported store items.

`--import` Read an archive from the standard input, and import the files listed therein into the store. Abort if the archive has an invalid digital signature, or if it is signed by a public key not among the authorized keys (see `--authorize` below.)

`--missing`
Read a list of store file names from the standard input, one per line, and write on the standard output the subset of these files missing from the store.

`--generate-key[=`*parameters*`]`
Generate a new key pair for the daemons. This is a prerequisite before archives can be exported with `--export`. Note that this operation usually takes time, because it needs to gather enough entropy to generate the key pair.

The generated key pair is typically stored under `/etc/guix`, in `signing-key.pub` (public key) and `signing-key.sec` (private key, which must be kept secret.) When *parameters* is omitted, an ECDSA key using the Ed25519 curve is generated, or, for Libgcrypt versions before 1.6.0, it is a 4096-bit RSA key. Alternately, *parameters* can specify `genkey` parameters suitable for Libgcrypt (see Section "General public-key related Functions" in *The Libgcrypt Reference Manual*).

`--authorize`

Authorize imports signed by the public key passed on standard input. The public key must be in "s-expression advanced format"—i.e., the same format as the `signing-key.pub` file.

The list of authorized keys is kept in the human-editable file `/etc/guix/acl`. The file contains "advanced-format s-expressions" and is structured as an access-control list in the Simple Public-Key Infrastructure (SPKI).

`--extract=`*directory*
`-x` *directory*

Read a single-item archive as served by substitute servers (see Section 3.3 [Substitutes], page 20) and extract it to *directory*. This is a low-level operation needed in only very narrow use cases; see below.

For example, the following command extracts the substitute for Emacs served by `hydra.gnu.org` to `/tmp/emacs`:

```
$ wget -O - \
  http://hydra.gnu.org/nar/...-emacs-24.5 \
  | bunzip2 | guix archive -x /tmp/emacs
```

Single-item archives are different from multiple-item archives produced by `guix archive --export`; they contain a single store item, and they do *not* embed a signature. Thus this operation does *no* signature verification and its output should be considered unsafe.

The primary purpose of this operation is to facilitate inspection of archive contents coming from possibly untrusted substitute servers.

To export store files as an archive to the standard output, run:

```
guix archive --export options specifications...
```

specifications may be either store file names or package specifications, as for `guix package` (see Section 3.2 [Invoking guix package], page 14). For instance, the following command creates an archive containing the `gui` output of the `git` package and the main output of `emacs`:

```
guix archive --export git:gui /gnu/store/...-emacs-24.3 > great.nar
```

If the specified packages are not built yet, `guix archive` automatically builds them. The build process may be controlled with the same options that can be passed to the `guix build` command (see Section 6.1 [Invoking guix build], page 60).

4 Emacs Interface

GNU Guix comes with several useful modules (known as "guix.el") for GNU Emacs which are intended to make an Emacs user interaction with Guix convenient and fun.

4.1 Initial Setup

On the Guix System Distribution (see Chapter 7 [GNU Distribution], page 82), "guix.el" is ready to use, provided Guix is installed system-wide, which is the case by default. So if that is what you're using, you can happily skip this section and read about the fun stuff.

If you're not yet a happy user of GuixSD, a little bit of setup is needed. To be able to use "guix.el", you need to install the following packages:

- GNU Emacs, version 24.3 or later;
- Geiser, version 0.3 or later: it is used for interacting with the Guile process.
- magit-popup library. You already have this library if you use Magit 2.1.0 or later. This library is an optional dependency—it is required only for *`M-x guix`* command (see Section 4.3 [Emacs Popup Interface], page 33).

When it is done "guix.el" may be configured by requiring a special `guix-init` file—i.e., by adding the following code into your init file (see Section "Init File" in *The GNU Emacs Manual*):

```
(add-to-list 'load-path "/path/to/directory-with-guix.el")
(require 'guix-init nil t)
```

So the only thing you need to figure out is where the directory with elisp files for Guix is placed. It depends on how you installed Guix:

- If it was installed by a package manager of your distribution or by a usual `./configure && make && make install` command sequence, then elisp files are placed in a standard directory with Emacs packages (usually it is `/usr/share/emacs/site-lisp/`), which is already in `load-path`, so there is no need to add that directory there.
- If you used a binary installation method (see Section 2.1 [Binary Installation], page 3), then Guix is installed somewhere in the store, so the elisp files are placed in `/gnu/store/...-guix-0.8.2/share/emacs/site-lisp/` or alike. However it is not recommended to refer directly to a store directory. Instead you can install Guix using Guix itself with `guix package -i guix` command (see Section 3.2 [Invoking guix package], page 14) and add `~/.guix-profile/share/emacs/site-lisp/` directory to `load-path` variable.
- If you did not install Guix at all and prefer a hacking way (see Section 8.2 [Running Guix Before It Is Installed], page 135), along with augmenting `load-path` you need to set `guix-load-path` variable to the same directory, so your final configuration will look like this:

```
(let ((dir "/path/to/your-guix-git-tree/emacs"))
  (add-to-list 'load-path dir)
  (setq guix-load-path dir))
(require 'guix-init nil t)
```

By default, along with autoloading (see Section "Autoload" in *The GNU Emacs Lisp Reference Manual*) the main interactive commands for "guix.el" (see Section 4.2.1 [Emacs Commands], page 28), requiring `guix-init` will also autoload commands for the Emacs packages installed in your user profile.

To disable automatic loading of installed Emacs packages, set `guix-package-enable-at-startup` variable to `nil` before requiring `guix-init`. This variable has the same meaning for Emacs packages installed with Guix, as `package-enable-at-startup` for the built-in Emacs package system (see Section "Package Installation" in *The GNU Emacs Manual*).

You can activate Emacs packages installed in your profile whenever you want using *M-x guix-emacs-load-autoloads*.

4.2 Package Management

Once "guix.el" has been successfully configured, you should be able to use a visual interface for routine package management tasks, pretty much like the `guix package` command (see Section 3.2 [Invoking guix package], page 14). Specifically, it makes it easy to:

- browse and display packages and generations;
- search, install, upgrade and remove packages;
- display packages from previous generations;
- do some other useful things.

4.2.1 Commands

All commands for displaying packages and generations use the current profile, which can be changed with *M-x guix-set-current-profile*. Alternatively, if you call any of these commands with prefix argument (*C-u*), you will be prompted for a profile just for that command.

Commands for displaying packages:

M-x guix-all-available-packages
M-x guix-newest-available-packages
: Display all/newest available packages.

M-x guix-installed-packages
: Display all installed packages.

M-x guix-obsolete-packages
: Display obsolete packages (the packages that are installed in a profile but cannot be found among available packages).

M-x guix-search-by-name
: Display package(s) with the specified name.

M-x guix-search-by-regexp
: Search for packages by a specified regexp. By default "name", "synopsis" and "description" of the packages will be searched. This can be changed by modifying `guix-search-params` variable.

By default, these commands display each output on a separate line. If you prefer to see a list of packages—i.e., a list with a package per line, use the following setting:

```
(setq guix-package-list-type 'package)
```

Commands for displaying generations:

M-x guix-generations
: List all the generations.

M-x guix-last-generations
: List the *N* last generations. You will be prompted for the number of generations.

M-x guix-generations-by-time
: List generations matching time period. You will be prompted for the period using Org mode time prompt based on Emacs calendar (see Section "The date/time prompt" in *The Org Manual*).

You can also invoke the `guix pull` command (see Section 3.6 [Invoking guix pull], page 24) from Emacs using:

M-x guix-pull
: With *C-u*, make it verbose.

Once `guix pull` has succeeded, the Guix REPL is restared. This allows you to keep using the Emacs interface with the updated Guix.

4.2.2 General information

The following keys are available for both "list" and "info" types of buffers:

l
r
: Go backward/forward by the history of the displayed results (this history is similar to the history of the Emacs `help-mode` or `Info-mode`).

g
: Revert current buffer: update information about the displayed packages/generations and redisplay it.

R
: Redisplay current buffer (without updating information).

M
: Apply manifest to the current profile or to a specified profile, if prefix argument is used. This has the same meaning as `--manifest` option (see Section 3.2 [Invoking guix package], page 14).

C-c C-z
: Go to the Guix REPL (see Section "The REPL" in *Geiser User Manual*).

h
?
: Describe current mode to see all available bindings.

Hint: If you need several "list" or "info" buffers, you can simlpy *M-x clone-buffer* them, and each buffer will have its own history.

Warning: Name/version pairs cannot be used to identify packages (because a name is not necessarily unique), so "guix.el" uses special identifiers that live only during a guile session, so if the Guix REPL was restarted, you may want to revert "list" buffer (by pressing *g*).

4.2.3 “List” buffer

An interface of a “list” buffer is similar to the interface provided by “package.el” (see Section “Package Menu” in *The GNU Emacs Manual*).

Default key bindings available for both “package-list” and “generation-list” buffers:

`m` Mark the current entry (with prefix, mark all entries).

`u` Unmark the current entry (with prefix, unmark all entries).

`DEL` Unmark backward.

`S` Sort entries by a specified column.

A “package-list” buffer additionally provides the following bindings:

`RET` Describe marked packages (display available information in a “package-info” buffer).

`i` Mark the current package for installation.

`d` Mark the current package for deletion.

`U` Mark the current package for upgrading.

`^` Mark all obsolete packages for upgrading.

`e` Edit the definition of the curent package (go to its location). This is similar to `guix edit` command (see Section 6.2 [Invoking guix edit], page 64), but for opening a package recipe in the current Emacs instance.

`x` Execute actions on the marked packages.

A “generation-list” buffer additionally provides the following bindings:

`RET` List packages installed in the current generation.

`i` Describe marked generations (display available information in a “generation-info” buffer).

`s` Switch profile to the current generation.

`d` Mark the current generation for deletion (with prefix, mark all generations).

`x` Execute actions on the marked generations—i.e., delete generations.

`e` Run Ediff (see *The Ediff Manual*) on package outputs installed in the 2 marked generations. With prefix argument, run Ediff on manifests of the marked generations.

`D`
`=` Run Diff (see Section “Diff Mode” in *The GNU Emacs Manual*) on package outputs installed in the 2 marked generations. With prefix argument, run Diff on manifests of the marked generations.

`+` List package outputs added to the latest marked generation comparing with another marked generation.

`-` List package outputs removed from the latest marked generation comparing with another marked generation.

4.2.4 “Info” buffer

The interface of an “info” buffer is similar to the interface of `help-mode` (see Section “Help Mode” in *The GNU Emacs Manual*).

“Info” buffer contains some buttons (as usual you may use `TAB` / *`S-TAB`* to move between buttons—see Section “Mouse References” in *The GNU Emacs Manual*) which can be used to:

- (in a “package-info” buffer)
  - install/remove a package;
  - jump to a package location;
  - browse home page of a package;
  - describe packages from “Inputs” fields.
- (in a “generation-info” buffer)
  - remove a generation;
  - switch to a generation;
  - list packages installed in a generation;
  - jump to a generation directory.

It is also possible to copy a button label (a link to an URL or a file) by pressing *`c`* on a button.

4.2.5 Configuration

There are many variables you can modify to change the appearance or behavior of Emacs user interface. Some of these variables are described in this section. Also you can use Custom Interface (see Section “Easy Customization” in *The GNU Emacs Manual*) to explore/set variables (not all) and faces.

4.2.5.1 Guile and Build Options

`guix-guile-program`
: If you have some special needs for starting a Guile process, you may set this variable, for example:

```
(setq guix-guile-program '("/bin/guile" "--no-auto-compile"))
```

`guix-use-substitutes`
: Has the same meaning as `--no-substitutes` option (see Section 6.1 [Invoking guix build], page 60).

`guix-dry-run`
: Has the same meaning as `--dry-run` option (see Section 6.1 [Invoking guix build], page 60).

4.2.5.2 Buffer Names

Default names of “guix.el” buffers (“*Guix . . .*”) may be changed with the following variables:

```
guix-package-list-buffer-name
guix-output-list-buffer-name
guix-generation-list-buffer-name
guix-package-info-buffer-name
guix-output-info-buffer-name
guix-generation-info-buffer-name
guix-repl-buffer-name
guix-internal-repl-buffer-name
```

By default, the name of a profile is also displayed in a "list" or "info" buffer name. To change this behavior, use `guix-buffer-name-function` variable.

For example, if you want to display all types of results in a single buffer (in such case you will probably use a history (*`l`*/*`r`*) extensively), you may do it like this:

```
(let ((name "Guix Universal"))
  (setq
   guix-package-list-buffer-name    name
   guix-output-list-buffer-name     name
   guix-generation-list-buffer-name name
   guix-package-info-buffer-name    name
   guix-output-info-buffer-name     name
   guix-generation-info-buffer-name name
   guix-buffer-name-function        #'guix-buffer-name-simple))
```

4.2.5.3 Keymaps

If you want to change default key bindings, use the following keymaps (see Section "Init Rebinding" in *The GNU Emacs Manual*):

`guix-root-map`
: Parent keymap with general keys for all guix modes.

`guix-list-mode-map`
: Parent keymap with general keys for "list" buffers.

`guix-package-list-mode-map`
: Keymap with specific keys for "package-list" buffers.

`guix-output-list-mode-map`
: Keymap with specific keys for "output-list" buffers.

`guix-generation-list-mode-map`
: Keymap with specific keys for "generation-list" buffers.

`guix-info-mode-map`
: Parent keymap with general keys for "info" buffers.

`guix-package-info-mode-map`
: Keymap with specific keys for "package-info" buffers.

`guix-output-info-mode-map`
: Keymap with specific keys for "output-info" buffers.

`guix-generation-info-mode-map`
: Keymap with specific keys for "generation-info" buffers.

`guix-info-button-map`
: Keymap with keys available when a point is placed on a button.

4.2.5.4 Appearance

You can change almost any aspect of "list" / "info" buffers using the following variables:

`guix-list-column-format`
`guix-list-column-titles`
`guix-list-column-value-methods`
: Specify the columns, their names, what and how is displayed in "list" buffers.

`guix-info-displayed-params`
`guix-info-insert-methods`
`guix-info-ignore-empty-vals`
`guix-info-param-title-format`
`guix-info-multiline-prefix`
`guix-info-indent`
`guix-info-fill-column`
`guix-info-delimiter`
: Various settings for "info" buffers.

4.3 Popup Interface

If you ever used Magit, you know what "popup interface" is (see *Magit-Popup User Manual*). Even if you are not acquainted with Magit, there should be no worries as it is very intuitive.

So *`M-x guix`* command provides a top-level popup interface for all available guix commands. When you select an option, you'll be prompted for a value in the minibuffer. Many values have completions, so don't hesitate to press `TAB` key. Multiple values (for example, packages or lint checkers) should be separated by commas.

After specifying all options and switches for a command, you may choose one of the available actions. The following default actions are available for all commands:

- Run the command in the Guix REPL. It is faster than running `guix ...` command directly in shell, as there is no need to run another guile process and to load required modules there.
- Run the command in a shell buffer. You can set `guix-run-in-shell-function` variable to fine tune the shell buffer you want to use.
- Add the command line to the kill ring (see Section "Kill Ring" in *The GNU Emacs Manual*).

Several commands (`guix graph`, `guix system dmd-graph` and `guix system extension-graph`) also have a "View graph" action, which allows you to view a generated graph using `dot` command (specified by `guix-dot-program` variable). By default a PNG file will be saved in `/tmp` directory and will be opened directly in Emacs. This behavior may be changed with the following variables:

`guix-find-file-function`
: Function used to open a generated graph. If you want to open a graph in an external program, you can do it by modifying this variable—for example, you can use a functionality provided by the Org Mode (see *The Org Manual*):

```
(setq guix-find-file-function 'org-open-file)
(add-to-list 'org-file-apps '("\\.png\\'" . "sxiv %s"))
```

`guix-dot-default-arguments`
: Command line arguments to run `dot` command. If you change an output format (for example, into `-Tpdf`), you also need to change the next variable.

`guix-dot-file-name-function`
: Function used to define a name of the generated graph file. Default name is `/tmp/guix-emacs-graph-XXXXXX.png`.

So, for example, if you want to generate and open a PDF file in your Emacs, you may change the settings like this:

```
(defun my-guix-pdf-graph ()
  "/tmp/my-current-guix-graph.pdf")

(setq guix-dot-default-arguments '("-Tpdf")
      guix-dot-file-name-function 'my-guix-pdf-graph)
```

4.4 Guix Prettify Mode

GNU Guix also comes with "guix-prettify.el". It provides a minor mode for abbreviating store file names by replacing hash sequences of symbols with ". . .":

```
/gnu/store/72f54nfp6g1hz873w8z3gfcah0h4nl9p-foo-0.1
⇒ /gnu/store/...-foo-0.1
```

Once you set up "guix.el" (see Section 4.1 [Emacs Initial Setup], page 27), the following commands become available:

M-x guix-prettify-mode
: Enable/disable prettifying for the current buffer.

M-x global-guix-prettify-mode
: Enable/disable prettifying globally.

To automatically enable `guix-prettify-mode` globally on Emacs start, add the following line to your init file:

```
(global-guix-prettify-mode)
```

If you want to enable it only for specific major modes, add it to the mode hooks (see Section "Hooks" in *The GNU Emacs Manual*), for example:

```
(add-hook 'shell-mode-hook 'guix-prettify-mode)
(add-hook 'dired-mode-hook 'guix-prettify-mode)
```

4.5 Build Log Mode

GNU Guix provides major and minor modes for highlighting build logs. So when you have a file with a package build output—for example, a file returned by `guix build --log-file ...` command (see Section 6.1 [Invoking guix build], page 60), you may call *M-x guix-build-log-mode* command in the buffer with this file. This major mode highlights some lines specific to build output and provides the following key bindings:

M-n Move to the next build phase.

M-p Move to the previous build phase.

TAB Toggle (show/hide) the body of the current build phase.

S-TAB Toggle (show/hide) the bodies of all build phases.

There is also *M-x guix-build-log-minor-mode* which also provides the same highlighting and the same key bindings as the major mode, but prefixed with *C-c*. By default, this minor mode is enabled in shell buffers (see Section "Interactive Shell" in *The GNU Emacs Manual*). If you don't like it, set `guix-build-log-minor-mode-activate` to nil.

4.6 Shell Completions

Another feature that becomes available after configuring Emacs interface (see Section 4.1 [Emacs Initial Setup], page 27) is completing of `guix` subcommands, options, packages and other things in `shell` (see Section "Interactive Shell" in *The GNU Emacs Manual*) and `eshell` (see *Eshell: The Emacs Shell*).

It works the same way as other completions do. Just press `TAB` when your intuition tells you.

And here are some examples, where pressing `TAB` may complete something:

```
guix paTAB
guix package -TAB
guix package --TAB
guix package -i geiTAB
guix build -L/tmTAB
guix build --syTAB
guix build --system=iTAB
guix system recTAB
guix lint --checkers=syTAB
guix lint --checkers=synopsis,desTAB
```

4.7 Development

By default, when you open a Scheme file, `guix-devel-mode` will be activated (if you don't want it, set `guix-devel-activate-mode` to nil). This minor mode provides the following key bindings:

C-c . k Copy the name of the current Guile module into kill ring (`guix-devel-copy-module-as-kill`).

C-c . u Use the current Guile module. Often after opening a Scheme file, you want to use a module it defines, so you switch to the Geiser REPL and write `,use (some module)` there. You may just use this command instead (`guix-devel-use-module`).

C-c . b Build a package defined by the current variable definition. The building process is run in the current Geiser REPL. If you modified the current package

definition, don't forget to reevaluate it before calling this command—for example, with `C-M-x` (see Section "To eval or not to eval" in *Geiser User Manual*) (`guix-devel-build-package-definition`).

`C-c . s` Build a source derivation of the package defined by the current variable definition. This command has the same meaning as `guix build -S` shell command (see Section 6.1 [Invoking guix build], page 60) (`guix-devel-build-package-source`).

`C-c . l` Lint (check) a package defined by the current variable definition (see Section 6.7 [Invoking guix lint], page 70) (`guix-devel-lint-package`).

Unluckily, there is a limitation related to long-running REPL commands. When there is a running process in a Geiser REPL, you are not supposed to evaluate anything in a scheme buffer, because this will "freeze" the REPL: it will stop producing any output (however, the evaluating process will continue—you will just not see any progress anymore). Be aware: even moving the point in a scheme buffer may "break" the REPL if Autodoc (see Section "Autodoc and friends" in *Geiser User Manual*) is enabled (which is the default).

So you have to postpone editing your scheme buffers until the running evaluation will be finished in the REPL.

Alternatively, to avoid this limitation, you may just run another Geiser REPL, and while something is being evaluated in the previous REPL, you can continue editing a scheme file with the help of the current one.

5 Programming Interface

GNU Guix provides several Scheme programming interfaces (APIs) to define, build, and query packages. The first interface allows users to write high-level package definitions. These definitions refer to familiar packaging concepts, such as the name and version of a package, its build system, and its dependencies. These definitions can then be turned into concrete build actions.

Build actions are performed by the Guix daemon, on behalf of users. In a standard setup, the daemon has write access to the store—the `/gnu/store` directory—whereas users do not. The recommended setup also has the daemon perform builds in chroots, under a specific build users, to minimize interference with the rest of the system.

Lower-level APIs are available to interact with the daemon and the store. To instruct the daemon to perform a build action, users actually provide it with a *derivation*. A derivation is a low-level representation of the build actions to be taken, and the environment in which they should occur—derivations are to package definitions what assembly is to C programs. The term "derivation" comes from the fact that build results *derive* from them.

This chapter describes all these APIs in turn, starting from high-level package definitions.

5.1 Defining Packages

The high-level interface to package definitions is implemented in the `(guix packages)` and `(guix build-system)` modules. As an example, the package definition, or *recipe*, for the GNU Hello package looks like this:

```
(define-module (gnu packages hello)
  #:use-module (guix packages)
  #:use-module (guix download)
  #:use-module (guix build-system gnu)
  #:use-module (guix licenses)
  #:use-module (gnu packages gawk))

(define-public hello
  (package
    (name "hello")
    (version "2.10")
    (source (origin
              (method url-fetch)
              (uri (string-append "mirror://gnu/hello/hello-" version
                                  ".tar.gz"))
              (sha256
               (base32
                "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
    (build-system gnu-build-system)
    (arguments `(#:configure-flags '("--enable-silent-rules")))
    (inputs `(("gawk" ,gawk)))
    (synopsis "Hello, GNU world: An example GNU package")
    (description "Guess what GNU Hello prints!")
```

```
        (home-page "http://www.gnu.org/software/hello/")
        (license gpl3+)))
```

Without being a Scheme expert, the reader may have guessed the meaning of the various fields here. This expression binds variable `hello` to a `<package>` object, which is essentially a record (see Section "SRFI-9" in *GNU Guile Reference Manual*). This package object can be inspected using procedures found in the `(guix packages)` module; for instance, `(package-name hello)` returns—surprise!—`"hello"`.

With luck, you may be able to import part or all of the definition of the package you are interested in from another repository, using the `guix import` command (see Section 6.5 [Invoking guix import], page 66).

In the example above, *hello* is defined into a module of its own, `(gnu packages hello)`. Technically, this is not strictly necessary, but it is convenient to do so: all the packages defined in modules under `(gnu packages ...)` are automatically known to the command-line tools (see Section 7.5 [Package Modules], page 126).

There are a few points worth noting in the above package definition:

- The `source` field of the package is an `<origin>` object (see Section 5.1.2 [origin Reference], page 41, for the complete reference). Here, the `url-fetch` method from `(guix download)` is used, meaning that the source is a file to be downloaded over FTP or HTTP.

  The `mirror://gnu` prefix instructs `url-fetch` to use one of the GNU mirrors defined in `(guix download)`.

  The `sha256` field specifies the expected SHA256 hash of the file being downloaded. It is mandatory, and allows Guix to check the integrity of the file. The `(base32 ...)` form introduces the base32 representation of the hash. You can obtain this information with `guix download` (see Section 6.3 [Invoking guix download], page 65) and `guix hash` (see Section 6.4 [Invoking guix hash], page 65).

  When needed, the `origin` form can also have a `patches` field listing patches to be applied, and a `snippet` field giving a Scheme expression to modify the source code.

- The `build-system` field specifies the procedure to build the package (see Section 5.2 [Build Systems], page 42). Here, *gnu-build-system* represents the familiar GNU Build System, where packages may be configured, built, and installed with the usual `./configure && make && make check && make install` command sequence.

- The `arguments` field specifies options for the build system (see Section 5.2 [Build Systems], page 42). Here it is interpreted by *gnu-build-system* as a request run `configure` with the `--enable-silent-rules` flag.

- The `inputs` field specifies inputs to the build process—i.e., build-time or run-time dependencies of the package. Here, we define an input called `"gawk"` whose value is that of the *gawk* variable; *gawk* is itself bound to a `<package>` object.

  Note that GCC, Coreutils, Bash, and other essential tools do not need to be specified as inputs here. Instead, *gnu-build-system* takes care of ensuring that they are present (see Section 5.2 [Build Systems], page 42).

  However, any other dependencies need to be specified in the `inputs` field. Any dependency not specified here will simply be unavailable to the build process, possibly leading to a build failure.

See Section 5.1.1 [package Reference], page 39, for a full description of possible fields.

Once a package definition is in place, the package may actually be built using the `guix build` command-line tool (see Section 6.1 [Invoking guix build], page 60). You can easily jump back to the package definition using the `guix edit` command (see Section 6.2 [Invoking guix edit], page 64). See Section 7.6 [Packaging Guidelines], page 127, for more information on how to test package definitions, and Section 6.7 [Invoking guix lint], page 70, for information on how to check a definition for style conformance.

Eventually, updating the package definition to a new upstream version can be partly automated by the `guix refresh` command (see Section 6.6 [Invoking guix refresh], page 68).

Behind the scenes, a derivation corresponding to the `<package>` object is first computed by the `package-derivation` procedure. That derivation is stored in a `.drv` file under `/gnu/store`. The build actions it prescribes may then be realized by using the `build-derivations` procedure (see Section 5.3 [The Store], page 46).

`package-derivation` *store package* [*system*] [Scheme Procedure]

Return the `<derivation>` object of *package* for *system* (see Section 5.4 [Derivations], page 47).

package must be a valid `<package>` object, and *system* must be a string denoting the target system type—e.g., `"x86_64-linux"` for an x86_64 Linux-based GNU system. *store* must be a connection to the daemon, which operates on the store (see Section 5.3 [The Store], page 46).

Similarly, it is possible to compute a derivation that cross-builds a package for some other system:

`package-cross-derivation` *store package target* [*system*] [Scheme Procedure]

Return the `<derivation>` object of *package* cross-built from *system* to *target*.

target must be a valid GNU triplet denoting the target hardware and operating system, such as `"mips64el-linux-gnu"` (see Section "Configuration Names" in *GNU Configure and Build System*).

5.1.1 `package` Reference

This section summarizes all the options available in `package` declarations (see Section 5.1 [Defining Packages], page 37).

`package` [Data Type]

This is the data type representing a package recipe.

`name` The name of the package, as a string.

`version` The version of the package, as a string.

`source` An origin object telling how the source code for the package should be acquired (see Section 5.1.2 [origin Reference], page 41).

`build-system`
The build system that should be used to build the package (see Section 5.2 [Build Systems], page 42).

`arguments` (default: `'()`)
: The arguments that should be passed to the build system. This is a list, typically containing sequential keyword-value pairs.

`inputs` (default: `'()`)
: Package or derivation inputs to the build. This is a list of lists, where each list has the name of the input (a string) as its first element, a package or derivation object as its second element, and optionally the name of the output of the package or derivation that should be used, which defaults to `"out"`.

`propagated-inputs` (default: `'()`)
: This field is like `inputs`, but the specified packages will be force-installed alongside the package they belong to (see [package-cmd-propagated-inputs], page 15, for information on how `guix package` deals with propagated inputs.)

  For example this is necessary when a library needs headers of another library to compile, or needs another shared library to be linked alongside itself when a program wants to link to it.

`native-inputs` (default: `'()`)
: This field is like `inputs`, but in case of a cross-compilation it will be ensured that packages for the architecture of the build machine are present, such that executables from them can be used during the build.

  This is typically where you would list tools needed at build time but not at run time, such as Autoconf, Automake, pkg-config, Gettext, or Bison. `guix lint` can report likely mistakes in this area (see Section 6.7 [Invoking guix lint], page 70).

`self-native-input?` (default: `#f`)
: This is a Boolean field telling whether the package should use itself as a native input when cross-compiling.

`outputs` (default: `'("out")`)
: The list of output names of the package. See Section 3.4 [Packages with Multiple Outputs], page 21, for typical uses of additional outputs.

`native-search-paths` (default: `'()`)
`search-paths` (default: `'()`)
: A list of `search-path-specification` objects describing search-path environment variables honored by the package.

`replacement` (default: `#f`)
: This must either `#f` or a package object that will be used as a *replacement* for this package. See Section 7.4 [Security Updates], page 125, for details.

`synopsis` A one-line description of the package.

`description`
: A more elaborate description of the package.

`license` The license of the package; a value from `(guix licenses)`.

`home-page`
: The URL to the home-page of the package, as a string.

`supported-systems` (default: *%supported-systems*)
: The list of systems supported by the package, as strings of the form `architecture-kernel`, for example `"x86_64-linux"`.

`maintainers` (default: `'()`)
: The list of maintainers of the package, as `maintainer` objects.

`location` (default: source location of the `package` form)
: The source location of the package. It's useful to override this when inheriting from another package, in which case this field is not automatically corrected.

5.1.2 `origin` Reference

This section summarizes all the options available in `origin` declarations (see Section 5.1 [Defining Packages], page 37).

`origin` [Data Type]
: This is the data type representing a source code origin.

`uri`
: An object containing the URI of the source. The object type depends on the `method` (see below). For example, when using the *url-fetch* method of `(guix download)`, the valid `uri` values are: a URL represented as a string, or a list thereof.

`method`
: A procedure that will handle the URI.

  Examples include:

  url-fetch from `(guix download)`
  : download a file the HTTP, HTTPS, or FTP URL specified in the `uri` field;

  git-fetch from `(guix git-download)`
  : clone the Git version control repository, and check out the revision specified in the `uri` field as a `git-reference` object; a `git-reference` looks like this:

```
(git-reference
  (url "git://git.debian.org/git/pkg-shadow/shadow")
  (commit "v4.1.5.1"))
```

`sha256`
: A bytevector containing the SHA-256 hash of the source. Typically the `base32` form is used here to generate the bytevector from a base-32 string.

`file-name` (default: `#f`)
: The file name under which the source code should be saved. When this is `#f`, a sensible default value will be used in most cases. In case the source is fetched from a URL, the file name from the URL will be used. For version control checkouts, it's recommended to provide the file name explicitly because the default is not very descriptive.

`patches` (default: `'()`)
: A list of file names containing patches to be applied to the source.

`snippet` (default: `#f`)
: A quoted piece of code that will be run in the source directory to make any modifications, which is sometimes more convenient than a patch.

`patch-flags` (default: `'("-p1")`)
: A list of command-line flags that should be passed to the `patch` command.

`patch-inputs` (default: `#f`)
: Input packages or derivations to the patching process. When this is `#f`, the usual set of inputs necessary for patching are provided, such as GNU Patch.

`modules` (default: `'()`)
: A list of Guile modules that should be loaded during the patching process and while running the code in the `snippet` field.

`imported-modules` (default: `'()`)
: The list of Guile modules to import in the patch derivation, for use by the `snippet`.

`patch-guile` (default: `#f`)
: The Guile package that should be used in the patching process. When this is `#f`, a sensible default is used.

5.2 Build Systems

Each package definition specifies a *build system* and arguments for that build system (see Section 5.1 [Defining Packages], page 37). This `build-system` field represents the build procedure of the package, as well implicit dependencies of that build procedure.

Build systems are `<build-system>` objects. The interface to create and manipulate them is provided by the `(guix build-system)` module, and actual build systems are exported by specific modules.

Under the hood, build systems first compile package objects to *bags*. A *bag* is like a package, but with less ornamentation—in other words, a bag is a lower-level representation of a package, which includes all the inputs of that package, including some that were implicitly added by the build system. This intermediate representation is then compiled to a derivation (see Section 5.4 [Derivations], page 47).

Build systems accept an optional list of *arguments*. In package definitions, these are passed *via* the `arguments` field (see Section 5.1 [Defining Packages], page 37). They are typically keyword arguments (see Section "Optional Arguments" in *GNU Guile Reference Manual*). The value of these arguments is usually evaluated in the *build stratum*—i.e., by a Guile process launched by the daemon (see Section 5.4 [Derivations], page 47).

The main build system is *gnu-build-system*, which implements the standard build procedure for GNU packages and many other packages. It is provided by the `(guix build-system gnu)` module.

`gnu-build-system` [Scheme Variable]

gnu-build-system represents the GNU Build System, and variants thereof (see Section "Configuration" in *GNU Coding Standards*).

In a nutshell, packages using it configured, built, and installed with the usual `./configure && make && make check && make install` command sequence. In practice, a few additional steps are often needed. All these steps are split up in separate *phases*, notably[1]:

`unpack` Unpack the source tarball, and change the current directory to the extracted source tree. If the source is actually a directory, copy it to the build tree, and enter that directory.

`patch-source-shebangs`
Patch shebangs encountered in source files so they refer to the right store file names. For instance, this changes `#!/bin/sh` to `#!/gnu/store/...-bash-4.3/bin/sh`.

`configure`
Run the `configure` script with a number of default options, such as `--prefix=/gnu/store/...`, as well as the options specified by the `#:configure-flags` argument.

`build` Run `make` with the list of flags specified with `#:make-flags`. If the `#:parallel-builds?` argument is true (the default), build with `make -j`.

`check` Run `make check`, or some other target specified with `#:test-target`, unless `#:tests? #f` is passed. If the `#:parallel-tests?` argument is true (the default), run `make check -j`.

`install` Run `make install` with the flags listed in `#:make-flags`.

`patch-shebangs`
Patch shebangs on the installed executable files.

`strip` Strip debugging symbols from ELF files (unless `#:strip-binaries?` is false), copying them to the `debug` output when available (see Section 7.3 [Installing Debugging Files], page 124).

The build-side module `(guix build gnu-build-system)` defines *%standard-phases* as the default list of build phases. *%standard-phases* is a list of symbol/procedure pairs, where the procedure implements the actual phase.

The list of phases used for a particular package can be changed with the `#:phases` parameter. For instance, passing:

```
#:phases (alist-delete 'configure %standard-phases)
```

means that all the phases described above will be used, except the `configure` phase.

In addition, this build system ensures that the "standard" environment for GNU packages is available. This includes tools such as GCC, libc, Coreutils, Bash, Make, Diffutils, grep, and sed (see the `(guix build-system gnu)` module for a complete

[1] Please see the `(guix build gnu-build-system)` modules for more details about the build phases.

list.) We call these the *implicit inputs* of a package, because package definitions don't have to mention them.

Other `<build-system>` objects are defined to support other conventions and tools used by free software packages. They inherit most of *gnu-build-system*, and differ mainly in the set of inputs implicitly added to the build process, and in the list of phases executed. Some of these build systems are listed below.

`cmake-build-system` [Scheme Variable]

This variable is exported by `(guix build-system cmake)`. It implements the build procedure for packages using the CMake build tool.

It automatically adds the `cmake` package to the set of inputs. Which package is used can be specified with the `#:cmake` parameter.

The `#:configure-flags` parameter is taken as a list of flags passed to the `cmake` command. The `#:build-type` parameter specifies in abstract terms the flags passed to the compiler; it defaults to `"RelWithDebInfo"` (short for "release mode with debugging information"), which roughly means that code is compiled with `-O2 -g`, as is the case for Autoconf-based packages by default.

`glib-or-gtk-build-system` [Scheme Variable]

This variable is exported by `(guix build-system glib-or-gtk)`. It is intended for use with packages making use of GLib or GTK+.

This build system adds the following two phases to the ones defined by *gnu-build-system*:

`glib-or-gtk-wrap`

The phase `glib-or-gtk-wrap` ensures that programs found under `bin/` are able to find GLib's "schemas" and GTK+ modules. This is achieved by wrapping the programs in launch scripts that appropriately set the `XDG_DATA_DIRS` and `GTK_PATH` environment variables.

It is possible to exclude specific package outputs from that wrapping process by listing their names in the `#:glib-or-gtk-wrap-excluded-outputs` parameter. This is useful when an output is known not to contain any GLib or GTK+ binaries, and where wrapping would gratuitously add a dependency of that output on GLib and GTK+.

`glib-or-gtk-compile-schemas`

The phase `glib-or-gtk-compile-schemas` makes sure that all GLib's GSettings schemas are compiled. Compilation is performed by the `glib-compile-schemas` program. It is provided by the package `glib:bin` which is automatically imported by the build system. The `glib` package providing `glib-compile-schemas` can be specified with the `#:glib` parameter.

Both phases are executed after the `install` phase.

`python-build-system` [Scheme Variable]

This variable is exported by `(guix build-system python)`. It implements the more or less standard build procedure used by Python packages, which

consists in running `python setup.py build` and then `python setup.py install --prefix=/gnu/store/...`.

For packages that install stand-alone Python programs under `bin/`, it takes care of wrapping these programs so their `PYTHONPATH` environment variable points to all the Python libraries they depend on.

Which Python package is used can be specified with the `#:python` parameter.

`perl-build-system` [Scheme Variable]

This variable is exported by `(guix build-system perl)`. It implements the standard build procedure for Perl packages, which either consists in running `perl Build.PL --prefix=/gnu/store/...`, followed by `Build` and `Build install`; or in running `perl Makefile.PL PREFIX=/gnu/store/...`, followed by `make` and `make install`; depending on which of `Build.PL` or `Makefile.PL` is present in the package distribution. Preference is given to the former if both `Build.PL` and `Makefile.PL` exist in the package distribution. This preference can be reversed by specifying `#t` for the `#:make-maker?` parameter.

The initial `perl Makefile.PL` or `perl Build.PL` invocation passes flags specified by the `#:make-maker-flags` or `#:module-build-flags` parameter, respectively.

Which Perl package is used can be specified with `#:perl`.

`r-build-system` [Scheme Variable]

This variable is exported by `(guix build-system r)`. It implements the build procedure used by R packages, which essentially is little more than running `R CMD INSTALL --library=/gnu/store/...` in an environment where `R_LIBS_SITE` contains the paths to all R package inputs. Tests are run after installation using the R function `tools::testInstalledPackage`.

`ruby-build-system` [Scheme Variable]

This variable is exported by `(guix build-system ruby)`. It implements the RubyGems build procedure used by Ruby packages, which involves running `gem build` followed by `gem install`.

The `source` field of a package that uses this build system typically references a gem archive, since this is the format that Ruby developers use when releasing their software. The build system unpacks the gem archive, potentially patches the source, runs the test suite, repackages the gem, and installs it. Additionally, directories and tarballs may be referenced to allow building unreleased gems from Git or a traditional source release tarball.

Which Ruby package is used can be specified with the `#:ruby` parameter. A list of additional flags to be passed to the `gem` command can be specified with the `#:gem-flags` parameter.

`waf-build-system` [Scheme Variable]

This variable is exported by `(guix build-system waf)`. It implements a build procedure around the `waf` script. The common phases—`configure`, `build`, and `install`—are implemented by passing their names as arguments to the `waf` script.

The `waf` script is executed by the Python interpreter. Which Python package is used to run the script can be specified with the `#:python` parameter.

`haskell-build-system` [Scheme Variable]

This variable is exported by `(guix build-system haskell)`. It implements the Cabal build procedure used by Haskell packages, which involves running `runhaskell Setup.hs configure --prefix=/gnu/store/...` and `runhaskell Setup.hs build`. Instead of installing the package by running `runhaskell Setup.hs install`, to avoid trying to register libraries in the read-only compiler store directory, the build system uses `runhaskell Setup.hs copy`, followed by `runhaskell Setup.hs register`. In addition, the build system generates the package documentation by running `runhaskell Setup.hs haddock`, unless `#:haddock? #f` is passed. Optional Haddock parameters can be passed with the help of the `#:haddock-flags` parameter. If the file `Setup.hs` is not found, the build system looks for `Setup.lhs` instead.

Which Haskell compiler is used can be specified with the `#:haskell` parameter which defaults to `ghc`.

`emacs-build-system` [Scheme Variable]

This variable is exported by `(guix build-system emacs)`. It implements an installation procedure similar to the one of Emacs' own packaging system (see Section "Packages" in *The GNU Emacs Manual*).

It first creates the *package*`-autoloads.el` file, then it byte compiles all Emacs Lisp files. Differently from the Emacs packaging system, the Info documentation files are moved to the standard documentation directory and the `dir` file is deleted. Each package is installed in its own directory under `share/emacs/site-lisp/guix.d`.

Lastly, for packages that do not need anything as sophisticated, a "trivial" build system is provided. It is trivial in the sense that it provides basically no support: it does not pull any implicit inputs, and does not have a notion of build phases.

`trivial-build-system` [Scheme Variable]

This variable is exported by `(guix build-system trivial)`.

This build system requires a `#:builder` argument. This argument must be a Scheme expression that builds the package's output(s)—as with `build-expression->derivation` (see Section 5.4 [Derivations], page 47).

5.3 The Store

Conceptually, the *store* is where derivations that have been successfully built are stored—by default, under `/gnu/store`. Sub-directories in the store are referred to as *store paths*. The store has an associated database that contains information such as the store paths referred to by each store path, and the list of *valid* store paths—paths that result from a successful build.

The store is always accessed by the daemon on behalf of its clients (see Section 2.5 [Invoking guix-daemon], page 8). To manipulate the store, clients connect to the daemon over a Unix-domain socket, send it requests, and read the result—these are remote procedure calls, or RPCs.

The `(guix store)` module provides procedures to connect to the daemon, and to perform RPCs. These are described below.

`open-connection` [*file*] [*#:reserve-space? #t*] [Scheme Procedure]
Connect to the daemon over the Unix-domain socket at *file*. When *reserve-space?* is true, instruct it to reserve a little bit of extra space on the file system so that the garbage collector can still operate, should the disk become full. Return a server object.

file defaults to *%default-socket-path*, which is the normal location given the options that were passed to `configure`.

`close-connection` *server* [Scheme Procedure]
Close the connection to *server*.

`current-build-output-port` [Scheme Variable]
This variable is bound to a SRFI-39 parameter, which refers to the port where build and error logs sent by the daemon should be written.

Procedures that make RPCs all take a server object as their first argument.

`valid-path?` *server path* [Scheme Procedure]
Return `#t` when *path* is a valid store path.

`add-text-to-store` *server name text* [*references*] [Scheme Procedure]
Add *text* under file *name* in the store, and return its store path. *references* is the list of store paths referred to by the resulting store path.

`build-derivations` *server derivations* [Scheme Procedure]
Build *derivations* (a list of `<derivation>` objects or derivation paths), and return when the worker is done building them. Return `#t` on success.

Note that the `(guix monads)` module provides a monad as well as monadic versions of the above procedures, with the goal of making it more convenient to work with code that accesses the store (see Section 5.5 [The Store Monad], page 50).

This section is currently incomplete.

5.4 Derivations

Low-level build actions and the environment in which they are performed are represented by *derivations*. A derivation contain the following pieces of information:

- The outputs of the derivation—derivations produce at least one file or directory in the store, but may produce more.
- The inputs of the derivations, which may be other derivations or plain files in the store (patches, build scripts, etc.)
- The system type targeted by the derivation—e.g., `x86_64-linux`.
- The file name of a build script in the store, along with the arguments to be passed.
- A list of environment variables to be defined.

Derivations allow clients of the daemon to communicate build actions to the store. They exist in two forms: as an in-memory representation, both on the client- and daemon-side, and as files in the store whose name end in `.drv`—these files are referred to as *derivation*

paths. Derivations paths can be passed to the `build-derivations` procedure to perform the build actions they prescribe (see Section 5.3 [The Store], page 46).

The `(guix derivations)` module provides a representation of derivations as Scheme objects, along with procedures to create and otherwise manipulate derivations. The lowest-level primitive to create a derivation is the `derivation` procedure:

`derivation` *store name builder args* [*#:outputs '("out")*] [Scheme Procedure]
[*#:hash #f*] [*#:hash-algo #f*] [*#:recursive? #f*] [*#:inputs '()*] [*#:env-vars '()*]
[*#:system (%current-system)*] [*#:references-graphs #f*] [*#:allowed-references #f*] [*#:leaked-env-vars #f*] [*#:local-build? #f*] [*#:substitutable? #t*]

Build a derivation with the given arguments, and return the resulting `<derivation>` object.

When *hash* and *hash-algo* are given, a *fixed-output derivation* is created—i.e., one whose result is known in advance, such as a file download. If, in addition, *recursive?* is true, then that fixed output may be an executable file or a directory and *hash* must be the hash of an archive containing this output.

When *references-graphs* is true, it must be a list of file name/store path pairs. In that case, the reference graph of each store path is exported in the build environment in the corresponding file, in a simple text format.

When *allowed-references* is true, it must be a list of store items or outputs that the derivation's output may refer to.

When *leaked-env-vars* is true, it must be a list of strings denoting environment variables that are allowed to "leak" from the daemon's environment to the build environment. This is only applicable to fixed-output derivations—i.e., when *hash* is true. The main use is to allow variables such as `http_proxy` to be passed to derivations that download files.

When *local-build?* is true, declare that the derivation is not a good candidate for offloading and should rather be built locally (see Section 2.4.2 [Daemon Offload Setup], page 6). This is the case for small derivations where the costs of data transfers would outweigh the benefits.

When *substitutable?* is false, declare that substitutes of the derivation's output should not be used (see Section 3.3 [Substitutes], page 20). This is useful, for instance, when building packages that capture details of the host CPU instruction set.

Here's an example with a shell script as its builder, assuming *store* is an open connection to the daemon, and *bash* points to a Bash executable in the store:

```
(use-modules (guix utils)
             (guix store)
             (guix derivations))

(let ((builder   ; add the Bash script to the store
        (add-text-to-store store "my-builder.sh"
                           "echo hello world > $out\n" '())))
  (derivation store "foo"
              bash `("-e" ,builder)
              #:inputs `((,bash) (,builder))
```

```
                #:env-vars '(("HOME" . "/homeless"))))
⇒ #<derivation /gnu/store/...-foo.drv => /gnu/store/...-foo>
```

As can be guessed, this primitive is cumbersome to use directly. A better approach is to write build scripts in Scheme, of course! The best course of action for that is to write the build code as a "G-expression", and to pass it to `gexp->derivation`. For more information, see Section 5.6 [G-Expressions], page 53.

Once upon a time, `gexp->derivation` did not exist and constructing derivations with build code written in Scheme was achieved with `build-expression->derivation`, documented below. This procedure is now deprecated in favor of the much nicer `gexp->derivation`.

`build-expression->derivation` *store name exp* [*#:system* (*%current-system*)] [*#:inputs* '()] [*#:outputs* '("*out*")] [*#:hash #f*] [*#:hash-algo #f*] [*#:recursive? #f*] [*#:env-vars* '()] [*#:modules* '()] [*#:references-graphs #f*] [*#:allowed-references #f*] [*#:local-build? #f*] [*#:substitutable? #t*] [*#:guile-for-build #f*] [Scheme Procedure]

Return a derivation that executes Scheme expression *exp* as a builder for derivation *name*. *inputs* must be a list of `(name drv-path sub-drv)` tuples; when *sub-drv* is omitted, `"out"` is assumed. *modules* is a list of names of Guile modules from the current search path to be copied in the store, compiled, and made available in the load path during the execution of *exp*—e.g., `((guix build utils) (guix build gnu-build-system))`.

exp is evaluated in an environment where `%outputs` is bound to a list of output/path pairs, and where `%build-inputs` is bound to a list of string/output-path pairs made from *inputs*. Optionally, *env-vars* is a list of string pairs specifying the name and value of environment variables visible to the builder. The builder terminates by passing the result of *exp* to `exit`; thus, when *exp* returns `#f`, the build is considered to have failed.

exp is built using *guile-for-build* (a derivation). When *guile-for-build* is omitted or is `#f`, the value of the `%guile-for-build` fluid is used instead.

See the `derivation` procedure for the meaning of *references-graphs*, *allowed-references*, *local-build?*, and *substitutable?*.

Here's an example of a single-output derivation that creates a directory containing one file:

```
(let ((builder '(let ((out (assoc-ref %outputs "out")))
                  (mkdir out)    ; create /gnu/store/...-goo
                  (call-with-output-file (string-append out "/test")
                    (lambda (p)
                      (display '(hello guix) p))))))
  (build-expression->derivation store "goo" builder))

⇒ #<derivation /gnu/store/...-goo.drv => ...>
```

5.5 The Store Monad

The procedures that operate on the store described in the previous sections all take an open connection to the build daemon as their first argument. Although the underlying model is functional, they either have side effects or depend on the current state of the store.

The former is inconvenient: the connection to the build daemon has to be carried around in all those functions, making it impossible to compose functions that do not take that parameter with functions that do. The latter can be problematic: since store operations have side effects and/or depend on external state, they have to be properly sequenced.

This is where the `(guix monads)` module comes in. This module provides a framework for working with *monads*, and a particularly useful monad for our uses, the *store monad*. Monads are a construct that allows two things: associating "context" with values (in our case, the context is the store), and building sequences of computations (here computations include accesses to the store.) Values in a monad—values that carry this additional context—are called *monadic values*; procedures that return such values are called *monadic procedures*.

Consider this "normal" procedure:

```
(define (sh-symlink store)
  ;; Return a derivation that symlinks the 'bash' executable.
  (let* ((drv (package-derivation store bash))
         (out (derivation->output-path drv))
         (sh  (string-append out "/bin/bash")))
    (build-expression->derivation store "sh"
                                  `(symlink ,sh %output))))
```

Using `(guix monads)` and `(guix gexp)`, it may be rewritten as a monadic function:

```
(define (sh-symlink)
  ;; Same, but return a monadic value.
  (mlet %store-monad ((drv (package->derivation bash)))
    (gexp->derivation "sh"
                      #~(symlink (string-append #$drv "/bin/bash")
                                 #$output))))
```

There several things to note in the second version: the `store` parameter is now implicit and is "threaded" in the calls to the `package->derivation` and `gexp->derivation` monadic procedures, and the monadic value returned by `package->derivation` is *bound* using `mlet` instead of plain `let`.

As it turns out, the call to `package->derivation` can even be omitted since it will take place implicitly, as we will see later (see Section 5.6 [G-Expressions], page 53):

```
(define (sh-symlink)
  (gexp->derivation "sh"
                    #~(symlink (string-append #$bash "/bin/bash")
                               #$output)))
```

Calling the monadic `sh-symlink` has no effect. As someone once said, "you exit a monad like you exit a building on fire: by running". So, to exit the monad and get the desired effect, one must use `run-with-store`:

```
(run-with-store (open-connection) (sh-symlink))
```

```
⇒ /gnu/store/...-sh-symlink
```

Note that the `(guix monad-repl)` module extends Guile's REPL with new "meta-commands" to make it easier to deal with monadic procedures: `run-in-store`, and `enter-store-monad`. The former, is used to "run" a single monadic value through the store:

```
scheme@(guile-user)> ,run-in-store (package->derivation hello)
$1 = #<derivation /gnu/store/...-hello-2.9.drv => ...>
```

The latter enters a recursive REPL, where all the return values are automatically run through the store:

```
scheme@(guile-user)> ,enter-store-monad
store-monad@(guile-user) [1]> (package->derivation hello)
$2 = #<derivation /gnu/store/...-hello-2.9.drv => ...>
store-monad@(guile-user) [1]> (text-file "foo" "Hello!")
$3 = "/gnu/store/...-foo"
store-monad@(guile-user) [1]> ,q
scheme@(guile-user)>
```

Note that non-monadic values cannot be returned in the `store-monad` REPL.

The main syntactic forms to deal with monads in general are provided by the `(guix monads)` module and are described below.

`with-monad` *monad body ...* [Scheme Syntax]

Evaluate any `>>=` or `return` forms in *body* as being in *monad*.

`return` *val* [Scheme Syntax]

Return a monadic value that encapsulates *val*.

`>>=` *mval mproc ...* [Scheme Syntax]

Bind monadic value *mval*, passing its "contents" to monadic procedures *mproc*...[2]. There can be one *mproc* or several of them, as in this example:

```
(run-with-state
    (with-monad %state-monad
      (>>= (return 1)
           (lambda (x) (return (+ 1 x)))
           (lambda (x) (return (* 2 x)))))
  'some-state)

⇒ 4
⇒ some-state
```

`mlet` *monad* ((*var mval*) ...) *body ...* [Scheme Syntax]
`mlet*` *monad* ((*var mval*) ...) *body ...* [Scheme Syntax]

Bind the variables *var* to the monadic values *mval* in *body*. The form (*var* `->` *val*) binds *var* to the "normal" value *val*, as per `let`.

`mlet*` is to `mlet` what `let*` is to `let` (see Section "Local Bindings" in *GNU Guile Reference Manual*).

[2] This operation is commonly referred to as "bind", but that name denotes an unrelated procedure in Guile. Thus we use this somewhat cryptic symbol inherited from the Haskell language.

`mbegin` *monad mexp ...* [Scheme System]
Bind *mexp* and the following monadic expressions in sequence, returning the result of the last expression.

This is akin to `mlet`, except that the return values of the monadic expressions are ignored. In that sense, it is analogous to `begin`, but applied to monadic expressions.

The `(guix monads)` module provides the *state monad*, which allows an additional value—the state—to be *threaded* through monadic procedure calls.

`%state-monad` [Scheme Variable]
The state monad. Procedures in the state monad can access and change the state that is threaded.

Consider the example below. The `square` procedure returns a value in the state monad. It returns the square of its argument, but also increments the current state value:

```
(define (square x)
  (mlet %state-monad ((count (current-state)))
    (mbegin %state-monad
      (set-current-state (+ 1 count))
      (return (* x x)))))

(run-with-state (sequence %state-monad (map square (iota 3))) 0)
⇒ (0 1 4)
⇒ 3
```

When "run" through *%state-monad*, we obtain that additional state value, which is the number of `square` calls.

`current-state` [Monadic Procedure]
Return the current state as a monadic value.

`set-current-state` *value* [Monadic Procedure]
Set the current state to *value* and return the previous state as a monadic value.

`state-push` *value* [Monadic Procedure]
Push *value* to the current state, which is assumed to be a list, and return the previous state as a monadic value.

`state-pop` [Monadic Procedure]
Pop a value from the current state and return it as a monadic value. The state is assumed to be a list.

`run-with-state` *mval* [*state*] [Scheme Procedure]
Run monadic value *mval* starting with *state* as the initial state. Return two values: the resulting value, and the resulting state.

The main interface to the store monad, provided by the `(guix store)` module, is as follows.

`%store-monad` [Scheme Variable]

The store monad—an alias for *%state-monad*.

Values in the store monad encapsulate accesses to the store. When its effect is needed, a value of the store monad must be "evaluated" by passing it to the `run-with-store` procedure (see below.)

`run-with-store` *store mval* [*#:guile-for-build*] [*#:system* (*%current-system*)] [Scheme Procedure]

Run *mval*, a monadic value in the store monad, in *store*, an open store connection.

`text-file` *name text* [*references*] [Monadic Procedure]

Return as a monadic value the absolute file name in the store of the file containing *text*, a string. *references* is a list of store items that the resulting text file refers to; it defaults to the empty list.

`interned-file` *file* [*name*] [*#:recursive? #t*] [Monadic Procedure]

Return the name of *file* once interned in the store. Use *name* as its store name, or the basename of *file* if *name* is omitted.

When *recursive?* is true, the contents of *file* are added recursively; if *file* designates a flat file and *recursive?* is true, its contents are added, and its permission bits are kept.

The example below adds a file to the store, under two different names:

```
(run-with-store (open-connection)
  (mlet %store-monad ((a (interned-file "README"))
                      (b (interned-file "README" "LEGU-MIN")))
    (return (list a b))))

⇒ ("/gnu/store/rwm...-README" "/gnu/store/44i...-LEGU-MIN")
```

The `(guix packages)` module exports the following package-related monadic procedures:

`package-file` *package* [*file*] [*#:system* (*%current-system*)] [*#:target #f*] [*#:output* "*out*"] [Monadic Procedure]

Return as a monadic value in the absolute file name of *file* within the *output* directory of *package*. When *file* is omitted, return the name of the *output* directory of *package*. When *target* is true, use it as a cross-compilation target triplet.

`package->derivation` *package* [*system*] [Monadic Procedure]

`package->cross-derivation` *package target* [*system*] [Monadic Procedure]

Monadic version of `package-derivation` and `package-cross-derivation` (see Section 5.1 [Defining Packages], page 37).

5.6 G-Expressions

So we have "derivations", which represent a sequence of build actions to be performed to produce an item in the store (see Section 5.4 [Derivations], page 47). Those build actions are performed when asking the daemon to actually build the derivations; they are run by the daemon in a container (see Section 2.5 [Invoking guix-daemon], page 8).

It should come as no surprise that we like to write those build actions in Scheme. When we do that, we end up with two *strata* of Scheme code[3]: the "host code"—code that defines packages, talks to the daemon, etc.—and the "build code"—code that actually performs build actions, such as making directories, invoking `make`, etc.

To describe a derivation and its build actions, one typically needs to embed build code inside host code. It boils down to manipulating build code as data, and Scheme's homoiconicity—code has a direct representation as data—comes in handy for that. But we need more than Scheme's normal `quasiquote` mechanism to construct build expressions.

The `(guix gexp)` module implements *G-expressions*, a form of S-expressions adapted to build expressions. G-expressions, or *gexps*, consist essentially in three syntactic forms: `gexp`, `ungexp`, and `ungexp-splicing` (or simply: `#~`, `#$`, and `#$@`), which are comparable respectively to `quasiquote`, `unquote`, and `unquote-splicing` (see Section "Expression Syntax" in *GNU Guile Reference Manual*). However, there are major differences:

- Gexps are meant to be written to a file and run or manipulated by other processes.
- When a high-level object such as a package or derivation is unquoted inside a gexp, the result is as if its output file name had been introduced.
- Gexps carry information about the packages or derivations they refer to, and these dependencies are automatically added as inputs to the build processes that use them.

This mechanism is not limited to package and derivation objects: *compilers* able to "lower" other high-level objects to derivations or files in the store can be defined, such that these objects can also be inserted into gexps. For example, a useful type of high-level object that can be inserted in a gexp is "file-like objects", which make it easy to add files to the store and refer to them in derivations and such (see `local-file` and `plain-file` below.)

To illustrate the idea, here is an example of a gexp:

```
(define build-exp
  #~(begin
      (mkdir #$output)
      (chdir #$output)
      (symlink (string-append #$coreutils "/bin/ls")
               "list-files")))
```

This gexp can be passed to `gexp->derivation`; we obtain a derivation that builds a directory containing exactly one symlink to `/gnu/store/...-coreutils-8.22/bin/ls`:

```
(gexp->derivation "the-thing" build-exp)
```

As one would expect, the `"/gnu/store/...-coreutils-8.22"` string is substituted to the reference to the *coreutils* package in the actual build code, and *coreutils* is automatically made an input to the derivation. Likewise, `#$output` (equivalent to `(ungexp output)`) is replaced by a string containing the derivation's output directory name.

In a cross-compilation context, it is useful to distinguish between references to the *native* build of a package—that can run on the host—versus references to cross builds of a package. To that end, the `#+` plays the same role as `#$`, but is a reference to a native package build:

[3] The term *stratum* in this context was coined by Manuel Serrano et al. in the context of their work on Hop. Oleg Kiselyov, who has written insightful essays and code on this topic, refers to this kind of code generation as *staging*.

```
(gexp->derivation "vi"
   #~(begin
       (mkdir #$output)
       (system* (string-append #+coreutils "/bin/ln")
                "-s"
                (string-append #$emacs "/bin/emacs")
                (string-append #$output "/bin/vi")))
   #:target "mips64el-linux")
```

In the example above, the native build of *coreutils* is used, so that `ln` can actually run on the host; but then the cross-compiled build of *emacs* is referenced.

The syntactic form to construct gexps is summarized below.

`#~`*exp* [Scheme Syntax]
`(gexp` *exp*`)` [Scheme Syntax]

Return a G-expression containing *exp*. *exp* may contain one or more of the following forms:

`#$`*obj*
`(ungexp` *obj*`)`

Introduce a reference to *obj*. *obj* may have one of the supported types, for example a package or a derivation, in which case the `ungexp` form is replaced by its output file name—e.g., `"/gnu/store/...-coreutils-8.22`.

If *obj* is a list, it is traversed and references to supported objects are substituted similarly.

If *obj* is another gexp, its contents are inserted and its dependencies are added to those of the containing gexp.

If *obj* is another kind of object, it is inserted as is.

`#$`*obj*`:`*output*
`(ungexp` *obj output*`)`

This is like the form above, but referring explicitly to the *output* of *obj*—this is useful when *obj* produces multiple outputs (see Section 3.4 [Packages with Multiple Outputs], page 21).

`#+`*obj*
`#+`*obj*`:output`
`(ungexp-native` *obj*`)`
`(ungexp-native` *obj output*`)`

Same as `ungexp`, but produces a reference to the *native* build of *obj* when used in a cross compilation context.

`#$output[:`*output*`]`
`(ungexp output [`*output*`])`

Insert a reference to derivation output *output*, or to the main output when *output* is omitted.

This only makes sense for gexps passed to `gexp->derivation`.

`#$@`*lst*
`(ungexp-splicing` *lst*`)`
: Like the above, but splices the contents of *lst* inside the containing list.

`#+@`*lst*
`(ungexp-native-splicing` *lst*`)`
: Like the above, but refers to native builds of the objects listed in *lst*.

G-expressions created by `gexp` or `#~` are run-time objects of the `gexp?` type (see below.)

`gexp?` *obj* [Scheme Procedure]
: Return `#t` if *obj* is a G-expression.

G-expressions are meant to be written to disk, either as code building some derivation, or as plain files in the store. The monadic procedures below allow you to do that (see Section 5.5 [The Store Monad], page 50, for more information about monads.)

`gexp->derivation` *name exp* [*#:system* (*%current-system*)] [Monadic Procedure]
[*#:target #f*] [*#:graft? #t*] [*#:hash #f*] [*#:hash-algo #f*] [*#:recursive? #f*]
[*#:env-vars '()*] [*#:modules '()*] [*#:module-path* `%load-path`]
[*#:references-graphs #f*] [*#:allowed-references #f*] [*#:leaked-env-vars #f*]
[*#:script-name* (*string-append* `name` *"-builder"*)] [*#:local-build? #f*]
[*#:substitutable? #t*] [*#:guile-for-build #f*]
: Return a derivation *name* that runs *exp* (a gexp) with *guile-for-build* (a derivation) on *system*; *exp* is stored in a file called *script-name*. When *target* is true, it is used as the cross-compilation target triplet for packages referred to by *exp*.

Make *modules* available in the evaluation context of *exp*; *modules* is a list of names of Guile modules searched in *module-path* to be copied in the store, compiled, and made available in the load path during the execution of *exp*—e.g., `((guix build utils) (guix build gnu-build-system))`.

graft? determines whether packages referred to by *exp* should be grafted when applicable.

When *references-graphs* is true, it must be a list of tuples of one of the following forms:

```
(file-name package)
(file-name package output)
(file-name derivation)
(file-name derivation output)
(file-name store-item)
```

The right-hand-side of each element of *references-graphs* is automatically made an input of the build process of *exp*. In the build environment, each *file-name* contains the reference graph of the corresponding item, in a simple text format.

allowed-references must be either `#f` or a list of output names and packages. In the latter case, the list denotes store items that the result is allowed to refer to. Any reference to another store item will lead to a build error.

The other arguments are as for `derivation` (see Section 5.4 [Derivations], page 47).

The `local-file`, `plain-file`, `computed-file`, `program-file`, and `scheme-file` procedures below return *file-like objects*. That is, when unquoted in a G-expression, these objects lead to a file in the store. Consider this G-expression:

```
#~(system* (string-append #$glibc "/sbin/nscd") "-f"
           #$(local-file "/tmp/my-nscd.conf"))
```

The effect here is to "intern" `/tmp/my-nscd.conf` by copying it to the store. Once expanded, for instance *via* `gexp->derivation`, the G-expression refers to that copy under `/gnu/store`; thus, modifying or removing the file in `/tmp` does not have any effect on what the G-expression does. `plain-file` can be used similarly; it differs in that the file content is directly passed as a string.

`local-file` *file* [*name*] [*#:recursive? #t*] [Scheme Procedure]

Return an object representing local file *file* to add to the store; this object can be used in a gexp. *file* will be added to the store under *name*–by default the base name of *file*.

When *recursive?* is true, the contents of *file* are added recursively; if *file* designates a flat file and *recursive?* is true, its contents are added, and its permission bits are kept.

This is the declarative counterpart of the `interned-file` monadic procedure (see Section 5.5 [The Store Monad], page 50).

`plain-file` *name content* [Scheme Procedure]

Return an object representing a text file called *name* with the given *content* (a string) to be added to the store.

This is the declarative counterpart of `text-file`.

`computed-file` *name gexp* [*#:modules '()*] [*#:options '(#:local-build? #t)*] [Scheme Procedure]

Return an object representing the store item *name*, a file or directory computed by *gexp*. *modules* specifies the set of modules visible in the execution context of *gexp*. *options* is a list of additional arguments to pass to `gexp->derivation`.

This is the declarative counterpart of `gexp->derivation`.

`gexp->script` *name exp* [Monadic Procedure]

Return an executable script *name* that runs *exp* using *guile* with *modules* in its search path.

The example below builds a script that simply invokes the `ls` command:

```
(use-modules (guix gexp) (gnu packages base))

(gexp->script "list-files"
              #~(execl (string-append #$coreutils "/bin/ls")
                       "ls"))
```

When "running" it through the store (see Section 5.5 [The Store Monad], page 50), we obtain a derivation that produces an executable file `/gnu/store/...-list-files` along these lines:

```
#!/gnu/store/...-guile-2.0.11/bin/guile -ds
!#
(execl (string-append "/gnu/store/...-coreutils-8.22"/bin/ls")
       "ls")
```

`program-file` *name exp* [*#:modules '()*] [*#:guile #f*] [Scheme Procedure]
Return an object representing the executable store item *name* that runs *gexp*. *guile* is the Guile package used to execute that script, and *modules* is the list of modules visible to that script.

This is the declarative counterpart of `gexp->script`.

`gexp->file` *name exp* [Monadic Procedure]
Return a derivation that builds a file *name* containing *exp*.

The resulting file holds references to all the dependencies of *exp* or a subset thereof.

`scheme-file` *name exp* [Scheme Procedure]
Return an object representing the Scheme file *name* that contains *exp*.

This is the declarative counterpart of `gexp->file`.

`text-file*` *name text ...* [Monadic Procedure]
Return as a monadic value a derivation that builds a text file containing all of *text*. *text* may list, in addition to strings, objects of any type that can be used in a gexp: packages, derivations, local file objects, etc. The resulting store file holds references to all these.

This variant should be preferred over `text-file` anytime the file to create will reference items from the store. This is typically the case when building a configuration file that embeds store file names, like this:

```
(define (profile.sh)
  ;; Return the name of a shell script in the store that
  ;; initializes the 'PATH' environment variable.
  (text-file* "profile.sh"
              "export PATH=" coreutils "/bin:"
              grep "/bin:" sed "/bin\n"))
```

In this example, the resulting `/gnu/store/...-profile.sh` file will references *coreutils*, *grep*, and *sed*, thereby preventing them from being garbage-collected during its lifetime.

`mixed-text-file` *name text ...* [Scheme Procedure]
Return an object representing store file *name* containing *text*. *text* is a sequence of strings and file-like objects, as in:

```
(mixed-text-file "profile"
                 "export PATH=" coreutils "/bin:" grep "/bin")
```

This is the declarative counterpart of `text-file*`.

Of course, in addition to gexps embedded in "host" code, there are also modules containing build tools. To make it clear that they are meant to be used in the build stratum, these modules are kept in the `(guix build ...)` name space.

Internally, high-level objects are *lowered*, using their compiler, to either derivations or store items. For instance, lowering a package yields a derivation, and lowering a `plain-file` yields a store item. This is achieved using the `lower-object` monadic procedure.

`lower-object` *obj* [*system*] [*#:target #f*] [Monadic Procedure]
Return as a value in *%store-monad* the derivation or store item corresponding to *obj* for *system*, cross-compiling for *target* if *target* is true. *obj* must be an object that has an associated gexp compiler, such as a `<package>`.

6 Utilities

This section describes tools primarily targeted at developers and users who write new package definitions. They complement the Scheme programming interface of Guix in a convenient way.

6.1 Invoking `guix build`

The `guix build` command builds packages or derivations and their dependencies, and prints the resulting store paths. Note that it does not modify the user's profile—this is the job of the `guix package` command (see Section 3.2 [Invoking guix package], page 14). Thus, it is mainly useful for distribution developers.

The general syntax is:

```
guix build options package-or-derivation...
```

package-or-derivation may be either the name of a package found in the software distribution such as `coreutils` or `coreutils-8.20`, or a derivation such as `/gnu/store/...-coreutils-8.19.drv`. In the former case, a package with the corresponding name (and optionally version) is searched for among the GNU distribution modules (see Section 7.5 [Package Modules], page 126).

Alternatively, the `--expression` option may be used to specify a Scheme expression that evaluates to a package; this is useful when disambiguation among several same-named packages or package variants is needed.

The *options* may be zero or more of the following:

`--file=`*file*
`-f` *file*

Build the package or derivation that the code within *file* evaluates to.

As an example, *file* might contain a package definition like this (see Section 5.1 [Defining Packages], page 37):

```
(use-modules (guix)
             (guix build-system gnu)
             (guix licenses))

(package
  (name "hello")
  (version "2.10")
  (source (origin
            (method url-fetch)
            (uri (string-append "mirror://gnu/hello/hello-" version
                                ".tar.gz"))
            (sha256
             (base32
              "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
  (build-system gnu-build-system)
  (synopsis "Hello, GNU world:  An example GNU package")
  (description "Guess what GNU Hello prints!")
```

```
                (home-page "http://www.gnu.org/software/hello/")
                (license gpl3+))
```

`--expression=`*expr*

`-e` *expr* Build the package or derivation *expr* evaluates to.

For example, *expr* may be `(@ (gnu packages guile) guile-1.8)`, which unambiguously designates this specific variant of version 1.8 of Guile.

Alternately, *expr* may be a G-expression, in which case it is used as a build program passed to `gexp->derivation` (see Section 5.6 [G-Expressions], page 53).

Lastly, *expr* may refer to a zero-argument monadic procedure (see Section 5.5 [The Store Monad], page 50). The procedure must return a derivation as a monadic value, which is then passed through `run-with-store`.

`--source`

`-S` Build the packages' source derivations, rather than the packages themselves.

For instance, `guix build -S gcc` returns something like `/gnu/store/...-gcc-4.7.2.tar.bz2`, which is GCC's source tarball.

The returned source tarball is the result of applying any patches and code snippets specified in the package's `origin` (see Section 5.1 [Defining Packages], page 37).

`--sources`

Fetch and return the source of *package-or-derivation* and all their dependencies, recursively. This is a handy way to obtain a local copy of all the source code needed to build *packages*, allowing you to eventually build them even without network access. It is an extension of the `--source` option and can accept one of the following optional argument values:

`package` This value causes the `--sources` option to behave in the same way as the `--source` option.

`all` Build all packages' source derivations, including any source that might be listed as `inputs`. This is the default value.

```
$ guix build --sources tzdata
The following derivations will be built:
   /gnu/store/...-tzdata2015b.tar.gz.drv
   /gnu/store/...-tzcode2015b.tar.gz.drv
```

`transitive`
Build all packages' source derivations, as well as all source derivations for packages' transitive inputs. This can be used e.g. to prefetch package source for later offline building.

```
$ guix build --sources=transitive tzdata
The following derivations will be built:
   /gnu/store/...-tzcode2015b.tar.gz.drv
   /gnu/store/...-findutils-4.4.2.tar.xz.drv
   /gnu/store/...-grep-2.21.tar.xz.drv
   /gnu/store/...-coreutils-8.23.tar.xz.drv
   /gnu/store/...-make-4.1.tar.xz.drv
```

```
/gnu/store/...-bash-4.3.tar.xz.drv
...
```

`--system=system`
`-s system` Attempt to build for *system*—e.g., `i686-linux`—instead of the host's system type.

An example use of this is on Linux-based systems, which can emulate different personalities. For instance, passing `--system=i686-linux` on an `x86_64-linux` system allows users to build packages in a complete 32-bit environment.

`--target=triplet`
Cross-build for *triplet*, which must be a valid GNU triplet, such as `"mips64el-linux-gnu"` (see Section "Configuration Names" in *GNU Configure and Build System*).

`--with-source=source`
Use *source* as the source of the corresponding package. *source* must be a file name or a URL, as for `guix download` (see Section 6.3 [Invoking guix download], page 65).

The "corresponding package" is taken to be one specified on the command line whose name matches the base of *source*—e.g., if *source* is `/src/guile-2.0.10.tar.gz`, the corresponding package is `guile`. Likewise, the version string is inferred from *source*; in the previous example, it's `2.0.10`.

This option allows users to try out versions of packages other than the one provided by the distribution. The example below downloads `ed-1.7.tar.gz` from a GNU mirror and uses that as the source for the `ed` package:

```
guix build ed --with-source=mirror://gnu/ed/ed-1.7.tar.gz
```

As a developer, `--with-source` makes it easy to test release candidates:

```
guix build guile --with-source=../guile-2.0.9.219-e1bb7.tar.xz
```

... or to build from a checkout in a pristine environment:

```
$ git clone git://git.sv.gnu.org/guix.git
$ guix build guix --with-source=./guix
```

`--no-grafts`
Do not "graft" packages. In practice, this means that package updates available as grafts are not applied. See Section 7.4 [Security Updates], page 125, for more information on grafts.

`--derivations`
`-d` Return the derivation paths, not the output paths, of the given packages.

`--root=file`
`-r file` Make *file* a symlink to the result, and register it as a garbage collector root.

`--log-file`
Return the build log file names or URLs for the given *package-or-derivation*s, or raise an error if build logs are missing.

This works regardless of how packages or derivations are specified. For instance, the following invocations are equivalent:

```
guix build --log-file `guix build -d guile`
guix build --log-file `guix build guile`
guix build --log-file guile
guix build --log-file -e '(@ (gnu packages guile) guile-2.0)'
```

If a log is unavailable locally, and unless `--no-substitutes` is passed, the command looks for a corresponding log on one of the substitute servers (as specified with `--substitute-urls`.)

So for instance, let's say you want to see the build log of GDB on MIPS but you're actually on an `x86_64` machine:

```
$ guix build --log-file gdb -s mips64el-linux
http://hydra.gnu.org/log/...-gdb-7.10
```

You can freely access a huge library of build logs!

In addition, a number of options that control the build process are common to `guix build` and other commands that can spawn builds, such as `guix package` or `guix archive`. These are the following:

`--load-path=`*directory*
`-L` *directory*
: Add *directory* to the front of the package module search path (see Section 7.5 [Package Modules], page 126).

  This allows users to define their own packages and make them visible to the command-line tools.

`--keep-failed`
`-K`
: Keep the build tree of failed builds. Thus, if a build fail, its build tree is kept under `/tmp`, in a directory whose name is shown at the end of the build log. This is useful when debugging build issues.

`--dry-run`
`-n`
: Do not build the derivations.

`--fallback`
: When substituting a pre-built binary fails, fall back to building packages locally.

`--substitute-urls=`*urls*
: Consider *urls* the whitespace-separated list of substitute source URLs, overriding the default list of URLs of `guix-daemon` (see [guix-daemon URLs], page 9).

  This means that substitutes may be downloaded from *urls*, provided they are signed by a key authorized by the system administrator (see Section 3.3 [Substitutes], page 20).

`--no-substitutes`
: Do not use substitutes for build products. That is, always build things locally instead of allowing downloads of pre-built binaries (see Section 3.3 [Substitutes], page 20).

`--no-build-hook`
: Do not attempt to offload builds *via* the daemon's "build hook" (see Section 2.4.2 [Daemon Offload Setup], page 6). That is, always build things locally instead of offloading builds to remote machines.

`--max-silent-time=`*seconds*
: When the build or substitution process remains silent for more than *seconds*, terminate it and report a build failure.

`--timeout=`*seconds*
: Likewise, when the build or substitution process lasts for more than *seconds*, terminate it and report a build failure.

  By default there is no timeout. This behavior can be restored with `--timeout=0`.

`--verbosity=`*level*
: Use the given verbosity level. *level* must be an integer between 0 and 5; higher means more verbose output. Setting a level of 4 or more may be helpful when debugging setup issues with the build daemon.

`--cores=`*n*
`-c` *n*
: Allow the use of up to *n* CPU cores for the build. The special value `0` means to use as many CPU cores as available.

`--max-jobs=`*n*
`-M` *n*
: Allow at most *n* build jobs in parallel. See Section 2.5 [Invoking guix-daemon], page 8, for details about this option and the equivalent `guix-daemon` option.

Behind the scenes, `guix build` is essentially an interface to the `package-derivation` procedure of the `(guix packages)` module, and to the `build-derivations` procedure of the `(guix derivations)` module.

In addition to options explicitly passed on the command line, `guix build` and other `guix` commands that support building honor the `GUIX_BUILD_OPTIONS` environment variable.

`GUIX_BUILD_OPTIONS` [Environment Variable]
: Users can define this variable to a list of command line options that will automatically be used by `guix build` and other `guix` commands that can perform builds, as in the example below:

```
$ export GUIX_BUILD_OPTIONS="--no-substitutes -c 2 -L /foo/bar"
```

  These options are parsed independently, and the result is appended to the parsed command-line options.

6.2 Invoking `guix edit`

So many packages, so many source files! The `guix edit` command facilitates the life of packagers by pointing their editor at the source file containing the definition of the specified packages. For instance:

```
guix edit gcc-4.8 vim
```

launches the program specified in the `EDITOR` environment variable to edit the recipe of GCC 4.8.4 and that of Vim.

If you are using Emacs, note that the Emacs user interface provides similar functionality in the "package info" and "package list" buffers created by *M-x guix-search-by-name* and similar commands (see Section 4.2.1 [Emacs Commands], page 28).

6.3 Invoking `guix download`

When writing a package definition, developers typically need to download the package's source tarball, compute its SHA256 hash, and write that hash in the package definition (see Section 5.1 [Defining Packages], page 37). The `guix download` tool helps with this task: it downloads a file from the given URI, adds it to the store, and prints both its file name in the store and its SHA256 hash.

The fact that the downloaded file is added to the store saves bandwidth: when the developer eventually tries to build the newly defined package with `guix build`, the source tarball will not have to be downloaded again because it is already in the store. It is also a convenient way to temporarily stash files, which may be deleted eventually (see Section 3.5 [Invoking guix gc], page 22).

The `guix download` command supports the same URIs as used in package definitions. In particular, it supports `mirror://` URIs. `https` URIs (HTTP over TLS) are supported *provided* the Guile bindings for GnuTLS are available in the user's environment; when they are not available, an error is raised. See Section "Guile Preparations" in *GnuTLS-Guile*, for more information.

The following option is available:

`--format=`*fmt*
`-f` *fmt* Write the hash in the format specified by *fmt*. For more information on the valid values for *fmt*, see Section 6.4 [Invoking guix hash], page 65.

6.4 Invoking `guix hash`

The `guix hash` command computes the SHA256 hash of a file. It is primarily a convenience tool for anyone contributing to the distribution: it computes the cryptographic hash of a file, which can be used in the definition of a package (see Section 5.1 [Defining Packages], page 37).

The general syntax is:

```
guix hash option file
```

`guix hash` has the following option:

`--format=`*fmt*
`-f` *fmt* Write the hash in the format specified by *fmt*.

Supported formats: `nix-base32`, `base32`, `base16` (`hex` and `hexadecimal` can be used as well).

If the `--format` option is not specified, `guix hash` will output the hash in `nix-base32`. This representation is used in the definitions of packages.

`--recursive`
`-r` Compute the hash on *file* recursively.

In this case, the hash is computed on an archive containing *file*, including its children if it is a directory. Some of *file*'s meta-data is part of the archive; for instance, when *file* is a regular file, the hash is different depending on whether *file* is executable or not. Meta-data such as time stamps has no impact on the hash (see Section 3.7 [Invoking guix archive], page 24).

6.5 Invoking `guix import`

The `guix import` command is useful for people willing to add a package to the distribution but who'd rather do as little work as possible to get there—a legitimate demand. The command knows of a few repositories from which it can "import" package meta-data. The result is a package definition, or a template thereof, in the format we know (see Section 5.1 [Defining Packages], page 37).

The general syntax is:

```
guix import importer options...
```

importer specifies the source from which to import package meta-data, and *options* specifies a package identifier and other options specific to *importer*. Currently, the available "importers" are:

`gnu` Import meta-data for the given GNU package. This provides a template for the latest version of that GNU package, including the hash of its source tarball, and its canonical synopsis and description.

Additional information such as the package's dependencies and its license needs to be figured out manually.

For example, the following command returns a package definition for GNU Hello:

```
guix import gnu hello
```

Specific command-line options are:

`--key-download=policy`
As for `guix refresh`, specify the policy to handle missing OpenPGP keys when verifying the package's signature. See Section 6.6 [Invoking guix refresh], page 68.

`pypi` Import meta-data from the Python Package Index[1]. Information is taken from the JSON-formatted description available at `pypi.python.org` and usually includes all the relevant information, including package dependencies.

The command below imports meta-data for the `itsdangerous` Python package:

```
guix import pypi itsdangerous
```

`gem` Import meta-data from RubyGems[2]. Information is taken from the JSON-formatted description available at `rubygems.org` and includes most relevant information, including runtime dependencies. There are some caveats, however. The meta-data doesn't distinguish between synopses and descriptions, so the same string is used for both fields. Additionally, the details of non-Ruby dependencies required to build native extensions is unavailable and left as an exercise to the packager.

The command below imports meta-data for the `rails` Ruby package:

```
guix import gem rails
```

[1] This functionality requires Guile-JSON to be installed. See Section 2.2 [Requirements], page 4.

[2] This functionality requires Guile-JSON to be installed. See Section 2.2 [Requirements], page 4.

`cpan` Import meta-data from MetaCPAN. Information is taken from the JSON-formatted meta-data provided through MetaCPAN's API and includes most relevant information, such as module dependencies. License information should be checked closely. If Perl is available in the store, then the `corelist` utility will be used to filter core modules out of the list of dependencies.

The command command below imports meta-data for the `Acme::Boolean` Perl module:

```
guix import cpan Acme::Boolean
```

`cran` Import meta-data from CRAN, the central repository for the GNU R statistical and graphical environment.

Information is extracted from the HTML package description.

The command command below imports meta-data for the `Cairo` R package:

```
guix import cran Cairo
```

`nix` Import meta-data from a local copy of the source of the Nixpkgs distribution[3]. Package definitions in Nixpkgs are typically written in a mixture of Nix-language and Bash code. This command only imports the high-level package structure that is written in the Nix language. It normally includes all the basic fields of a package definition.

When importing a GNU package, the synopsis and descriptions are replaced by their canonical upstream variant.

As an example, the command below imports the package definition of LibreOffice (more precisely, it imports the definition of the package bound to the `libreoffice` top-level attribute):

```
guix import nix ~/path/to/nixpkgs libreoffice
```

`hackage` Import meta-data from Haskell community's central package archive Hackage. Information is taken from Cabal files and includes all the relevant information, including package dependencies.

Specific command-line options are:

`--stdin`
`-s` Read a Cabal file from the standard input.

`--no-test-dependencies`
`-t` Do not include dependencies required by the test suites only.

`--cabal-environment=`*alist*
`-e` *alist* *alist* is a Scheme alist defining the environment in which the Cabal conditionals are evaluated. The accepted keys are: `os`, `arch`, `impl` and a string representing the name of a flag. The value associated with a flag has to be either the symbol `true` or `false`. The value associated with other keys has to conform to the Cabal file format definition. The default value associated with the keys `os`, `arch` and `impl` is '`linux`', '`x86_64`' and '`ghc`' respectively.

[3] This relies on the `nix-instantiate` command of Nix.

The command below imports meta-data for the latest version of the `HTTP` Haskell package without including test dependencies and specifying the value of the flag '`network-uri`' as `false`:

```
guix import hackage -t -e "'((\"network-uri\" . false))" HTTP
```

A specific package version may optionally be specified by following the package name by a hyphen and a version number as in the following example:

```
guix import hackage mtl-2.1.3.1
```

`elpa` Import meta-data from an Emacs Lisp Package Archive (ELPA) package repository (see Section "Packages" in *The GNU Emacs Manual*).

Specific command-line options are:

`--archive=`*repo*
`-a` *repo* *repo* identifies the archive repository from which to retrieve the information. Currently the supported repositories and their identifiers are:

- GNU, selected by the `gnu` identifier. This is the default.
- MELPA-Stable, selected by the `melpa-stable` identifier.
- MELPA, selected by the `melpa` identifier.

The structure of the `guix import` code is modular. It would be useful to have more importers for other package formats, and your help is welcome here (see Chapter 8 [Contributing], page 135).

6.6 Invoking `guix refresh`

The primary audience of the `guix refresh` command is developers of the GNU software distribution. By default, it reports any packages provided by the distribution that are outdated compared to the latest upstream version, like this:

```
$ guix refresh
gnu/packages/gettext.scm:29:13: gettext would be upgraded from 0.18.1.1 to 0.18.2.1
gnu/packages/glib.scm:77:12: glib would be upgraded from 2.34.3 to 2.37.0
```

It does so by browsing each package's FTP directory and determining the highest version number of the source tarballs therein. The command knows how to update specific types of packages: GNU packages, ELPA packages, etc.—see the documentation for `--type` below. The are many packages, though, for which it lacks a method to determine whether a new upstream release is available. However, the mechanism is extensible, so feel free to get in touch with us to add a new method!

When passed `--update`, it modifies distribution source files to update the version numbers and source tarball hashes of those packages' recipes (see Section 5.1 [Defining Packages], page 37). This is achieved by downloading each package's latest source tarball and its associated OpenPGP signature, authenticating the downloaded tarball against its signature using `gpg`, and finally computing its hash. When the public key used to sign the tarball is missing from the user's keyring, an attempt is made to automatically retrieve it from a public key server; when it's successful, the key is added to the user's keyring; otherwise, `guix refresh` reports an error.

The following options are supported:

`--update`
`-u` Update distribution source files (package recipes) in place. This is usually run from a checkout of the Guix source tree (see Section 8.2 [Running Guix Before It Is Installed], page 135):

```
$ ./pre-inst-env guix refresh -s non-core
```

See Section 5.1 [Defining Packages], page 37, for more information on package definitions.

`--select=[`*subset*`]`
`-s` *subset* Select all the packages in *subset*, one of `core` or `non-core`.

The `core` subset refers to all the packages at the core of the distribution—i.e., packages that are used to build "everything else". This includes GCC, libc, Binutils, Bash, etc. Usually, changing one of these packages in the distribution entails a rebuild of all the others. Thus, such updates are an inconvenience to users in terms of build time or bandwidth used to achieve the upgrade.

The `non-core` subset refers to the remaining packages. It is typically useful in cases where an update of the core packages would be inconvenient.

`--type=`*updater*
`-t` *updater*

Select only packages handled by *updater* (may be a comma-separated list of updaters). Currently, *updater* may be one of:

`gnu` the updater for GNU packages;

`elpa` the updater for ELPA packages;

`cran` the updater for CRAN packages;

`pypi` the updater for PyPI packages.

For instance, the following commands only checks for updates of Emacs packages hosted at `elpa.gnu.org` and updates of CRAN packages:

```
$ guix refresh --type=elpa,cran
gnu/packages/statistics.scm:819:13: r-testthat would be upgraded from 0.10.0
gnu/packages/emacs.scm:856:13: emacs-auctex would be upgraded from 11.88.6 t
```

In addition, `guix refresh` can be passed one or more package names, as in this example:

```
$ ./pre-inst-env guix refresh -u emacs idutils gcc-4.8.4
```

The command above specifically updates the `emacs` and `idutils` packages. The `--select` option would have no effect in this case.

When considering whether to upgrade a package, it is sometimes convenient to know which packages would be affected by the upgrade and should be checked for compatibility. For this the following option may be used when passing `guix refresh` one or more package names:

`--list-updaters`
`-L` List available updaters and exit (see `--type` above.)

`--list-dependent`
`-l` List top-level dependent packages that would need to be rebuilt as a result of upgrading one or more packages.

Be aware that the `--list-dependent` option only *approximates* the rebuilds that would be required as a result of an upgrade. More rebuilds might be required under some circumstances.

```
$ guix refresh --list-dependent flex
Building the following 120 packages would ensure 213 dependent packages are rebuilt:
hop-2.4.0 geiser-0.4 notmuch-0.18 mu-0.9.9.5 cflow-1.4 idutils-4.6 ...
```

The command above lists a set of packages that could be built to check for compatibility with an upgraded `flex` package.

The following options can be used to customize GnuPG operation:

`--gpg=`*command*
: Use *command* as the GnuPG 2.x command. *command* is searched for in `$PATH`.

`--key-download=`*policy*
: Handle missing OpenPGP keys according to *policy*, which may be one of:

  `always`
  : Always download missing OpenPGP keys from the key server, and add them to the user's GnuPG keyring.

  `never`
  : Never try to download missing OpenPGP keys. Instead just bail out.

  `interactive`
  : When a package signed with an unknown OpenPGP key is encountered, ask the user whether to download it or not. This is the default behavior.

`--key-server=`*host*
: Use *host* as the OpenPGP key server when importing a public key.

6.7 Invoking `guix lint`

The `guix lint` is meant to help package developers avoid common errors and use a consistent style. It runs a number of checks on a given set of packages in order to find common mistakes in their definitions. Available *checkers* include (see `--list-checkers` for a complete list):

`synopsis`
`description`
: Validate certain typographical and stylistic rules about package descriptions and synopses.

`inputs-should-be-native`
: Identify inputs that should most likely be native inputs.

`source`
`home-page`
`source-file-name`
: Probe `home-page` and `source` URLs and report those that are invalid. Check that the source file name is meaningful, e.g. is not just a version number or "git-checkout", and should not have a `file-name` declared (see Section 5.1.2 [origin Reference], page 41).

`formatting`
: Warn about obvious source code formatting issues: trailing white space, use of tabulations, etc.

The general syntax is:

```
guix lint options package...
```

If no package is given on the command line, then all packages are checked. The *options* may be zero or more of the following:

`--checkers`
`-c`
: Only enable the checkers specified in a comma-separated list using the names returned by `--list-checkers`.

`--list-checkers`
`-l`
: List and describe all the available checkers that will be run on packages and exit.

6.8 Invoking `guix size`

The `guix size` command helps package developers profile the disk usage of packages. It is easy to overlook the impact of an additional dependency added to a package, or the impact of using a single output for a package that could easily be split (see Section 3.4 [Packages with Multiple Outputs], page 21). These are the typical issues that `guix size` can highlight.

The command can be passed a package specification such as `gcc-4.8` or `guile:debug`, or a file name in the store. Consider this example:

```
$ guix size coreutils
store item                                 total    self
/gnu/store/...-coreutils-8.23             70.0    13.9  19.8%
/gnu/store/...-gmp-6.0.0a                 55.3     2.5   3.6%
/gnu/store/...-acl-2.2.52                 53.7     0.5   0.7%
/gnu/store/...-attr-2.4.46                53.2     0.3   0.5%
/gnu/store/...-gcc-4.8.4-lib              52.9    15.7  22.4%
/gnu/store/...-glibc-2.21                 37.2    37.2  53.1%
```

The store items listed here constitute the *transitive closure* of Coreutils—i.e., Coreutils and all its dependencies, recursively—as would be returned by:

```
$ guix gc -R /gnu/store/...-coreutils-8.23
```

Here the output shows 3 columns next to store items. The first column, labeled "total", shows the size in mebibytes (MiB) of the closure of the store item—that is, its own size plus the size of all its dependencies. The next column, labeled "self", shows the size of the item itself. The last column shows the ratio of the item's size to the space occupied by all the items listed here.

In this example, we see that the closure of Coreutils weighs in at 70 MiB, half of which is taken by libc. (That libc represents a large fraction of the closure is not a problem *per se* because it is always available on the system anyway.)

When the package passed to `guix size` is available in the store, `guix size` queries the daemon to determine its dependencies, and measures its size in the store, similar to `du -ms --apparent-size` (see Section "du invocation" in *GNU Coreutils*).

When the given package is *not* in the store, `guix size` reports information based on information about the available substitutes (see Section 3.3 [Substitutes], page 20). This allows it to profile disk usage of store items that are not even on disk, only available remotely.

The available options are:

`--substitute-urls=`*urls*
: Use substitute information from *urls*. See [client-substitute-urls], page 63.

`--map-file=`*file*
: Write to *file* a graphical map of disk usage as a PNG file.

  For the example above, the map looks like this:

  This option requires that Guile-Charting be installed and visible in Guile's module search path. When that is not the case, `guix size` fails as it tries to load it.

`--system=`*system*
`-s` *system* Consider packages for *system*—e.g., `x86_64-linux`.

6.9 Invoking `guix graph`

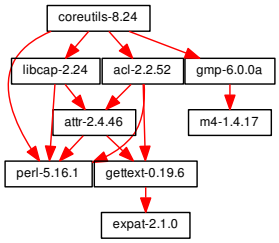
Packages and their dependencies form a *graph*, specifically a directed acyclic graph (DAG). It can quickly become difficult to have a mental model of the package DAG, so the `guix graph` command is here to provide a visual representation of the DAG. `guix graph` emits a DAG representation in the input format of Graphviz, so its output can be passed directly to Graphviz's `dot` command, for instance. The general syntax is:

```
guix graph options package...
```

For example, the following command generates a PDF file representing the package DAG for the GNU Core Utilities, showing its build-time dependencies:

```
guix graph coreutils | dot -Tpdf > dag.pdf
```

The output looks like this:



Nice little graph, no?

But there's more than one graph! The one above is concise: it's the graph of package objects, omitting implicit inputs such as GCC, libc, grep, etc. It's often useful to have such a concise graph, but sometimes you want to see more details. `guix graph` supports several types of graphs, allowing you to choose the level of details:

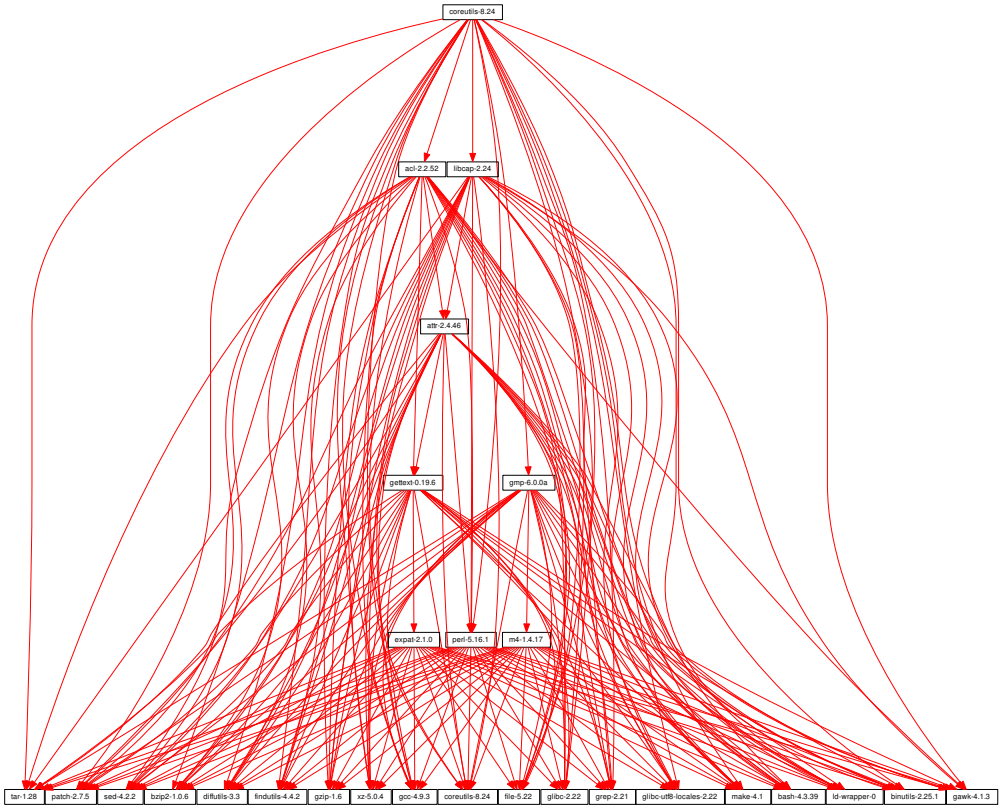
`package` This is the default type, the one we used above. It shows the DAG of package objects, excluding implicit dependencies. It is concise, but filters out many details.

`bag-emerged`
This is the package DAG, *including* implicit inputs.

For instance, the following command:

```
guix graph --type=bag-emerged coreutils | dot -Tpdf > dag.pdf
```

... yields this bigger graph:



At the bottom of the graph, we see all the implicit inputs of *gnu-build-system* (see Section 5.2 [Build Systems], page 42).

Now, note that the dependencies of those implicit inputs—that is, the *bootstrap dependencies* (see Section 7.7 [Bootstrapping], page 131)—are not shown here, for conciseness.

`bag` Similar to `bag-emerged`, but this time including all the bootstrap dependencies.

`derivations`
This is the most detailed representation: It shows the DAG of derivations (see Section 5.4 [Derivations], page 47) and plain store items. Compared to the above representation, many additional nodes are visible, including builds scripts, patches, Guile modules, etc.

All the above types correspond to *build-time dependencies*. The following graph type represents the *run-time dependencies*:

`references`
: This is the graph of *references* of a package output, as returned by `guix gc --references` (see Section 3.5 [Invoking guix gc], page 22).

  If the given package output is not available in the store, `guix graph` attempts to obtain dependency information from substitutes.

The available options are the following:

`--type=type`
`-t type`
: Produce a graph output of *type*, where *type* must be one of the values listed above.

`--list-types`
: List the supported graph types.

`--expression=expr`
`-e expr`
: Consider the package *expr* evaluates to.

  This is useful to precisely refer to a package, as in this example:

```
guix graph -e '(@@ (gnu packages commencement) gnu-make-final)'
```

6.10 Invoking `guix environment`

The purpose of `guix environment` is to assist hackers in creating reproducible development environments without polluting their package profile. The `guix environment` tool takes one or more packages, builds all of the necessary inputs, and creates a shell environment to use them.

The general syntax is:

```
guix environment options package...
```

The following example spawns a new shell set up for the development of GNU Guile:

```
guix environment guile
```

If the specified packages are not built yet, `guix environment` automatically builds them. The new shell's environment is an augmented version of the environment that `guix environment` was run in. It contains the necessary search paths for building the given package added to the existing environment variables. To create a "pure" environment in which the original environment variables have been unset, use the `--pure` option[4].

`guix environment` defines the `GUIX_ENVIRONMENT` variable in the shell it spaws. This allows users to, say, define a specific prompt for development environments in their `.bashrc` (see Section "Bash Startup Files" in *The GNU Bash Reference Manual*):

```
if [ -n "$GUIX_ENVIRONMENT" ]
then
    export PS1="\u@\h \w [dev]\$ "
fi
```

[4] Users sometimes wrongfully augment environment variables such as `PATH` in their `~/.bashrc` file. As a consequence, when `guix environment` launches it, Bash may read `~/.bashrc`, thereby introducing "impurities" in these environment variables. It is an error to define such environment variables in `.bashrc`; instead, they should be defined in `.bash_profile`, which is sourced only by log-in shells. See Section "Bash Startup Files" in *The GNU Bash Reference Manual*, for details on Bash start-up files.

Additionally, more than one package may be specified, in which case the union of the inputs for the given packages are used. For example, the command below spawns a shell where all of the dependencies of both Guile and Emacs are available:

```
guix environment guile emacs
```

Sometimes an interactive shell session is not desired. An arbitrary command may be invoked by placing the `--` token to separate the command from the rest of the arguments:

```
guix environment guile -- make -j4
```

In other situations, it is more convenient to specify the list of packages needed in the environment. For example, the following command runs `python` from an environment containing Python 2.7 and NumPy:

```
guix environment --ad-hoc python2-numpy python-2.7 -- python
```

Furthermore, one might want the dependencies of a package and also some additional packages that are not build-time or runtime dependencies, but are useful when developing nonetheless. Because of this, the `--ad-hoc` flag is positional. Packages appearing before `--ad-hoc` are interpreted as packages whose dependencies will be added to the environment. Packages appearing after are interpreted as packages that will be added to the environment directly. For example, the following command creates a Guix development environment that additionally includes Git and strace:

```
guix environment guix --ad-hoc git strace
```

Sometimes it is desirable to isolate the environment as much as possible, for maximal purity and reproducibility. In particular, when using Guix on a host distro that is not GuixSD, it is desirable to prevent access to `/usr/bin` and other system-wide resources from the development environment. For example, the following command spawns a Guile REPL in a "container" where only the store and the current working directory are mounted:

```
guix environment --ad-hoc --container guile -- guile
```

Note: The `--container` option requires Linux-libre 3.19 or newer.

The available options are summarized below.

`--expression=`*expr*
`-e` *expr*
: Create an environment for the package or list of packages that *expr* evaluates to.

  For example, running:

  ```
  guix environment -e '(@ (gnu packages maths) petsc-openmpi)'
  ```

  starts a shell with the environment for this specific variant of the PETSc package.

  Running:

  ```
  guix environment --ad-hoc -e '(@ (gnu) %base-packages)'
  ```

  starts a shell with all the GuixSD base packages available.

`--load=`*file*
`-l` *file*
: Create an environment for the package or list of packages that the code within *file* evaluates to.

  As an example, *file* might contain a definition like this (see Section 5.1 [Defining Packages], page 37):

```
(use-modules (guix)
             (gnu packages gdb)
             (gnu packages autotools)
             (gnu packages texinfo))

;; Augment the package definition of GDB with the build tools
;; needed when developing GDB (and which are not needed when
;; simply installing it.)
(package (inherit gdb)
  (native-inputs `(("autoconf" ,autoconf-2.64)
                   ("automake" ,automake)
                   ("texinfo" ,texinfo)
                   ,@(package-native-inputs gdb))))
```

`--ad-hoc` Include all specified packages in the resulting environment, as if an *ad hoc* package were defined with them as inputs. This option is useful for quickly creating an environment without having to write a package expression to contain the desired inputs.

For instance, the command:

```
guix environment --ad-hoc guile guile-sdl -- guile
```

runs `guile` in an environment where Guile and Guile-SDL are available.

Note that this example implicitly asks for the default output of `guile` and `guile-sdl` but it is possible to ask for a specific output—e.g., `glib:bin` asks for the `bin` output of `glib` (see Section 3.4 [Packages with Multiple Outputs], page 21).

This option may be composed with the default behavior of `guix environment`. Packages appearing before `--ad-hoc` are interpreted as packages whose dependencies will be added to the environment, the default behavior. Packages appearing after are interpreted as packages that will be added to the environment directly.

`--pure` Unset existing environment variables when building the new environment. This has the effect of creating an environment in which search paths only contain package inputs.

`--search-paths`
Display the environment variable definitions that make up the environment.

`--system=`*system*
`-s` *system* Attempt to build for *system*—e.g., `i686-linux`.

`--container`
`-C` Run *command* within an isolated container. The current working directory outside the container is mapped to `/env` inside the container. Additionally, the spawned process runs as the current user outside the container, but has root privileges in the context of the container.

`--network`
`-N` For containers, share the network namespace with the host system. Containers created without this flag only have access to the loopback device.

`--expose=`*source*`[=`*target*`]`
: For containers, expose the file system *source* from the host system as the read-only file system *target* within the container. If *target* is not specified, *source* is used as the target mount point in the container.

  The example below spawns a Guile REPL in a container in which the user's home directory is accessible read-only via the `/exchange` directory:

  ```
  guix environment --container --expose=$HOME=/exchange guile -- guile
  ```

`--share=`*source*`[=`*target*`]`
: For containers, share the file system *source* from the host system as the writable file system *target* within the container. If *target* is not specified, *source* is used as the target mount point in the container.

  The example below spawns a Guile REPL in a container in which the user's home directory is accessible for both reading and writing via the `/exchange` directory:

  ```
  guix environment --container --share=$HOME=/exchange guile -- guile
  ```

It also supports all of the common build options that `guix build` supports (see Section 6.1 [Invoking guix build], page 60).

6.11 Invoking `guix publish`

The purpose of `guix publish` is to enable users to easily share their store with others, which can then use it as a substitute server (see Section 3.3 [Substitutes], page 20).

When `guix publish` runs, it spawns an HTTP server which allows anyone with network access to obtain substitutes from it. This means that any machine running Guix can also act as if it were a build farm, since the HTTP interface is compatible with Hydra, the software behind the `hydra.gnu.org` build farm.

For security, each substitute is signed, allowing recipients to check their authenticity and integrity (see Section 3.3 [Substitutes], page 20). Because `guix publish` uses the system's signing key, which is only readable by the system administrator, it must be started as root; the `--user` option makes it drop root privileges early on.

The signing key pair must be generated before `guix publish` is launched, using `guix archive --generate-key` (see Section 3.7 [Invoking guix archive], page 24).

The general syntax is:

```
guix publish options...
```

Running `guix publish` without any additional arguments will spawn an HTTP server on port 8080:

```
guix publish
```

Once a publishing server has been authorized (see Section 3.7 [Invoking guix archive], page 24), the daemon may download substitutes from it:

```
guix-daemon --substitute-urls=http://example.org:8080
```

The following options are available:

`--port=`*port*
`-p` *port* Listen for HTTP requests on *port*.

`--listen=`*host*
: Listen on the network interface for *host*. The default is to accept connections from any interface.

`--user=`*user*
`-u` *user*
: Change privileges to *user* as soon as possible—i.e., once the server socket is open and the signing key has been read.

`--repl[=`*port*`]`
`-r [`*port*`]`
: Spawn a Guile REPL server (see Section "REPL Servers" in *GNU Guile Reference Manual*) on *port* (37146 by default). This is used primarily for debugging a running `guix publish` server.

Enabling `guix publish` on a GuixSD system is a one-liner: just add a call to `guix-publish-service` in the `services` field of the `operating-system` declaration (see [guix-publish-service], page 101).

6.12 Invoking `guix challenge`

Do the binaries provided by this server really correspond to the source code it claims to build? Is this package's build process deterministic? These are the questions the `guix challenge` command attempts to answer.

The former is obviously an important question: Before using a substitute server (see Section 3.3 [Substitutes], page 20), you'd rather *verify* that it provides the right binaries, and thus *challenge* it. The latter is what enables the former: If package builds are deterministic, then independent builds of the package should yield the exact same result, bit for bit; if a server provides a binary different from the one obtained locally, it may be either corrupt or malicious.

We know that the hash that shows up in `/gnu/store` file names is the hash of all the inputs of the process that built the file or directory—compilers, libraries, build scripts, etc. (see Chapter 1 [Introduction], page 2). Assuming deterministic build processes, one store file name should map to exactly one build output. `guix challenge` checks whether there is, indeed, a single mapping by comparing the build outputs of several independent builds of any given store item.

The command's output looks like this:

```
$ guix challenge --substitute-urls="http://hydra.gnu.org http://guix.example.org"
updating list of substitutes from 'http://hydra.gnu.org'... 100.0%
updating list of substitutes from 'http://guix.example.org'... 100.0%
/gnu/store/...-openssl-1.0.2d contents differ:
  local hash: 0725l22r5jnzazaacncwsvp9kgf42266ayyp814v7djxs7nk963q
  http://hydra.gnu.org/nar/...-openssl-1.0.2d: 0725l22r5jnzazaacncwsvp9kgf42266ayyp814v7djxs7nk963q
  http://guix.example.org/nar/...-openssl-1.0.2d: 1zy4fmaaqcnjrzzajkdn3f5gmjk754b43qkq47llbyak9z0qjyim
/gnu/store/...-git-2.5.0 contents differ:
  local hash: 00p3bmryhjxrhpn2gxs2fy0a15lnip05l97205pgbk5ra395hyha
  http://hydra.gnu.org/nar/...-git-2.5.0: 069nb85bv4d4a6slrwjdy8v1cn4cwspm3kdbmyb81d6zckj3nq9f
  http://guix.example.org/nar/...-git-2.5.0: 0mdqa9w1p6cmli6976v4wi0sw9r4p5prkj7lzfd1877wk11c9c73
/gnu/store/...-pius-2.1.1 contents differ:
  local hash: 0k4v3m9z1zp8xzzizb7d8kjj72f9172xv078sq4wl73vnq9ig3ax
  http://hydra.gnu.org/nar/...-pius-2.1.1: 0k4v3m9z1zp8xzzizb7d8kjj72f9172xv078sq4wl73vnq9ig3ax
  http://guix.example.org/nar/...-pius-2.1.1: 1cy25x1a4fzq5rk0pmvc8xhwyffnqz95h2bpvqsz2mpvlbccy0gs
```

In this example, `guix challenge` first scans the store to determine the set of locally-built derivations—as opposed to store items that were downloaded from a substitute server—and then queries all the substitute servers. It then reports those store items for which the servers obtained a result different from the local build.

As an example, `guix.example.org` always gets a different answer. Conversely, `hydra.gnu.org` agrees with local builds, except in the case of Git. This might indicate that the build process of Git is non-deterministic, meaning that its output varies as a function of various things that Guix does not fully control, in spite of building packages in isolated environments (see Section 3.1 [Features], page 13). Most common sources of non-determinism include the addition of timestamps in build results, the inclusion of random numbers, and directory listings sorted by inode number. See `http://reproducible.debian.net/howto/`, for more information.

To find out what's wrong with this Git binary, we can do something along these lines (see Section 3.7 [Invoking guix archive], page 24):

```
$ wget -q -O - http://hydra.gnu.org/nar/...-git-2.5.0 \
   | guix archive -x /tmp/git
$ diff -ur --no-dereference /gnu/store/...-git.2.5.0 /tmp/git
```

This command shows the difference between the files resulting from the local build, and the files resulting from the build on `hydra.gnu.org` (see Section "Overview" in *Comparing and Merging Files*). The `diff` command works great for text files. When binary files differ, a better option is Diffoscope, a tool that helps visualize differences for all kinds of files.

Once you've done that work, you can tell whether the differences are due to a non-deterministic build process or to a malicious server. We try hard to remove sources of non-determinism in packages to make it easier to verify substitutes, but of course, this is a process, one that involves not just Guix but a large part of the free software community. In the meantime, `guix challenge` is one tool to help address the problem.

If you are writing packages for Guix, you are encouraged to check whether `hydra.gnu.org` and other substitute servers obtain the same build result as you did with:

```
$ guix challenge package
```

... where *package* is a package specification such as `guile-2.0` or `glibc:debug`.

The general syntax is:

```
guix challenge options [packages...]
```

The one option that matters is:

`--substitute-urls=urls`
: Consider *urls* the whitespace-separated list of substitute source URLs to compare to.

6.13 Invoking `guix container`

> **Note:** As of version 0.9.0, this tool is experimental. The interface is subject to radical change in the future.

The purpose of `guix container` is to manipulate processes running within an isolated environment, commonly known as a "container", typically created by the `guix environment`

(see Section 6.10 [Invoking guix environment], page 75) and `guix system container` (see Section 7.2.13 [Invoking guix system], page 114) commands.

The general syntax is:

```
guix container action options...
```

action specifies the operation to perform with a container, and *options* specifies the context-specific arguments for the action.

The following actions are available:

`exec` Execute a command within the context of a running container.

The syntax is:

```
guix container exec pid program arguments...
```

pid specifies the process ID of the running container. *program* specifies an executable file name within the container's root file system. *arguments* are the additional options that will be passed to *program*.

The following command launches an interactive login shell inside a GuixSD container, started by `guix system container`, and whose process ID is 9001:

```
guix container exec 9001 /run/current-system/profile/bin/bash --login
```

Note that the *pid* cannot be the parent process of a container. It must be the container's PID 1 or one of its child processes.

7 GNU Distribution

Guix comes with a distribution of the GNU system consisting entirely of free software[1]. The distribution can be installed on its own (see Section 7.1 [System Installation], page 82), but it is also possible to install Guix as a package manager on top of an installed GNU/Linux system (see Chapter 2 [Installation], page 3). To distinguish between the two, we refer to the standalone distribution as the Guix System Distribution, or GuixSD.

The distribution provides core GNU packages such as GNU libc, GCC, and Binutils, as well as many GNU and non-GNU applications. The complete list of available packages can be browsed on-line or by running `guix package` (see Section 3.2 [Invoking guix package], page 14):

```
guix package --list-available
```

Our goal has been to provide a practical 100% free software distribution of Linux-based and other variants of GNU, with a focus on the promotion and tight integration of GNU components, and an emphasis on programs and tools that help users exert that freedom.

Packages are currently available on the following platforms:

`x86_64-linux`
: Intel/AMD `x86_64` architecture, Linux-Libre kernel;

`i686-linux`
: Intel 32-bit architecture (IA32), Linux-Libre kernel;

`armhf-linux`
: ARMv7-A architecture with hard float, Thumb-2 and NEON, using the EABI hard-float ABI, and Linux-Libre kernel.

`mips64el-linux`
: little-endian 64-bit MIPS processors, specifically the Loongson series, n32 application binary interface (ABI), and Linux-Libre kernel.

GuixSD itself is currently only available on `i686` and `x86_64`.

For information on porting to other architectures or kernels, See Section 7.8 [Porting], page 134.

Building this distribution is a cooperative effort, and you are invited to join! See Chapter 8 [Contributing], page 135, for information about how you can help.

7.1 System Installation

This section explains how to install the Guix System Distribution on a machine. The Guix package manager can also be installed on top of a running GNU/Linux system, see Chapter 2 [Installation], page 3.

7.1.1 Limitations

As of version 0.9.0, the Guix System Distribution (GuixSD) is not production-ready. It may contain bugs and lack important features. Thus, if you are looking for a stable production system that respects your freedom as a computer user, a good solution at this point is to

[1] The term "free" here refers to the freedom provided to users of that software.

consider one of more established GNU/Linux distributions. We hope you can soon switch to the GuixSD without fear, of course. In the meantime, you can also keep using your distribution and try out the package manager on top of it (see Chapter 2 [Installation], page 3).

Before you proceed with the installation, be aware of the following noteworthy limitations applicable to version 0.9.0:

- The installation process does not include a graphical user interface and requires familiarity with GNU/Linux (see the following subsections to get a feel of what that means.)
- The system does not yet provide full GNOME and KDE desktops. Xfce and Enlightenment are available though, if graphical desktop environments are your thing, as well as a number of X11 window managers.
- Support for the Logical Volume Manager (LVM) is missing.
- Few system services are currently supported out-of-the-box (see Section 7.2.7 [Services], page 97).
- More than 2,000 packages are available, but you may occasionally find that a useful package is missing.

You've been warned. But more than a disclaimer, this is an invitation to report issues (and success stories!), and join us in improving it. See Chapter 8 [Contributing], page 135, for more info.

7.1.2 USB Stick Installation

An installation image for USB sticks can be downloaded from '`ftp://alpha.gnu.org/gnu/guix/guixsd-usb-i`
where *system* is one of:

`x86_64-linux`
: for a GNU/Linux system on Intel/AMD-compatible 64-bit CPUs;

`i686-linux`
: for a 32-bit GNU/Linux system on Intel-compatible CPUs.

This image contains a single partition with the tools necessary for an installation. It is meant to be copied *as is* to a large-enough USB stick.

To copy the image to a USB stick, follow these steps:

1. Decompress the image using the `xz` command:

   ```
   xz -d guixsd-usb-install-0.9.0.system.xz
   ```

2. Insert a USB stick of 1 GiB or more in your machine, and determine its device name. Assuming that USB stick is known as `/dev/sdX`, copy the image with:

   ```
   dd if=guixsd-usb-install-0.9.0.x86_64 of=/dev/sdX
   ```

   Access to `/dev/sdX` usually requires root privileges.

Once this is done, you should be able to reboot the system and boot from the USB stick. The latter usually requires you to get in the BIOS' boot menu, where you can choose to boot from the USB stick.

7.1.3 Preparing for Installation

Once you have successfully booted the image on the USB stick, you should end up with a root prompt. Several console TTYs are configured and can be used to run commands as root. TTY2 shows this documentation, browsable using the Info reader commands (see Section "Help" in *Info: An Introduction*).

To install the system, you would:

1. Configure the network, by running `ifconfig eno1 up && dhclient eno1` (to get an automatically assigned IP address from the wired network interface controller[2]), or using the `ifconfig` command.

   The system automatically loads drivers for your network interface controllers.

   Setting up network access is almost always a requirement because the image does not contain all the software and tools that may be needed.

2. Unless this has already been done, you must partition, and then format the target partition.

   Preferably, assign partitions a label so that you can easily and reliably refer to them in `file-system` declarations (see Section 7.2.3 [File Systems], page 91). This is typically done using the `-L` option of `mkfs.ext4` and related commands.

   The installation image includes Parted (see Section "Overview" in *GNU Parted User Manual*), `fdisk`, Cryptsetup/LUKS for disk encryption, and e2fsprogs, the suite of tools to manipulate ext2/ext3/ext4 file systems.

3. Once that is done, mount the target root partition under `/mnt`.

4. Lastly, run `deco start cow-store /mnt`.

   This will make `/gnu/store` copy-on-write, such that packages added to it during the installation phase will be written to the target disk rather than kept in memory.

7.1.4 Proceeding with the Installation

With the target partitions ready, you now have to edit a file and provide the declaration of the operating system to be installed. To that end, the installation system comes with two text editors: GNU nano (see *GNU nano Manual*), and GNU Zile, an Emacs clone. It is better to store that file on the target root file system, say, as `/mnt/etc/config.scm`.

See Section 7.2.1 [Using the Configuration System], page 85, for examples of operating system configurations. These examples are available under `/etc/configuration` in the installation image, so you can copy them and use them as a starting point for your own configuration.

Once you are done preparing the configuration file, the new system must be initialized (remember that the target root file system is mounted under `/mnt`):

```
guix system init /mnt/etc/config.scm /mnt
```

This will copy all the necessary files, and install GRUB on `/dev/sdX`, unless you pass the `--no-grub` option. For more information, see Section 7.2.13 [Invoking guix system], page 114.

[2] The name `eno1` is for the first on-board Ethernet controller. The interface name for an Ethernet controller that is in the first slot of the first PCI bus, for instance, would be `enp1s0`. Use `ifconfig -a` to list all the available network interfaces.

This command may trigger downloads or builds of missing packages, which can take some time.

Once that command has completed—and hopefully succeeded!—you can run `reboot` and boot into the new system. The `root` password in the new system is initially empty; other users' passwords need to be initialized by running the `passwd` command as `root`, unless your configuration specifies otherwise (see [user-account-password], page 95).

Join us on `#guix` on the Freenode IRC network or on `guix-devel@gnu.org` to share your experience—good or not so good.

7.1.5 Building the Installation Image

The installation image described above was built using the `guix system` command, specifically:

```
guix system disk-image --image-size=850MiB gnu/system/install.scm
```

See Section 7.2.13 [Invoking guix system], page 114, for more information. See `gnu/system/install.scm` in the source tree for more information about the installation image.

7.2 System Configuration

The Guix System Distribution supports a consistent whole-system configuration mechanism. By that we mean that all aspects of the global system configuration—such as the available system services, timezone and locale settings, user accounts—are declared in a single place. Such a *system configuration* can be *instantiated*—i.e., effected.

One of the advantages of putting all the system configuration under the control of Guix is that it supports transactional system upgrades, and makes it possible to roll-back to a previous system instantiation, should something go wrong with the new one (see Section 3.1 [Features], page 13). Another one is that it makes it easy to replicate the exact same configuration across different machines, or at different points in time, without having to resort to additional administration tools layered on top of the system's own tools.

This section describes this mechanism. First we focus on the system administrator's viewpoint—explaining how the system is configured and instantiated. Then we show how this mechanism can be extended, for instance to support new system services.

7.2.1 Using the Configuration System

The operating system is configured by providing an `operating-system` declaration in a file that can then be passed to the `guix system` command (see Section 7.2.13 [Invoking guix system], page 114). A simple setup, with the default system services, the default Linux-Libre kernel, initial RAM disk, and boot loader looks like this:

```
;; This is an operating system configuration template
;; for a "bare bones" setup, with no X11 display server.

(use-modules (gnu))
(use-service-modules networking ssh)
(use-package-modules admin)
```

```
(operating-system
  (host-name "komputilo")
  (timezone "Europe/Berlin")
  (locale "en_US.UTF-8")

  ;; Assuming /dev/sdX is the target hard disk, and "root" is
  ;; the label of the target root file system.
  (bootloader (grub-configuration (device "/dev/sdX")))
  (file-systems (cons (file-system
                        (device "root")
                        (title 'label)
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))

  ;; This is where user accounts are specified.  The "root"
  ;; account is implicit, and is initially created with the
  ;; empty password.
  (users (cons (user-account
                (name "alice")
                (comment "Bob's sister")
                (group "users")

                ;; Adding the account to the "wheel" group
                ;; makes it a sudoer.  Adding it to "audio"
                ;; and "video" allows the user to play sound
                ;; and access the webcam.
                (supplementary-groups '("wheel"
                                        "audio" "video"))
                (home-directory "/home/alice"))
               %base-user-accounts))

  ;; Globally-installed packages.
  (packages (cons tcpdump %base-packages))

  ;; Add services to the baseline: a DHCP client and
  ;; an SSH server.
  (services (cons* (dhcp-client-service)
                   (lsh-service #:port-number 2222)
                   %base-services)))
```

This example should be self-describing. Some of the fields defined above, such as `host-name` and `bootloader`, are mandatory. Others, such as `packages` and `services`, can be omitted, in which case they get a default value.

The `packages` field lists packages that will be globally visible on the system, for all user accounts—i.e., in every user's `PATH` environment variable—in addition to the per-user profiles (see Section 3.2 [Invoking guix package], page 14). The *%base-packages* variable

provides all the tools one would expect for basic user and administrator tasks—including the GNU Core Utilities, the GNU Networking Utilities, the GNU Zile lightweight text editor, `find`, `grep`, etc. The example above adds tcpdump to those, taken from the `(gnu packages admin)` module (see Section 7.5 [Package Modules], page 126).

The `services` field lists *system services* to be made available when the system starts (see Section 7.2.7 [Services], page 97). The `operating-system` declaration above specifies that, in addition to the basic services, we want the `lshd` secure shell daemon listening on port 2222 (see Section 7.2.7.2 [Networking Services], page 101). Under the hood, `lsh-service` arranges so that `lshd` is started with the right command-line options, possibly with supporting configuration files generated as needed (see Section 7.2.14 [Defining Services], page 117).

Occasionally, instead of using the base services as is, you will want to customize them. For instance, to change the configuration of `guix-daemon` and Mingetty (the console log-in), you may write the following instead of *%base-services*:

```
(modify-services %base-services
  (guix-service-type config =>
                     (guix-configuration
                      (inherit config)
                      (use-substitutes? #f)
                      (extra-options '("--gc-keep-outputs"))))
  (mingetty-service-type config =>
                         (mingetty-configuration
                          (inherit config)
                          (motd (plain-file "motd" "Hi there!")))))
```

The effect here is to change the options passed to `guix-daemon` when it is started, as well as the "message of the day" that appears when logging in at the console. See Section 7.2.14.3 [Service Reference], page 120, for more on that.

The configuration for a typical "desktop" usage, with the X11 display server, a desktop environment, network management, power management, and more, would look like this:

```
;; This is an operating system configuration template
;; for a "desktop" setup with X11.

(use-modules (gnu) (gnu system nss))
(use-service-modules desktop)
(use-package-modules xfce ratpoison certs)

(operating-system
  (host-name "antelope")
  (timezone "Europe/Paris")
  (locale "en_US.UTF-8")

  ;; Assuming /dev/sdX is the target hard disk, and "root" is
  ;; the label of the target root file system.
  (bootloader (grub-configuration (device "/dev/sdX")))
  (file-systems (cons (file-system
```

```
                        (device "root")
                        (title 'label)
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))

  (users (cons (user-account
                (name "bob")
                (comment "Alice's brother")
                (group "users")
                (supplementary-groups '("wheel" "netdev"
                                        "audio" "video"))
                (home-directory "/home/bob"))
               %base-user-accounts))

  ;; Add Xfce and Ratpoison; that allows us to choose
  ;; sessions using either of these at the log-in screen.
  (packages (cons* xfce ratpoison    ;desktop environments
                   nss-certs         ;for HTTPS access
                   %base-packages))

  ;; Use the "desktop" services, which include the X11
  ;; log-in service, networking with Wicd, and more.
  (services %desktop-services)

  ;; Allow resolution of '.local' host names with mDNS.
  (name-service-switch %mdns-host-lookup-nss))
```

See Section 7.2.7.4 [Desktop Services], page 104, for the exact list of services provided by *%desktop-services*. See Section 7.2.9 [X.509 Certificates], page 109, for background information about the `nss-certs` package that is used here. See Section 7.2.2 [operating-system Reference], page 89, for details about all the available `operating-system` fields.

Assuming the above snippet is stored in the `my-system-config.scm` file, the `guix system reconfigure my-system-config.scm` command instantiates that configuration, and makes it the default GRUB boot entry (see Section 7.2.13 [Invoking guix system], page 114).

The normal way to change the system's configuration is by updating this file and re-running `guix system reconfigure`. One should never have to touch files in `/etc` or to run commands that modify the system state such as `useradd` or `grub-install`. In fact, you must avoid that since that would not only void your warranty but also prevent you from rolling back to previous versions of your system, should you ever need to.

Speaking of roll-back, each time you run `guix system reconfigure`, a new *generation* of the system is created—without modifying or deleting previous generations. Old system generations get an entry in the GRUB boot menu, allowing you to boot them in case something went wrong with the latest generation. Reassuring, no? The `guix system list-generations` command lists the system generations available on disk.

At the Scheme level, the bulk of an `operating-system` declaration is instantiated with the following monadic procedure (see Section 5.5 [The Store Monad], page 50):

`operating-system-derivation` *os* [Monadic Procedure]

Return a derivation that builds *os*, an `operating-system` object (see Section 5.4 [Derivations], page 47).

The output of the derivation is a single directory that refers to all the packages, configuration files, and other supporting files needed to instantiate *os*.

7.2.2 `operating-system` Reference

This section summarizes all the options available in `operating-system` declarations (see Section 7.2.1 [Using the Configuration System], page 85).

`operating-system` [Data Type]

This is the data type representing an operating system configuration. By that, we mean all the global system configuration, not per-user configuration (see Section 7.2.1 [Using the Configuration System], page 85).

`kernel` (default: *linux-libre*)
: The package object of the operating system kernel to use[3].

`kernel-arguments` (default: `'()`)
: List of strings or gexps representing additional arguments to pass on the kernel's command-line—e.g., `("console=ttyS0")`.

`bootloader`
: The system bootloader configuration object. See Section 7.2.12 [GRUB Configuration], page 113.

`initrd` (default: `base-initrd`)
: A two-argument monadic procedure that returns an initial RAM disk for the Linux kernel. See Section 7.2.11 [Initial RAM Disk], page 112.

`firmware` (default: *%base-firmware*)
: List of firmware packages loadable by the operating system kernel.

The default includes firmware needed for Atheros-based WiFi devices (Linux-libre module `ath9k`.)

`host-name`
: The host name.

`hosts-file`
: A file-like object (see Section 5.6 [G-Expressions], page 53) for use as `/etc/hosts` (see Section "Host Names" in *The GNU C Library Reference Manual*). The default is a file with entries for `localhost` and *host-name*.

`mapped-devices` (default: `'()`)
: A list of mapped devices. See Section 7.2.4 [Mapped Devices], page 93.

`file-systems`
: A list of file systems. See Section 7.2.3 [File Systems], page 91.

[3] Currently only the Linux-libre kernel is supported. In the future, it will be possible to use the GNU Hurd.

`swap-devices` (default: `'()`)
: A list of strings identifying devices to be used for "swap space" (see Section "Memory Concepts" in *The GNU C Library Reference Manual*). For example, `'("/dev/sda3")`.

`users` (default: `%base-user-accounts`)
`groups` (default: *%base-groups*)
: List of user accounts and groups. See Section 7.2.5 [User Accounts], page 94.

`skeletons` (default: `(default-skeletons)`)
: A monadic list of pairs of target file name and files. These are the files that will be used as skeletons as new accounts are created.

  For instance, a valid value may look like this:

```
(mlet %store-monad ((bashrc (text-file "bashrc" "\
     export PATH=$HOME/.guix-profile/bin")))
  (return `((".bashrc" ,bashrc))))
```

`issue` (default: *%default-issue*)
: A string denoting the contents of the `/etc/issue` file, which is what displayed when users log in on a text console.

`packages` (default: *%base-packages*)
: The set of packages installed in the global profile, which is accessible at `/run/current-system/profile`.

  The default set includes core utilities, but it is good practice to install non-core utilities in user profiles (see Section 3.2 [Invoking guix package], page 14).

`timezone` A timezone identifying string—e.g., `"Europe/Paris"`.

`locale` (default: `"en_US.utf8"`)
: The name of the default locale (see Section "Locale Names" in *The GNU C Library Reference Manual*). See Section 7.2.6 [Locales], page 96, for more information.

`locale-definitions` (default: *%default-locale-definitions*)
: The list of locale definitions to be compiled and that may be used at run time. See Section 7.2.6 [Locales], page 96.

`locale-libcs` (default: `(list` *glibc*`)`)
: The list of GNU libc packages whose locale data and tools are used to build the locale definitions. See Section 7.2.6 [Locales], page 96, for compatibility considerations that justify this option.

`name-service-switch` (default: *%default-nss*)
: Configuration of libc's name service switch (NSS)—a `<name-service-switch>` object. See Section 7.2.10 [Name Service Switch], page 110, for details.

`services` (default: *%base-services*)
: A list of monadic values denoting system services. See Section 7.2.7 [Services], page 97.

`pam-services` (default: `(base-pam-services)`)
: Linux *pluggable authentication module* (PAM) services.

`setuid-programs` (default: *%setuid-programs*)
: List of string-valued G-expressions denoting setuid programs. See Section 7.2.8 [Setuid Programs], page 108.

`sudoers-file` (default: *%sudoers-specification*)
: The contents of the `/etc/sudoers` file as a file-like object (see Section 5.6 [G-Expressions], page 53).

  This file specifies which users can use the `sudo` command, what they are allowed to do, and what privileges they may gain. The default is that only `root` and members of the `wheel` group may use `sudo`.

7.2.3 File Systems

The list of file systems to be mounted is specified in the `file-systems` field of the operating system's declaration (see Section 7.2.1 [Using the Configuration System], page 85). Each file system is declared using the `file-system` form, like this:

```
(file-system
  (mount-point "/home")
  (device "/dev/sda3")
  (type "ext4"))
```

As usual, some of the fields are mandatory—those shown in the example above—while others can be omitted. These are described below.

`file-system` [Data Type]
: Objects of this type represent file systems to be mounted. They contain the following members:

  `type` This is a string specifying the type of the file system—e.g., `"ext4"`.

  `mount-point`
  : This designates the place where the file system is to be mounted.

  `device` This names the "source" of the file system. By default it is the name of a node under `/dev`, but its meaning depends on the `title` field described below.

  `title` (default: `'device`)
  : This is a symbol that specifies how the `device` field is to be interpreted.

    When it is the symbol `device`, then the `device` field is interpreted as a file name; when it is `label`, then `device` is interpreted as a partition label name; when it is `uuid`, `device` is interpreted as a partition unique identifier (UUID).

    UUIDs may be converted from their string representation (as shown by the `tune2fs -l` command) using the `uuid` form, like this:

    ```
    (file-system
      (mount-point "/home")
      (type "ext4")
    ```

```
(title 'uuid)
(device (uuid "4dab5feb-d176-45de-b287-9b0a6e4c01cb")))
```

The `label` and `uuid` options offer a way to refer to disk partitions without having to hard-code their actual device name[4].

However, when a file system's source is a mapped device (see Section 7.2.4 [Mapped Devices], page 93), its `device` field *must* refer to the mapped device name—e.g., `/dev/mapper/root-partition`—and consequently `title` must be set to `'device`. This is required so that the system knows that mounting the file system depends on having the corresponding device mapping established.

`flags` (default: `'()`)
: This is a list of symbols denoting mount flags. Recognized flags include `read-only`, `bind-mount`, `no-dev` (disallow access to special files), `no-suid` (ignore setuid and setgid bits), and `no-exec` (disallow program execution.)

`options` (default: `#f`)
: This is either `#f`, or a string denoting mount options.

`needed-for-boot?` (default: `#f`)
: This Boolean value indicates whether the file system is needed when booting. If that is true, then the file system is mounted when the initial RAM disk (initrd) is loaded. This is always the case, for instance, for the root file system.

`check?` (default: `#t`)
: This Boolean indicates whether the file system needs to be checked for errors before being mounted.

`create-mount-point?` (default: `#f`)
: When true, the mount point is created if it does not exist yet.

`dependencies` (default: `'()`)
: This is a list of `<file-system>` objects representing file systems that must be mounted before (and unmounted after) this one.

  As an example, consider a hierarchy of mounts: `/sys/fs/cgroup` is a dependency of `/sys/fs/cgroup/cpu` and `/sys/fs/cgroup/memory`.

The `(gnu system file-systems)` exports the following useful variables.

`%base-file-systems` [Scheme Variable]
: These are essential file systems that are required on normal systems, such as *%pseudo-terminal-file-system* and *%immutable-store* (see below.) Operating system declarations should always contain at least these.

[4] Note that, while it is tempting to use `/dev/disk/by-uuid` and similar device names to achieve the same result, this is not recommended: These special device nodes are created by the udev daemon and may be unavailable at the time the device is mounted.

`%pseudo-terminal-file-system` [Scheme Variable]
: This is the file system to be mounted as `/dev/pts`. It supports *pseudo-terminals* created *via* `openpty` and similar functions (see Section "Pseudo-Terminals" in *The GNU C Library Reference Manual*). Pseudo-terminals are used by terminal emulators such as `xterm`.

`%shared-memory-file-system` [Scheme Variable]
: This file system is mounted as `/dev/shm` and is used to support memory sharing across processes (see Section "Memory-mapped I/O" in *The GNU C Library Reference Manual*).

`%immutable-store` [Scheme Variable]
: This file system performs a read-only "bind mount" of `/gnu/store`, making it read-only for all the users including `root`. This prevents against accidental modification by software running as `root` or by system administrators.

  The daemon itself is still able to write to the store: it remounts it read-write in its own "name space."

`%binary-format-file-system` [Scheme Variable]
: The `binfmt_misc` file system, which allows handling of arbitrary executable file types to be delegated to user space. This requires the `binfmt.ko` kernel module to be loaded.

`%fuse-control-file-system` [Scheme Variable]
: The `fusectl` file system, which allows unprivileged users to mount and unmount user-space FUSE file systems. This requires the `fuse.ko` kernel module to be loaded.

7.2.4 Mapped Devices

The Linux kernel has a notion of *device mapping*: a block device, such as a hard disk partition, can be *mapped* into another device, with additional processing over the data that flows through it[5]. A typical example is encryption device mapping: all writes to the mapped device are encrypted, and all reads are deciphered, transparently.

Mapped devices are declared using the `mapped-device` form:

```
(mapped-device
  (source "/dev/sda3")
  (target "home")
  (type luks-device-mapping))
```

This example specifies a mapping from `/dev/sda3` to `/dev/mapper/home` using LUKS—the Linux Unified Key Setup, a standard mechanism for disk encryption. The `/dev/mapper/home` device can then be used as the `device` of a `file-system` declaration (see Section 7.2.3 [File Systems], page 91). The `mapped-device` form is detailed below.

[5] Note that the GNU Hurd makes no difference between the concept of a "mapped device" and that of a file system: both boil down to *translating* input/output operations made on a file to operations on its backing store. Thus, the Hurd implements mapped devices, like file systems, using the generic *translator* mechanism (see Section "Translators" in *The GNU Hurd Reference Manual*).

`mapped-device` [Data Type]

Objects of this type represent device mappings that will be made when the system boots up.

`source` This string specifies the name of the block device to be mapped, such as `"/dev/sda3"`.

`target` This string specifies the name of the mapping to be established. For example, specifying `"my-partition"` will lead to the creation of the `"/dev/mapper/my-partition"` device.

`type` This must be a `mapped-device-kind` object, which specifies how *source* is mapped to *target*.

`luks-device-mapping` [Scheme Variable]

This defines LUKS block device encryption using the `cryptsetup` command, from the same-named package. This relies on the `dm-crypt` Linux kernel module.

7.2.5 User Accounts

User accounts and groups are entirely managed through the `operating-system` declaration. They are specified with the `user-account` and `user-group` forms:

```
(user-account
  (name "alice")
  (group "users")
  (supplementary-groups '("wheel"   ;allow use of sudo, etc.
                          "audio"   ;sound card
                          "video"   ;video devices such as webcams
                          "cdrom")) ;the good ol' CD-ROM
  (comment "Bob's sister")
  (home-directory "/home/alice"))
```

When booting or upon completion of `guix system reconfigure`, the system ensures that only the user accounts and groups specified in the `operating-system` declaration exist, and with the specified properties. Thus, account or group creations or modifications made by directly invoking commands such as `useradd` are lost upon reconfiguration or reboot. This ensures that the system remains exactly as declared.

`user-account` [Data Type]

Objects of this type represent user accounts. The following members may be specified:

`name` The name of the user account.

`group` This is the name (a string) or identifier (a number) of the user group this account belongs to.

`supplementary-groups` (default: `'()`)
Optionally, this can be defined as a list of group names that this account belongs to.

`uid` (default: `#f`)
This is the user ID for this account (a number), or `#f`. In the latter case, a number is automatically chosen by the system when the account is created.

`comment` (default: `""`)
: A comment about the account, such as the account's owner full name.

`home-directory`
: This is the name of the home directory for the account.

`shell` (default: Bash)
: This is a G-expression denoting the file name of a program to be used as the shell (see Section 5.6 [G-Expressions], page 53).

`system?` (default: `#f`)
: This Boolean value indicates whether the account is a "system" account. System accounts are sometimes treated specially; for instance, graphical login managers do not list them.

`password` (default: `#f`)
: You would normally leave this field to `#f`, initialize user passwords as `root` with the `passwd` command, and then let users change it with `passwd`. Passwords set with `passwd` are of course preserved across reboot and reconfiguration.

  If you *do* want to have a preset password for an account, then this field must contain the encrypted password, as a string. See Section "crypt" in *The GNU C Library Reference Manual*, for more information on password encryption, and Section "Encryption" in *GNU Guile Reference Manual*, for information on Guile's `crypt` procedure.

User group declarations are even simpler:

```
(user-group (name "students"))
```

`user-group` [Data Type]
: This type is for, well, user groups. There are just a few fields:

  `name` The group's name.

  `id` (default: `#f`)
  : The group identifier (a number). If `#f`, a new number is automatically allocated when the group is created.

  `system?` (default: `#f`)
  : This Boolean value indicates whether the group is a "system" group. System groups have low numerical IDs.

  `password` (default: `#f`)
  : What, user groups can have a password? Well, apparently yes. Unless `#f`, this field specifies the group's password.

For convenience, a variable lists all the basic user groups one may expect:

`%base-groups` [Scheme Variable]
: This is the list of basic user groups that users and/or packages expect to be present on the system. This includes groups such as "root", "wheel", and "users", as well as groups used to control access to specific devices such as "audio", "disk", and "cdrom".

`%base-user-accounts` [Scheme Variable]

This is the list of basic system accounts that programs may expect to find on a GNU/Linux system, such as the "nobody" account.

Note that the "root" account is not included here. It is a special-case and is automatically added whether or not it is specified.

7.2.6 Locales

A *locale* defines cultural conventions for a particular language and region of the world (see Section "Locales" in *The GNU C Library Reference Manual*). Each locale has a name that typically has the form *`language_territory.codeset`*—e.g., `fr_LU.utf8` designates the locale for the French language, with cultural conventions from Luxembourg, and using the UTF-8 encoding.

Usually, you will want to specify the default locale for the machine using the `locale` field of the `operating-system` declaration (see Section 7.2.2 [operating-system Reference], page 89).

That locale must be among the *locale definitions* that are known to the system—and these are specified in the `locale-definitions` slot of `operating-system`. The default value includes locale definition for some widely used locales, but not for all the available locales, in order to save space.

If the locale specified in the `locale` field is not among the definitions listed in `locale-definitions`, `guix system` raises an error. In that case, you should add the locale definition to the `locale-definitions` field. For instance, to add the North Frisian locale for Germany, the value of that field may be:

```
(cons (locale-definition
        (name "fy_DE.utf8") (source "fy_DE"))
      %default-locale-definitions)
```

Likewise, to save space, one might want `locale-definitions` to list only the locales that are actually used, as in:

```
(list (locale-definition
        (name "ja_JP.eucjp") (source "ja_JP")
        (charset "EUC-JP")))
```

The compiled locale definitions are available at `/run/current-system/locale/X.Y`, where `X.Y` is the libc version, which is the default location where the GNU libc provided by Guix looks for locale data. This can be overridden using the `LOCPATH` environment variable (see [locales-and-locpath], page 11).

The `locale-definition` form is provided by the `(gnu system locale)` module. Details are given below.

`locale-definition` [Data Type]

This is the data type of a locale definition.

`name` The name of the locale. See Section "Locale Names" in *The GNU C Library Reference Manual*, for more information on locale names.

`source` The name of the source for that locale. This is typically the *`language_territory`* part of the locale name.

`charset` (default: `"UTF-8"`)
The "character set" or "code set" for that locale, as defined by IANA.

`%default-locale-definitions` [Scheme Variable]
An arbitrary list of commonly used UTF-8 locales, used as the default value of the `locale-definitions` field of `operating-system` declarations.

These locale definitions use the *normalized codeset* for the part that follows the dot in the name (see Section "Using gettextized software" in *The GNU C Library Reference Manual*). So for instance it has `uk_UA.utf8` but *not*, say, `uk_UA.UTF-8`.

7.2.6.1 Locale Data Compatibility Considerations

`operating-system` declarations provide a `locale-libcs` field to specify the GNU libc packages that are used to compile locale declarations (see Section 7.2.2 [operating-system Reference], page 89). "Why would I care?", you may ask. Well, it turns out that the binary format of locale data is occasionally incompatible from one libc version to another.

For instance, a program linked against libc version 2.21 is unable to read locale data produced with libc 2.22; worse, that program *aborts* instead of simply ignoring the incompatible locale data[6]. Similarly, a program linked against libc 2.22 can read most, but not all, the locale data from libc 2.21 (specifically, `LC_COLLATE` data is incompatible); thus calls to `setlocale` may fail, but programs will not abort.

The "problem" in GuixSD is that users have a lot of freedom: They can choose whether and when to upgrade software in their profiles, and might be using a libc version different from the one the system administrator used to build the system-wide locale data.

Fortunately, unprivileged users can also install their own locale data and define *GUIX_LOCPATH* accordingly (see [locales-and-locpath], page 11).

Still, it is best if the system-wide locale data at `/run/current-system/locale` is built for all the libc versions actually in use on the system, so that all the programs can access it—this is especially crucial on a multi-user system. To do that, the administrator can specify several libc packages in the `locale-libcs` field of `operating-system`:

```
(use-package-modules base)

(operating-system
  ;; ...
  (locale-libcs (list glibc-2.21 (canonical-package glibc))))
```

This example would lead to a system containing locale definitions for both libc 2.21 and the current version of libc in `/run/current-system/locale`.

7.2.7 Services

An important part of preparing an `operating-system` declaration is listing *system services* and their configuration (see Section 7.2.1 [Using the Configuration System], page 85). System services are typically daemons launched when the system boots, or other actions needed at that time—e.g., configuring network access.

[6] Versions 2.23 and later of GNU libc will simply skip the incompatible locale data, which is already an improvement.

Services are managed by GNU dmd (see Section "Introduction" in *GNU dmd Manual*). On a running system, the `deco` command allows you to list the available services, show their status, start and stop them, or do other specific operations (see Section "Jump Start" in *GNU dmd Manual*). For example:

```
# deco status dmd
```

The above command, run as `root`, lists the currently defined services. The `deco doc` command shows a synopsis of the given service:

```
# deco doc nscd
Run libc's name service cache daemon (nscd).
```

The `start`, `stop`, and `restart` sub-commands have the effect you would expect. For instance, the commands below stop the nscd service and restart the Xorg display server:

```
# deco stop nscd
Service nscd has been stopped.
# deco restart xorg-server
Service xorg-server has been stopped.
Service xorg-server has been started.
```

The following sections document the available services, starting with the core services, that may be used in an `operating-system` declaration.

7.2.7.1 Base Services

The `(gnu services base)` module provides definitions for the basic services that one expects from the system. The services exported by this module are listed below.

`%base-services` [Scheme Variable]

This variable contains a list of basic services[7] one would expect from the system: a login service (mingetty) on each tty, syslogd, libc's name service cache daemon (nscd), the udev device manager, and more.

This is the default value of the `services` field of `operating-system` declarations. Usually, when customizing a system, you will want to append services to *%base-services*, like this:

```
(cons* (avahi-service) (lsh-service) %base-services)
```

`host-name-service` *name* [Scheme Procedure]

Return a service that sets the host name to *name*.

`mingetty-service` *config* [Scheme Procedure]

Return a service to run mingetty according to *config*, a `<mingetty-configuration>` object, which specifies the tty to run, among other things.

`mingetty-configuration` [Data Type]

This is the data type representing the configuration of Mingetty, which implements console log-in.

`tty` The name of the console this Mingetty runs on—e.g., `"tty1"`.

`motd` A file-like object containing the "message of the day".

[7] Technically, this is a list of monadic services. See Section 5.5 [The Store Monad], page 50.

`auto-login` (default: `#f`)
: When true, this field must be a string denoting the user name under which the the system automatically logs in. When it is `#f`, a user name and password must be entered to log in.

`login-program` (default: `#f`)
: This must be either `#f`, in which case the default log-in program is used (`login` from the Shadow tool suite), or a gexp denoting the name of the log-in program.

`login-pause?` (default: `#f`)
: When set to `#t` in conjunction with *auto-login*, the user will have to press a key before the log-in shell is launched.

`mingetty` (default: *mingetty*)
: The Mingetty package to use.

`nscd-service` [*config*] [*#:glibc glibc*] [*#:name-services '()*] [Scheme Procedure]
Return a service that runs libc's name service cache daemon (nscd) with the given *config*—an `<nscd-configuration>` object. See Section 7.2.10 [Name Service Switch], page 110, for an example.

`%nscd-default-configuration` [Scheme Variable]
This is the default `<nscd-configuration>` value (see below) used by `nscd-service`. This uses the caches defined by *%nscd-default-caches*; see below.

`nscd-configuration` [Data Type]
This is the type representing the name service cache daemon (nscd) configuration.

`name-services` (default: `'()`)
: List of packages denoting *name services* that must be visible to the nscd—e.g., `(list nss-mdns)`.

`glibc` (default: *glibc*)
: Package object denoting the GNU C Library providing the `nscd` command.

`log-file` (default: `"/var/log/nscd.log"`)
: Name of nscd's log file. This is where debugging output goes when `debug-level` is strictly positive.

`debug-level` (default: `0`)
: Integer denoting the debugging levels. Higher numbers mean more debugging output is logged.

`caches` (default: *%nscd-default-caches*)
: List of `<nscd-cache>` objects denoting things to be cached; see below.

`nscd-cache` [Data Type]
Data type representing a cache database of nscd and its parameters.

`database` This is a symbol representing the name of the database to be cached. Valid values are `passwd`, `group`, `hosts`, and `services`, which designate

the corresponding NSS database (see Section "NSS Basics" in *The GNU C Library Reference Manual*).

`positive-time-to-live`
`negative-time-to-live` (default: `20`)
A number representing the number of seconds during which a positive or negative lookup result remains in cache.

`check-files?` (default: `#t`)
Whether to check for updates of the files corresponding to *database*.

For instance, when *database* is `hosts`, setting this flag instructs nscd to check for updates in `/etc/hosts` and to take them into account.

`persistent?` (default: `#t`)
Whether the cache should be stored persistently on disk.

`shared?` (default: `#t`)
Whether the cache should be shared among users.

`max-database-size` (default: 32 MiB)
Maximum size in bytes of the database cache.

`%nscd-default-caches` [Scheme Variable]
List of `<nscd-cache>` objects used by default by `nscd-configuration` (see above.)

It enables persistent and aggressive caching of service and host name lookups. The latter provides better host name lookup performance, resilience in the face of unreliable name servers, and also better privacy—often the result of host name lookups is in local cache, so external name servers do not even need to be queried.

`syslog-service` [*#:config-file #f*] [Scheme Procedure]
Return a service that runs `syslogd`. If configuration file name *config-file* is not specified, use some reasonable default settings.

`guix-configuration` [Data Type]
This data type represents the configuration of the Guix build daemon. See Section 2.5 [Invoking guix-daemon], page 8, for more information.

`guix` (default: *guix*)
The Guix package to use.

`build-group` (default: `"guixbuild"`)
Name of the group for build user accounts.

`build-accounts` (default: `10`)
Number of build user accounts to create.

`authorize-key?` (default: `#t`)
Whether to authorize the substitute key for `hydra.gnu.org` (see Section 3.3 [Substitutes], page 20).

`use-substitutes?` (default: `#t`)
Whether to use substitutes.

`substitute-urls` (default: *%default-substitute-urls*)
The list of URLs where to look for substitutes by default.

`extra-options` (default: `'()`)
List of extra command-line options for `guix-daemon`.

`lsof` (default: *lsof*)
`lsh` (default: *lsh*)
The lsof and lsh packages to use.

`guix-service` *config* [Scheme Procedure]
Return a service that runs the Guix build daemon according to *config*.

`udev-service` [*#:udev udev*] [Scheme Procedure]
Run *udev*, which populates the `/dev` directory dynamically.

`console-keymap-service` *file* [Scheme Procedure]
Return a service to load console keymap from *file* using `loadkeys` command.

`guix-publish-service` [*#:guix guix*] [*#:port 80*] [*#:host "localhost"*] [Scheme Procedure]
Return a service that runs `guix publish` listening on *host* and *port* (see Section 6.11 [Invoking guix publish], page 78).

This assumes that `/etc/guix` already contains a signing key pair as created by `guix archive --generate-key` (see Section 3.7 [Invoking guix archive], page 24). If that is not the case, the service will fail to start.

7.2.7.2 Networking Services

The `(gnu services networking)` module provides services to configure the network interface.

`dhcp-client-service` [*#:dhcp isc-dhcp*] [Scheme Procedure]
Return a service that runs *dhcp*, a Dynamic Host Configuration Protocol (DHCP) client, on all the non-loopback network interfaces.

`static-networking-service` *interface ip* [*#:gateway #f*] [*#:name-services '()*] [Scheme Procedure]
Return a service that starts *interface* with address *ip*. If *gateway* is true, it must be a string specifying the default network gateway.

`wicd-service` [*#:wicd wicd*] [Scheme Procedure]
Return a service that runs Wicd, a network management daemon that aims to simplify wired and wireless networking.

This service adds the *wicd* package to the global profile, providing several commands to interact with the daemon and configure networking: `wicd-client`, a graphical user interface, and the `wicd-cli` and `wicd-curses` user interfaces.

`ntp-service` [*#:ntp ntp*] [*#:name-service %ntp-servers*] [Scheme Procedure]
Return a service that runs the daemon from *ntp*, the Network Time Protocol package. The daemon will keep the system clock synchronized with that of *servers*.

`%ntp-servers` [Scheme Variable]

List of host names used as the default NTP servers.

`tor-service` [*config-file*] [*#:tor* `tor`] [Scheme Procedure]

Return a service to run the Tor anonymous networking daemon.

The daemon runs as the `tor` unprivileged user. It is passed *config-file*, a file-like object, with an additional `User tor` line. Run `man tor` for information about the configuration file.

`bitlbee-service` [*#:bitlbee bitlbee*] [*#:interface* "*127.0.0.1*"] [*#:port 6667*] [*#:extra-settings* ""] [Scheme Procedure]

Return a service that runs BitlBee, a daemon that acts as a gateway between IRC and chat networks.

The daemon will listen to the interface corresponding to the IP address specified in *interface*, on *port*. `127.0.0.1` means that only local clients can connect, whereas `0.0.0.0` means that connections can come from any networking interface.

In addition, *extra-settings* specifies a string to append to the configuration file.

Furthermore, `(gnu services ssh)` provides the following service.

`lsh-service` [*#:host-key* "*/etc/lsh/host-key*"] [*#:daemonic? #t*] [*#:interfaces '()*] [*#:port-number 22*] [*#:allow-empty-passwords? #f*] [*#:root-login? #f*] [*#:syslog-output? #t*] [*#:x11-forwarding? #t*] [*#:tcp/ip-forwarding? #t*] [*#:password-authentication? #t*] [*#:public-key-authentication? #t*] [*#:initialize? #t*] [Scheme Procedure]

Run the `lshd` program from *lsh* to listen on port *port-number*. *host-key* must designate a file containing the host key, and readable only by root.

When *daemonic?* is true, `lshd` will detach from the controlling terminal and log its output to syslogd, unless one sets *syslog-output?* to false. Obviously, it also makes lsh-service depend on existence of syslogd service. When *pid-file?* is true, `lshd` writes its PID to the file called *pid-file*.

When *initialize?* is true, automatically create the seed and host key upon service activation if they do not exist yet. This may take long and require interaction.

When *initialize?* is false, it is up to the user to initialize the randomness generator (see Section "lsh-make-seed" in *LSH Manual*), and to create a key pair with the private key stored in file *host-key* (see Section "lshd basics" in *LSH Manual*).

When *interfaces* is empty, lshd listens for connections on all the network interfaces; otherwise, *interfaces* must be a list of host names or addresses.

allow-empty-passwords? specifies whether to accept log-ins with empty passwords, and *root-login?* specifies whether to accept log-ins as root.

The other options should be self-descriptive.

`%facebook-host-aliases` [Scheme Variable]

This variable contains a string for use in `/etc/hosts` (see Section "Host Names" in *The GNU C Library Reference Manual*). Each line contains a entry that maps a known server name of the Facebook on-line service—e.g., `www.facebook.com`—to the local host—`127.0.0.1` or its IPv6 equivalent, `::1`.

This variable is typically used in the `hosts-file` field of an `operating-system` declaration (see Section 7.2.2 [operating-system Reference], page 89):

```
(use-modules (gnu) (guix))

(operating-system
  (host-name "mymachine")
  ;; ...
  (hosts-file
    ;; Create a /etc/hosts file with aliases for "localhost"
    ;; and "mymachine", as well as for Facebook servers.
    (plain-file "hosts"
                (string-append (local-host-aliases host-name)
                               %facebook-host-aliases))))
```

This mechanism can prevent programs running locally, such as Web browsers, from accessing Facebook.

The `(gnu services avahi)` provides the following definition.

`avahi-service` [*#:avahi* `avahi`] [*#:host-name #f*] [*#:publish? #t*] [*#:ipv4? #t*] [*#:ipv6? #t*] [*#:wide-area? #f*] [*#:domains-to-browse '()*] [Scheme Procedure]

Return a service that runs `avahi-daemon`, a system-wide mDNS/DNS-SD responder that allows for service discovery and "zero-configuration" host name lookups (see `http://avahi.org/`), and extends the name service cache daemon (nscd) so that it can resolve `.local` host names using nss-mdns. Additionally, add the *avahi* package to the system profile so that commands such as `avahi-browse` are directly usable.

If *host-name* is different from `#f`, use that as the host name to publish for this machine; otherwise, use the machine's actual host name.

When *publish?* is true, publishing of host names and services is allowed; in particular, avahi-daemon will publish the machine's host name and IP address via mDNS on the local network.

When *wide-area?* is true, DNS-SD over unicast DNS is enabled.

Boolean values *ipv4?* and *ipv6?* determine whether to use IPv4/IPv6 sockets.

7.2.7.3 X Window

Support for the X Window graphical display system—specifically Xorg—is provided by the `(gnu services xorg)` module. Note that there is no `xorg-service` procedure. Instead, the X server is started by the *login manager*, currently SLiM.

`slim-service` [*#:allow-empty-passwords? #f*] [*#:auto-login? #f*] [*#:default-user* `""`] [*#:startx*] [*#:theme* `%default-slim-theme`] [*#:theme-name* `%default-slim-theme-name`] [Scheme Procedure]

Return a service that spawns the SLiM graphical login manager, which in turn starts the X display server with *startx*, a command as returned by `xorg-start-command`.

SLiM automatically looks for session types described by the `.desktop` files in `/run/current-system/profile/share/xsessions` and allows users to choose a session from the log-in screen using `F1`. Packages such as *xfce*, *sawfish*, and

ratpoison provide `.desktop` files; adding them to the system-wide set of packages automatically makes them available at the log-in screen.

In addition, `~/.xsession` files are honored. When available, `~/.xsession` must be an executable that starts a window manager and/or other X clients.

When *allow-empty-passwords?* is true, allow logins with an empty password. When *auto-login?* is true, log in automatically as *default-user*.

If *theme* is `#f`, the use the default log-in theme; otherwise *theme* must be a gexp denoting the name of a directory containing the theme to use. In that case, *theme-name* specifies the name of the theme.

`%default-theme` [Scheme Variable]
`%default-theme-name` [Scheme Variable]

The G-Expression denoting the default SLiM theme and its name.

`xorg-start-command` [*#:guile*] [*#:configuration-file #f*] [*#:xorg-server* ***xorg-server***] [Scheme Procedure]

Return a derivation that builds a *guile* script to start the X server from *xorg-server*. *configuration-file* is the server configuration file or a derivation that builds it; when omitted, the result of `xorg-configuration-file` is used.

Usually the X server is started by a login manager.

`xorg-configuration-file` [*#:drivers '()*] [*#:resolutions '()*] [*#:extra-config '()*] [Scheme Procedure]

Return a configuration file for the Xorg server containing search paths for all the common drivers.

drivers must be either the empty list, in which case Xorg chooses a graphics driver automatically, or a list of driver names that will be tried in this order—e.g., `(\"modesetting\" \"vesa\")`.

Likewise, when *resolutions* is the empty list, Xorg chooses an appropriate screen resolution; otherwise, it must be a list of resolutions—e.g., `((1024 768) (640 480))`.

Last, *extra-config* is a list of strings or objects appended to the `text-file*` argument list. It is used to pass extra text to be added verbatim to the configuration file.

`screen-locker-service` *package* [*name*] [Scheme Procedure]

Add *package*, a package for a screen-locker or screen-saver whose command is *program*, to the set of setuid programs and add a PAM entry for it. For example:

```
(screen-locker-service xlockmore "xlock")
```

makes the good ol' XlockMore usable.

7.2.7.4 Desktop Services

The `(gnu services desktop)` module provides services that are usually useful in the context of a "desktop" setup—that is, on a machine running a graphical display server, possibly with graphical user interfaces, etc.

To simplify things, the module defines a variable containing the set of services that users typically expect on a machine with a graphical environment and networking:

`%desktop-services` [Scheme Variable]

This is a list of services that builds upon *%base-services* and adds or adjust services for a typical "desktop" setup.

In particular, it adds a graphical login manager (see Section 7.2.7.3 [X Window], page 103), screen lockers, a network management tool (see Section 7.2.7.2 [Networking Services], page 101), energy and color management services, the `elogind` login and seat manager, the Polkit privilege service, the GeoClue location service, an NTP client (see Section 7.2.7.2 [Networking Services], page 101), the Avahi daemon, and has the name service switch service configured to be able to use `nss-mdns` (see Section 7.2.10 [Name Service Switch], page 110).

The *%desktop-services* variable can be used as the `services` field of an `operating-system` declaration (see Section 7.2.2 [operating-system Reference], page 89).

The actual service definitions provided by `(gnu services dbus)` and `(gnu services desktop)` are described below.

`dbus-service` [*#:dbus* `dbus`] [*#:services* '()] [Scheme Procedure]

Return a service that runs the "system bus", using *dbus*, with support for *services*.

D-Bus is an inter-process communication facility. Its system bus is used to allow system services to communicate and be notified of system-wide events.

services must be a list of packages that provide an `etc/dbus-1/system.d` directory containing additional D-Bus configuration and policy files. For example, to allow avahi-daemon to use the system bus, *services* must be equal to `(list avahi)`.

`elogind-service` [*#:config* `config`] [Scheme Procedure]

Return a service that runs the `elogind` login and seat management daemon. Elogind exposes a D-Bus interface that can be used to know which users are logged in, know what kind of sessions they have open, suspend the system, inhibit system suspend, reboot the system, and other tasks.

Elogind handles most system-level power events for a computer, for example suspending the system when a lid is closed, or shutting it down when the power button is pressed.

The *config* keyword argument specifies the configuration for elogind, and should be the result of a `(elogind-configuration (`*parameter value*`)...)` invocation. Available parameters and their default values are:

```
kill-user-processes?
          #f

kill-only-users
          ()

kill-exclude-users
          ("root")

inhibit-delay-max-seconds
          5

handle-power-key
          poweroff
```

```
handle-suspend-key
          suspend
handle-hibernate-key
          hibernate
handle-lid-switch
          suspend
handle-lid-switch-docked
          ignore
power-key-ignore-inhibited?
          #f
suspend-key-ignore-inhibited?
          #f
hibernate-key-ignore-inhibited?
          #f
lid-switch-ignore-inhibited?
          #t
holdoff-timeout-seconds
          30
idle-action
          ignore
idle-action-seconds
          (* 30 60)
runtime-directory-size-percent
          10
runtime-directory-size
          #f
remove-ipc?
          #t
suspend-state
          ("mem" "standby" "freeze")
suspend-mode
          ()
hibernate-state
          ("disk")
hibernate-mode
          ("platform" "shutdown")
hybrid-sleep-state
          ("disk")
hybrid-sleep-mode
          ("suspend" "platform" "shutdown")
```

`polkit-service` [*#:polkit* `polkit`] [Scheme Procedure]
Return a service that runs the Polkit privilege management service, which allows system administrators to grant access to privileged operations in a structured way. By querying the Polkit service, a privileged system component can know when it should grant additional capabilities to ordinary users. For example, an ordinary user can be granted the capability to suspend the system if the user is logged in locally.

`upower-service` [*#:upower* `upower`] [*#:watts-up-pro? #f*] [Scheme Procedure]
[*#:poll-batteries? #t*] [*#:ignore-lid? #f*] [*#:use-percentage-for-policy? #f*]
[*#:percentage-low 10*] [*#:percentage-critical 3*] [*#:percentage-action 2*]
[*#:time-low 1200*] [*#:time-critical 300*] [*#:time-action 120*]
[*#:critical-power-action 'hybrid-sleep*]
Return a service that runs `upowerd`, a system-wide monitor for power consumption and battery levels, with the given configuration settings. It implements the `org.freedesktop.UPower` D-Bus interface, and is notably used by GNOME.

`udisks-service` [*#:udisks* `udisks`] [Scheme Procedure]
Return a service for UDisks, a *disk management* daemon that provides user interfaces with notifications and ways to mount/unmount disks. Programs that talk to UDisks include the `udisksctl` command, part of UDisks, and GNOME Disks.

`colord-service` [*#:colord* `colord`] [Scheme Procedure]
Return a service that runs `colord`, a system service with a D-Bus interface to manage the color profiles of input and output devices such as screens and scanners. It is notably used by the GNOME Color Manager graphical tool. See the colord web site for more information.

`geoclue-application` *name* [*#:allowed? #t*] [*#:system? #f*] [Scheme Procedure]
[*#:users '()*]
Return an configuration allowing an application to access GeoClue location data. *name* is the Desktop ID of the application, without the `.desktop` part. If *allowed?* is true, the application will have access to location information by default. The boolean *system?* value indicates that an application is a system component or not. Finally *users* is a list of UIDs of all users for which this application is allowed location info access. An empty users list means that all users are allowed.

`%standard-geoclue-applications` [Scheme Variable]
The standard list of well-known GeoClue application configurations, granting authority to GNOME's date-and-time utility to ask for the current location in order to set the time zone, and allowing the Firefox (IceCat) and Epiphany web browsers to request location information. Firefox and Epiphany both query the user before allowing a web page to know the user's location.

`geoclue-service` [*#:colord* `colord`] [*#:whitelist '()*] [Scheme Procedure]
[*#:wifi-geolocation-url*
"*https://location.services.mozilla.com/v1/geolocate?key=geoclue*"]
[*#:submit-data? #f*]
[#:wifi-submission-url "https://location.services.mozilla.com/v1/submit?key=geoclue"]
[#:submission-nick "geoclue"] [#:applications %standard-geoclue-applications]

Return a service that runs the GeoClue location service. This service provides a D-Bus interface to allow applications to request access to a user's physical location, and optionally to add information to online location databases. See the GeoClue web site for more information.

7.2.7.5 Database Services

The `(gnu services databases)` module provides the following service.

`postgresql-service` [*#:postgresql postgresql*] [*#:config-file*] [*#:data-directory "/var/lib/postgresql/data"*] [Scheme Procedure]

Return a service that runs *postgresql*, the PostgreSQL database server.

The PostgreSQL daemon loads its runtime configuration from *config-file* and stores the database cluster in *data-directory*.

7.2.7.6 Web Services

The `(gnu services web)` module provides the following service:

`nginx-service` [*#:nginx nginx*] [*#:log-directory "/var/log/nginx"*] [*#:run-directory "/var/run/nginx"*] [*#:config-file*] [Scheme Procedure]

Return a service that runs *nginx*, the nginx web server.

The nginx daemon loads its runtime configuration from *config-file*. Log files are written to *log-directory* and temporary runtime data files are written to *run-directory*. For proper operation, these arguments should match what is in *config-file* to ensure that the directories are created when the service is activated.

7.2.7.7 Various Services

The `(gnu services lirc)` module provides the following service.

`lirc-service` [*#:lirc lirc*] [*#:device #f*] [*#:driver #f*] [*#:config-file #f*] [*#:extra-options '()*] [Scheme Procedure]

Return a service that runs LIRC, a daemon that decodes infrared signals from remote controls.

Optionally, *device*, *driver* and *config-file* (configuration file name) may be specified. See `lircd` manual for details.

Finally, *extra-options* is a list of additional command-line options passed to `lircd`.

7.2.8 Setuid Programs

Some programs need to run with "root" privileges, even when they are launched by unprivileged users. A notorious example is the `passwd` program, which users can run to change their password, and which needs to access the `/etc/passwd` and `/etc/shadow` files—something normally restricted to root, for obvious security reasons. To address that, these executables are *setuid-root*, meaning that they always run with root privileges (see Section "How Change Persona" in *The GNU C Library Reference Manual*, for more info about the setuid mechanisms.)

The store itself *cannot* contain setuid programs: that would be a security issue since any user on the system can write derivations that populate the store (see Section 5.3 [The Store],

page 46). Thus, a different mechanism is used: instead of changing the setuid bit directly on files that are in the store, we let the system administrator *declare* which programs should be setuid root.

The `setuid-programs` field of an `operating-system` declaration contains a list of G-expressions denoting the names of programs to be setuid-root (see Section 7.2.1 [Using the Configuration System], page 85). For instance, the `passwd` program, which is part of the Shadow package, can be designated by this G-expression (see Section 5.6 [G-Expressions], page 53):

```
#~(string-append #$shadow "/bin/passwd")
```

A default set of setuid programs is defined by the `%setuid-programs` variable of the `(gnu system)` module.

`%setuid-programs` [Scheme Variable]

A list of G-expressions denoting common programs that are setuid-root.

The list includes commands such as `passwd`, `ping`, `su`, and `sudo`.

Under the hood, the actual setuid programs are created in the `/run/setuid-programs` directory at system activation time. The files in this directory refer to the "real" binaries, which are in the store.

7.2.9 X.509 Certificates

Web servers available over HTTPS (that is, HTTP over the transport-layer security mechanism, TLS) send client programs an *X.509 certificate* that the client can then use to *authenticate* the server. To do that, clients verify that the server's certificate is signed by a so-called *certificate authority* (CA). But to verify the CA's signature, clients must have first acquired the CA's certificate.

Web browsers such as GNU IceCat include their own set of CA certificates, such that they are able to verify CA signatures out-of-the-box.

However, most other programs that can talk HTTPS—`wget`, `git`, `w3m`, etc.—need to be told where CA certificates can be found.

In GuixSD, this is done by adding a package that provides certificates to the `packages` field of the `operating-system` declaration (see Section 7.2.2 [operating-system Reference], page 89). GuixSD includes one such package, `nss-certs`, which is a set of CA certificates provided as part of Mozilla's Network Security Services.

Note that it is *not* part of *%base-packages*, so you need to explicitly add it. The `/etc/ssl/certs` directory, which is where most applications and libraries look for certificates by default, points to the certificates installed globally.

Unprivileged users can also install their own certificate package in their profile. A number of environment variables need to be defined so that applications and libraries know where to find them. Namely, the OpenSSL library honors the `SSL_CERT_DIR` and `SSL_CERT_FILE` variables. Some applications add their own environment variables; for instance, the Git version control system honors the certificate bundle pointed to by the `GIT_SSL_CAINFO` environment variable.

7.2.10 Name Service Switch

The `(gnu system nss)` module provides bindings to the configuration file of libc's *name service switch* or *NSS* (see Section "NSS Configuration File" in *The GNU C Library Reference Manual*). In a nutshell, the NSS is a mechanism that allows libc to be extended with new "name" lookup methods for system databases, which includes host names, service names, user accounts, and more (see Section "Name Service Switch" in *The GNU C Library Reference Manual*).

The NSS configuration specifies, for each system database, which lookup method is to be used, and how the various methods are chained together—for instance, under which circumstances NSS should try the next method in the list. The NSS configuration is given in the `name-service-switch` field of `operating-system` declarations (see Section 7.2.2 [operating-system Reference], page 89).

As an example, the declaration below configures the NSS to use the `nss-mdns` back-end, which supports host name lookups over multicast DNS (mDNS) for host names ending in `.local`:

```
(name-service-switch
   (hosts (list %files    ;first, check /etc/hosts

                ;; If the above did not succeed, try
                ;; with 'mdns_minimal'.
                (name-service
                  (name "mdns_minimal")

                  ;; 'mdns_minimal' is authoritative for
                  ;; '.local'.  When it returns "not found",
                  ;; no need to try the next methods.
                  (reaction (lookup-specification
                             (not-found => return))))

                ;; Then fall back to DNS.
                (name-service
                  (name "dns"))

                ;; Finally, try with the "full" 'mdns'.
                (name-service
                  (name "mdns")))))
```

Don't worry: the `%mdns-host-lookup-nss` variable (see below) contains this configuration, so you won't have to type it if all you want is to have `.local` host lookup working.

Note that, in this case, in addition to setting the `name-service-switch` of the `operating-system` declaration, you also need to use `avahi-service` (see Section 7.2.7.2 [Networking Services], page 101), or *%desktop-services*, which includes it (see Section 7.2.7.4 [Desktop Services], page 104). Doing this makes `nss-mdns` accessible to the name service cache daemon (see Section 7.2.7.1 [Base Services], page 98).

For convenience, the following variables provide typical NSS configurations.

`%default-nss` [Scheme Variable]
This is the default name service switch configuration, a `name-service-switch` object.

`%mdns-host-lookup-nss` [Scheme Variable]
This is the name service switch configuration with support for host name lookup over multicast DNS (mDNS) for host names ending in `.local`.

The reference for name service switch configuration is given below. It is a direct mapping of the C library's configuration file format, so please refer to the C library manual for more information (see Section "NSS Configuration File" in *The GNU C Library Reference Manual*). Compared to libc's NSS configuration file format, it has the advantage not only of adding this warm parenthetic feel that we like, but also static checks: you'll know about syntax errors and typos as soon as you run `guix system`.

`name-service-switch` [Data Type]
This is the data type representation the configuration of libc's name service switch (NSS). Each field below represents one of the supported system databases.

`aliases`
`ethers`
`group`
`gshadow`
`hosts`
`initgroups`
`netgroup`
`networks`
`password`
`public-key`
`rpc`
`services`
`shadow` The system databases handled by the NSS. Each of these fields must be a list of `<name-service>` objects (see below.)

`name-service` [Data Type]
This is the data type representing an actual name service and the associated lookup action.

`name` A string denoting the name service (see Section "Services in the NSS configuration" in *The GNU C Library Reference Manual*).

Note that name services listed here must be visible to nscd. This is achieved by passing the `#:name-services` argument to `nscd-service` the list of packages providing the needed name services (see Section 7.2.7.1 [Base Services], page 98).

`reaction` An action specified using the `lookup-specification` macro (see Section "Actions in the NSS configuration" in *The GNU C Library Reference Manual*). For example:

```
(lookup-specification (unavailable => continue)
                      (success => return))
```

7.2.11 Initial RAM Disk

For bootstrapping purposes, the Linux-Libre kernel is passed an *initial RAM disk*, or *initrd*. An initrd contains a temporary root file system, as well as an initialization script. The latter is responsible for mounting the real root file system, and for loading any kernel modules that may be needed to achieve that.

The `initrd` field of an `operating-system` declaration allows you to specify which initrd you would like to use. The `(gnu system linux-initrd)` module provides two ways to build an initrd: the high-level `base-initrd` procedure, and the low-level `expression->initrd` procedure.

The `base-initrd` procedure is intended to cover most common uses. For example, if you want to add a bunch of kernel modules to be loaded at boot time, you can define the `initrd` field of the operating system declaration like this:

```
(initrd (lambda (file-systems . rest)
          ;; Create a standard initrd that has modules "foo.ko"
          ;; and "bar.ko", as well as their dependencies, in
          ;; addition to the modules available by default.
          (apply base-initrd file-systems
                 #:extra-modules '("foo" "bar")
                 rest)))
```

The `base-initrd` procedure also handles common use cases that involves using the system as a QEMU guest, or as a "live" system whose root file system is volatile.

`base-initrd` *file-systems* [*#:qemu-networking? #f*] [*#:virtio? #f*] [*#:volatile-root? #f*] [*#:extra-modules '()*] [*#:mapped-devices '()*] [Monadic Procedure]

Return a monadic derivation that builds a generic initrd. *file-systems* is a list of file-systems to be mounted by the initrd, possibly in addition to the root file system specified on the kernel command line via `--root`. *mapped-devices* is a list of device mappings to realize before *file-systems* are mounted (see Section 7.2.4 [Mapped Devices], page 93).

When *qemu-networking?* is true, set up networking with the standard QEMU parameters. When *virtio?* is true, load additional modules so the initrd can be used as a QEMU guest with para-virtualized I/O drivers.

When *volatile-root?* is true, the root file system is writable but any changes to it are lost.

The initrd is automatically populated with all the kernel modules necessary for *file-systems* and for the given options. However, additional kernel modules can be listed in *extra-modules*. They will be added to the initrd, and loaded at boot time in the order in which they appear.

Needless to say, the initrds we produce and use embed a statically-linked Guile, and the initialization program is a Guile program. That gives a lot of flexibility. The `expression->initrd` procedure builds such an initrd, given the program to run in that initrd.

`expression->initrd` *exp* [*#:guile %guile-static-stripped*] [Monadic Procedure]
[*#:name* "*guile-initrd*"] [*#:modules '()*]

Return a derivation that builds a Linux initrd (a gzipped cpio archive) containing *guile* and that evaluates *exp*, a G-expression, upon booting. All the derivations referenced by *exp* are automatically copied to the initrd.

modules is a list of Guile module names to be embedded in the initrd.

7.2.12 GRUB Configuration

The operating system uses GNU GRUB as its boot loader (see Section "Overview" in *GNU GRUB Manual*). It is configured using `grub-configuration` declarations. This data type is exported by the `(gnu system grub)` module, and described below.

`grub-configuration` [Data Type]

The type of a GRUB configuration declaration.

`device` This is a string denoting the boot device. It must be a device name understood by the `grub-install` command, such as `/dev/sda` or `(hd0)` (see Section "Invoking grub-install" in *GNU GRUB Manual*).

`menu-entries` (default: `()`)
A possibly empty list of `menu-entry` objects (see below), denoting entries to appear in the GRUB boot menu, in addition to the current system entry and the entry pointing to previous system generations.

`default-entry` (default: `0`)
The index of the default boot menu entry. Index 0 is for the current system's entry.

`timeout` (default: `5`)
The number of seconds to wait for keyboard input before booting. Set to 0 to boot immediately, and to -1 to wait indefinitely.

`theme` (default: *%default-theme*)
The `grub-theme` object describing the theme to use.

Should you want to list additional boot menu entries *via* the `menu-entries` field above, you will need to create them with the `menu-entry` form:

`menu-entry` [Data Type]

The type of an entry in the GRUB boot menu.

`label` The label to show in the menu—e.g., `"GNU"`.

`linux` The Linux kernel to boot.

`linux-arguments` (default: `()`)
The list of extra Linux kernel command-line arguments—e.g., `("console=ttyS0")`.

`initrd` A G-Expression or string denoting the file name of the initial RAM disk to use (see Section 5.6 [G-Expressions], page 53).

Themes are created using the `grub-theme` form, which is not documented yet.

`%default-theme` [Scheme Variable]

This is the default GRUB theme used by the operating system, with a fancy background image displaying the GNU and Guix logos.

7.2.13 Invoking `guix system`

Once you have written an operating system declaration, as seen in the previous section, it can be *instantiated* using the `guix system` command. The synopsis is:

```
guix system options... action file
```

file must be the name of a file containing an `operating-system` declaration. *action* specifies how the operating system is instantiate. Currently the following values are supported:

`reconfigure`

Build the operating system described in *file*, activate it, and switch to it[8].

This effects all the configuration specified in *file*: user accounts, system services, global package list, setuid programs, etc.

It also adds a GRUB menu entry for the new OS configuration, and moves entries for older configurations to a submenu—unless `--no-grub` is passed.

It is highly recommended to run `guix pull` once before you run `guix system reconfigure` for the first time (see Section 3.6 [Invoking guix pull], page 24). Failing to do that you would see an older version of Guix once `reconfigure` has completed.

`build` Build the operating system's derivation, which includes all the configuration files and programs needed to boot and run the system. This action does not actually install anything.

`init` Populate the given directory with all the files necessary to run the operating system specified in *file*. This is useful for first-time installations of GuixSD. For instance:

```
guix system init my-os-config.scm /mnt
```

copies to `/mnt` all the store items required by the configuration specified in `my-os-config.scm`. This includes configuration files, packages, and so on. It also creates other essential files needed for the system to operate correctly—e.g., the `/etc`, `/var`, and `/run` directories, and the `/bin/sh` file.

This command also installs GRUB on the device specified in `my-os-config`, unless the `--no-grub` option was passed.

`vm` Build a virtual machine that contain the operating system declared in *file*, and return a script to run that virtual machine (VM). Arguments given to the script are passed as is to QEMU.

The VM shares its store with the host system.

Additional file systems can be shared between the host and the VM using the `--share` and `--expose` command-line options: the former specifies a directory

[8] This action is usable only on systems already running GuixSD.

to be shared with write access, while the latter provides read-only access to the shared directory.

The example below creates a VM in which the user's home directory is accessible read-only, and where the `/exchange` directory is a read-write mapping of the host's `$HOME/tmp`:

```
guix system vm my-config.scm \
   --expose=$HOME --share=$HOME/tmp=/exchange
```

On GNU/Linux, the default is to boot directly to the kernel; this has the advantage of requiring only a very tiny root disk image since the host's store can then be mounted.

The `--full-boot` option forces a complete boot sequence, starting with the bootloader. This requires more disk space since a root image containing at least the kernel, initrd, and bootloader data files must be created. The `--image-size` option can be used to specify the image's size.

`vm-image`
`disk-image`

Return a virtual machine or disk image of the operating system declared in *file* that stands alone. Use the `--image-size` option to specify the size of the image.

When using `vm-image`, the returned image is in qcow2 format, which the QEMU emulator can efficiently use.

When using `disk-image`, a raw disk image is produced; it can be copied as is to a USB stick, for instance. Assuming `/dev/sdc` is the device corresponding to a USB stick, one can copy the image on it using the following command:

```
# dd if=$(guix system disk-image my-os.scm) of=/dev/sdc
```

`container`

Return a script to run the operating system declared in *file* within a container. Containers are a set of lightweight isolation mechanisms provided by the kernel Linux-libre. Containers are substantially less resource-demanding than full virtual machines since the kernel, shared objects, and other resources can be shared with the host system; this also means they provide thinner isolation.

Currently, the script must be run as root in order to support more than a single user and group. The container shares its store with the host system.

As with the `vm` action (see [guix system vm], page 114), additional file systems to be shared between the host and container can be specified using the `--share` and `--expose` options:

```
guix system container my-config.scm \
   --expose=$HOME --share=$HOME/tmp=/exchange
```

Note: This option requires Linux-libre 3.19 or newer.

options can contain any of the common build options provided by `guix build` (see Section 6.1 [Invoking guix build], page 60). In addition, *options* can contain one of the following:

`--system=`*system*
`-s` *system* Attempt to build for *system* instead of the host's system type. This works as per `guix build` (see Section 6.1 [Invoking guix build], page 60).

`--derivation`
`-d` Return the derivation file name of the given operating system without building anything.

`--image-size=`*size*
For the `vm-image` and `disk-image` actions, create an image of the given *size*. *size* may be a number of bytes, or it may include a unit as a suffix (see Section "Block size" in *GNU Coreutils*).

`--on-error=`*strategy*
Apply *strategy* when an error occurs when reading *file*. *strategy* may be one of the following:

`nothing-special`
Report the error concisely and exit. This is the default strategy.

`backtrace`
Likewise, but also display a backtrace.

`debug` Report the error and enter Guile's debugger. From there, you can run commands such as `,bt` to get a backtrace, `,locals` to display local variable values, and more generally inspect the program's state. See Section "Debug Commands" in *GNU Guile Reference Manual*, for a list of available debugging commands.

Note that all the actions above, except `build` and `init`, rely on KVM support in the Linux-Libre kernel. Specifically, the machine should have hardware virtualization support, the corresponding KVM kernel module should be loaded, and the `/dev/kvm` device node must exist and be readable and writable by the user and by the daemon's build users.

Once you have built, configured, re-configured, and re-re-configured your GuixSD installation, you may find it useful to list the operating system generations available on disk—and that you can choose from the GRUB boot menu:

`list-generations`
List a summary of each generation of the operating system available on disk, in a human-readable way. This is similar to the `--list-generations` option of `guix package` (see Section 3.2 [Invoking guix package], page 14).

Optionally, one can specify a pattern, with the same syntax that is used in `guix package --list-generations`, to restrict the list of generations displayed. For instance, the following command displays generations up to 10-day old:

```
$ guix system list-generations 10d
```

The `guix system` command has even more to offer! The following sub-commands allow you to visualize how your system services relate to each other:

`extension-graph`
Emit in Dot/Graphviz format to standard output the *service extension graph* of the operating system defined in *file* (see Section 7.2.14.1 [Service Composition], page 117, for more information on service extensions.)

The command:

```
$ guix system extension-graph file | dot -Tpdf > services.pdf
```

produces a PDF file showing the extension relations among services.

`dmd-graph`
: Emit in Dot/Graphviz format to standard output the *dependency graph* of dmd services of the operating system defined in *file*. See Section 7.2.14.4 [dmd Services], page 123, for more information and for an example graph.

7.2.14 Defining Services

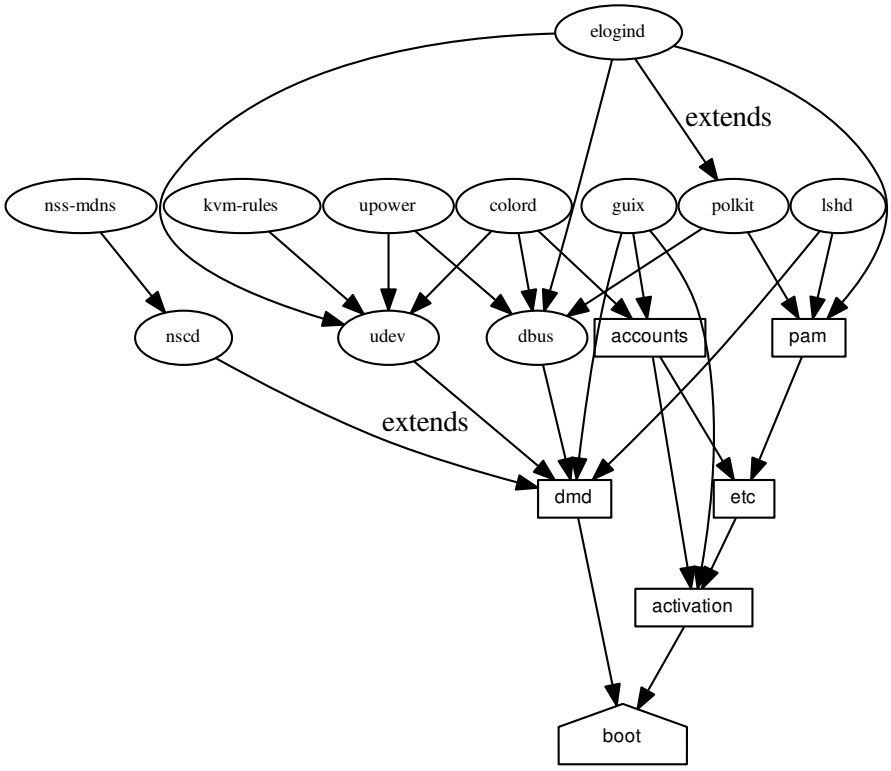
The previous sections show the available services and how one can combine them in an `operating-system` declaration. But how do we define them in the first place? And what is a service anyway?

7.2.14.1 Service Composition

Here we define a *service* as, broadly, something that extends the operating system's functionality. Often a service is a process—a *daemon*—started when the system boots: a secure shell server, a Web server, the Guix build daemon, etc. Sometimes a service is a daemon whose execution can be triggered by another daemon—e.g., an FTP server started by `inetd` or a D-Bus service activated by `dbus-daemon`. Occasionally, a service does not map to a daemon. For instance, the "account" service collects user accounts and makes sure they exist when the system runs; the "udev" service collects device management rules and makes them available to the eudev daemon; the `/etc` service populates the system's `/etc` directory.

GuixSD services are connected by *extensions*. For instance, the secure shell service *extends* dmd—GuixSD's initialization system, running as PID 1—by giving it the command lines to start and stop the secure shell daemon (see Section 7.2.7.2 [Networking Services], page 101); the UPower service extends the D-Bus service by passing it its `.service` specification, and extends the udev service by passing it device management rules (see Section 7.2.7.4 [Desktop Services], page 104); the Guix daemon service extends dmd by passing it the command lines to start and stop the daemon, and extends the account service by passing it a list of required build user accounts (see Section 7.2.7.1 [Base Services], page 98).

All in all, services and their “extends” relations form a directed acyclic graph (DAG). If we represent services as boxes and extensions as arrows, a typical system might provide something like this:



At the bottom, we see the *system service*, which produces the directory containing everything to run and boot the system, as returned by the `guix system build` command. See Section 7.2.14.3 [Service Reference], page 120, to learn about the other service types shown here. See [system-extension-graph], page 116, for information on how to generate this representation for a particular operating system definition.

Technically, developers can define *service types* to express these relations. There can be any number of services of a given type on the system—for instance, a system running two instances of the GNU secure shell server (lsh) has two instances of *lsh-service-type*, with different parameters.

The following section describes the programming interface for service types and services.

7.2.14.2 Service Types and Services

A *service type* is a node in the DAG described above. Let us start with a simple example, the service type for the Guix build daemon (see Section 2.5 [Invoking guix-daemon], page 8):

```
(define guix-service-type
  (service-type
   (name 'guix)
   (extensions
    (list (service-extension dmd-root-service-type guix-dmd-service)
          (service-extension account-service-type guix-accounts)
          (service-extension activation-service-type guix-activation)))))
```

It defines a two things:

1. A name, whose sole purpose is to make inspection and debugging easier.
2. A list of *service extensions*, where each extension designates the target service type and a procedure that, given the service's parameters, returns a list of object to extend the service of that type.

   Every service type has at least one service extension. The only exception is the *boot service type*, which is the ultimate service.

In this example, *guix-service-type* extends three services:

dmd-root-service-type
: The *guix-dmd-service* procedure defines how the dmd service is extended. Namely, it returns a `<dmd-service>` object that defines how `guix-daemon` is started and stopped (see Section 7.2.14.4 [dmd Services], page 123).

account-service-type
: This extension for this service is computed by *guix-accounts*, which returns a list of `user-group` and `user-account` objects representing the build user accounts (see Section 2.5 [Invoking guix-daemon], page 8).

activation-service-type
: Here *guix-activation* is a procedure that returns a gexp, which is a code snippet to run at "activation time"—e.g., when the service is booted.

A service of this type is instantiated like this:

```
(service guix-service-type
         (guix-configuration
           (build-accounts 5)
           (use-substitutes? #f)))
```

The second argument to the `service` form is a value representing the parameters of this specific service instance. See [guix-configuration-type], page 100, for information about the `guix-configuration` data type.

guix-service-type is quite simple because it extends other services but is not extensible itself.

The service type for an *extensible* service looks like this:

```
(define udev-service-type
  (service-type (name 'udev)
                (extensions
                 (list (service-extension dmd-root-service-type
                                          udev-dmd-service)))
```

```
(compose concatenate)       ;concatenate the list of rules
(extend (lambda (config rules)
          (match config
            (($ <udev-configuration> udev initial-rules)
             (udev-configuration
              (udev udev)   ;the udev package to use
              (rules (append initial-rules rules)))))))))
```

This is the service type for the eudev device management daemon. Compared to the previous example, in addition to an extension of *dmd-root-service-type*, we see two new fields:

`compose` This is the procedure to *compose* the list of extensions to services of this type.

Services can extend the udev service by passing it lists of rules; we compose those extensions simply by concatenating them.

`extend` This procedure defines how the service's value is *extended* with the composition of the extensions.

Udev extensions are composed into a list of rules, but the udev service value is itself a `<udev-configuration>` record. So here, we extend that record by appending the list of rules is contains to the list of contributed rules.

There can be only one instance of an extensible service type such as *udev-service-type*. If there were more, the `service-extension` specifications would be ambiguous.

Still here? The next section provides a reference of the programming interface for services.

7.2.14.3 Service Reference

We have seen an overview of service types (see Section 7.2.14.2 [Service Types and Services], page 118). This section provides a reference on how to manipulate services and service types. This interface is provided by the `(gnu services)` module.

`service` *type value* [Scheme Procedure]
Return a new service of *type*, a `<service-type>` object (see below.) *value* can be any object; it represents the parameters of this particular service instance.

`service?` *obj* [Scheme Procedure]
Return true if *obj* is a service.

`service-kind` *service* [Scheme Procedure]
Return the type of *service*—i.e., a `<service-type>` object.

`service-parameters` *service* [Scheme Procedure]
Return the value associated with *service*. It represents its parameters.

Here is an example of how a service is created and manipulated:

```
(define s
  (service nginx-service-type
           (nginx-configuration
            (nginx nginx)
```

```
                (log-directory log-directory)
                (run-directory run-directory)
                (file config-file))))

(service? s)
⇒ #t

(eq? (service-kind s) nginx-service-type)
⇒ #t
```

The `modify-services` form provides a handy way to change the parameters of some of the services of a list such as *%base-services* (see Section 7.2.7.1 [Base Services], page 98). Of course, you could always use standard list combinators such as `map` and `fold` to do that (see Section "SRFI-1" in *GNU Guile Reference Manual*); `modify-services` simply provides a more concise form for this common pattern.

`modify-services` *services* (*type variable* `=>` *body*) ... [Scheme Syntax]

Modify the services listed in *services* according to the given clauses. Each clause has the form:

```
(type variable => body)
```

where *type* is a service type, such as *guix-service-type*, and *variable* is an identifier that is bound within *body* to the value of the service of that *type*. See Section 7.2.1 [Using the Configuration System], page 85, for an example.

This is a shorthand for:

```
(map (lambda (service) ...) services)
```

Next comes the programming interface for service types. This is something you want to know when writing new service definitions, but not necessarily when simply looking for ways to customize your `operating-system` declaration.

`service-type` [Data Type]

This is the representation of a *service type* (see Section 7.2.14.2 [Service Types and Services], page 118).

`name` This is a symbol, used only to simplify inspection and debugging.

`extensions`
A non-empty list of `<service-extension>` objects (see below.)

`compose` (default: `#f`)
If this is `#f`, then the service type denotes services that cannot be extended—i.e., services that do not receive "values" from other services.

Otherwise, it must be a one-argument procedure. The procedure is called by `fold-services` and is passed a list of values collected from extensions. It must return a value that is a valid parameter value for the service instance.

`extend` (default: `#f`)
If this is `#f`, services of this type cannot be extended.

Otherwise, it must be a two-argument procedure: `fold-services` calls it, passing it the service's initial value as the first argument and the result of applying `compose` to the extension values as the second argument.

See Section 7.2.14.2 [Service Types and Services], page 118, for examples.

`service-extension` *target-type compute* [Scheme Procedure]
Return a new extension for services of type *target-type*. *compute* must be a one-argument procedure: `fold-services` calls it, passing it the value associated with the service that provides the extension; it must return a valid value for the target service.

`service-extension?` *obj* [Scheme Procedure]
Return true if *obj* is a service extension.

At the core of the service abstraction lies the `fold-services` procedure, which is responsible for "compiling" a list of services down to a single directory that contains everything needed to boot and run the system—the directory shown by the `guix system build` command (see Section 7.2.13 [Invoking guix system], page 114). In essence, it propagates service extensions down the service graph, updating each node parameters on the way, until it reaches the root node.

`fold-services` *services* [*#:target-type system-service-type*] [Scheme Procedure]
Fold *services* by propagating their extensions down to the root of type *target-type*; return the root service adjusted accordingly.

Lastly, the `(gnu services)` module also defines several essential service types, some of which are listed below.

`system-service-type` [Scheme Variable]
This is the root of the service graph. It produces the system directory as returned by the `guix system build` command.

`boot-service-type` [Scheme Variable]
The type of the "boot service", which produces the *boot script*. The boot script is what the initial RAM disk runs when booting.

`etc-service-type` [Scheme Variable]
The type of the `/etc` service. This service can be extended by passing it name/file tuples such as:

```
(list `("issue" ,(plain-file "issue" "Welcome!\n")))
```

In this example, the effect would be to add an `/etc/issue` file pointing to the given file.

`setuid-program-service-type` [Scheme Variable]
Type for the "setuid-program service". This service collects lists of executable file names, passed as gexps, and adds them to the set of setuid-root programs on the system (see Section 7.2.8 [Setuid Programs], page 108).

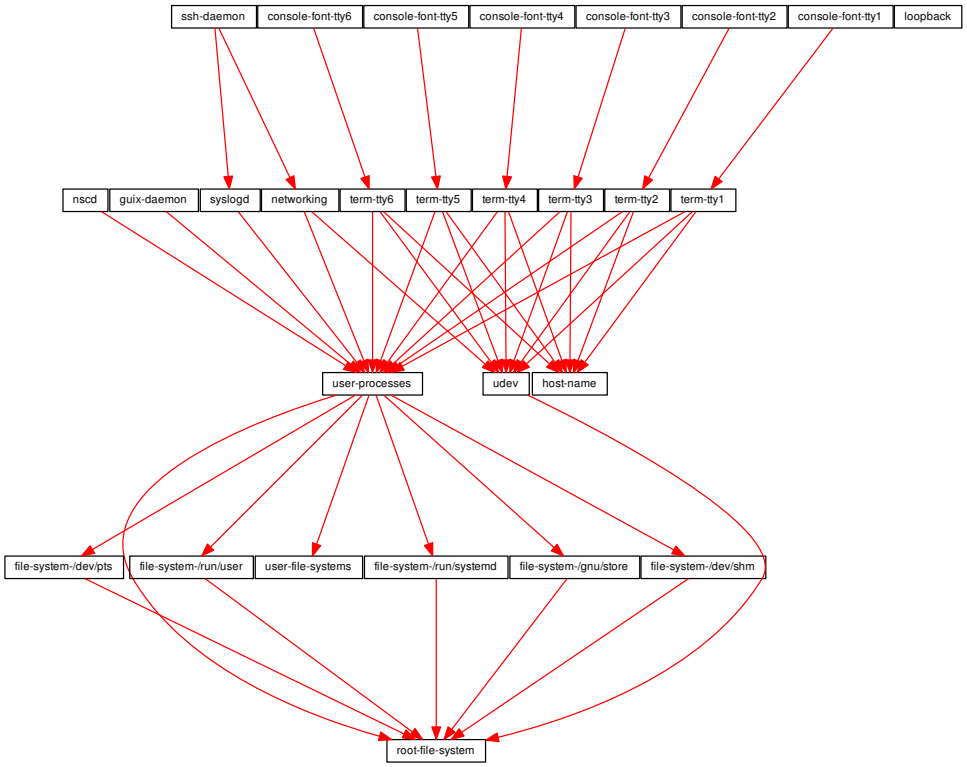
`profile-service-type` [Scheme Variable]

Type of the service that populates the *system profile*—i.e., the programs under `/run/current-system/profile`. Other services can extend it by passing it lists of packages to add to the system profile.

7.2.14.4 dmd Services

The `(gnu services dmd)` provides a way to define services managed by GNU dmd, which is GuixSD initialization system—the first process that is started when the system boots, aka. PID 1 (see Section "Introduction" in *GNU dmd Manual*).

Services in dmd can depend on each other. For instance, the SSH daemon may need to be started after the syslog daemon has been started, which in turn can only happen once all the file systems have been mounted. The simple operating system defined earlier (see Section 7.2.1 [Using the Configuration System], page 85) results in a service graph like this:



You can actually generate such a graph for any operating system definition using the `guix system dmd-graph` command (see [system-dmd-graph], page 117).

The *%dmd-root-service* is a service object representing PID 1, of type *dmd-root-service-type*; it can be extended by passing it lists of `<dmd-service>` objects.

`dmd-service` [Data Type]

The data type representing a service managed by dmd.

`provision`

This is a list of symbols denoting what the service provides.

These are the names that may be passed to `deco start`, `deco status`, and similar commands (see Section "Invoking deco" in *GNU dmd Manual*). See Section "Slots of services" in *GNU dmd Manual*, for details.

`requirements` (default: `'()`)

List of symbols denoting the dmd services this one depends on.

`respawn?` (default: `#t`)

Whether to restart the service when it stops, for instance when the underlying process dies.

`start`

`stop` (default: `#~(const #f)`)

The `start` and `stop` fields refer to dmd's facilities to start and stop processes (see Section "Service De- and Constructors" in *GNU dmd Manual*). They are given as G-expressions that get expanded in the dmd configuration file (see Section 5.6 [G-Expressions], page 53).

`documentation`

A documentation string, as shown when running:

```
deco doc service-name
```

where *service-name* is one of the symbols in *provision* (see Section "Invoking deco" in *GNU dmd Manual*).

`dmd-root-service-type` [Scheme Variable]

The service type for the dmd "root service"—i.e., PID 1.

This is the service type that extensions target when they want to create dmd services (see Section 7.2.14.2 [Service Types and Services], page 118, for an example). Each extension must pass a list of `<dmd-service>`.

`%dmd-root-service` [Scheme Variable]

This service represents PID 1.

7.3 Installing Debugging Files

Program binaries, as produced by the GCC compilers for instance, are typically written in the ELF format, with a section containing *debugging information*. Debugging information is what allows the debugger, GDB, to map binary code to source code; it is required to debug a compiled program in good conditions.

The problem with debugging information is that is takes up a fair amount of disk space. For example, debugging information for the GNU C Library weighs in at more than 60 MiB. Thus, as a user, keeping all the debugging info of all the installed programs is usually not an option. Yet, space savings should not come at the cost of an impediment to debugging—especially in the GNU system, which should make it easier for users to exert their computing freedom (see Chapter 7 [GNU Distribution], page 82).

Thankfully, the GNU Binary Utilities (Binutils) and GDB provide a mechanism that allows users to get the best of both worlds: debugging information can be stripped from the binaries and stored in separate files. GDB is then able to load debugging information from those files, when they are available (see Section "Separate Debug Files" in *Debugging with GDB*).

The GNU distribution takes advantage of this by storing debugging information in the `lib/debug` sub-directory of a separate package output unimaginatively called `debug` (see Section 3.4 [Packages with Multiple Outputs], page 21). Users can choose to install the `debug` output of a package when they need it. For instance, the following command installs the debugging information for the GNU C Library and for GNU Guile:

```
guix package -i glibc:debug guile:debug
```

GDB must then be told to look for debug files in the user's profile, by setting the `debug-file-directory` variable (consider setting it from the `~/.gdbinit` file, see Section "Startup" in *Debugging with GDB*):

```
(gdb) set debug-file-directory ~/.guix-profile/lib/debug
```

From there on, GDB will pick up debugging information from the `.debug` files under `~/.guix-profile/lib/debug`.

In addition, you will most likely want GDB to be able to show the source code being debugged. To do that, you will have to unpack the source code of the package of interest (obtained with `guix build --source`, see Section 6.1 [Invoking guix build], page 60), and to point GDB to that source directory using the `directory` command (see Section "Source Path" in *Debugging with GDB*).

The `debug` output mechanism in Guix is implemented by the `gnu-build-system` (see Section 5.2 [Build Systems], page 42). Currently, it is opt-in—debugging information is available only for those packages whose definition explicitly declares a `debug` output. This may be changed to opt-out in the future, if our build farm servers can handle the load. To check whether a package has a `debug` output, use `guix package --list-available` (see Section 3.2 [Invoking guix package], page 14).

7.4 Security Updates

Note: As of version 0.9.0, the feature described in this section is experimental.

Occasionally, important security vulnerabilities are discovered in core software packages and must be patched. Guix follows a functional package management discipline (see Chapter 1 [Introduction], page 2), which implies that, when a package is changed, *every package that depends on it* must be rebuilt. This can significantly slow down the deployment of fixes in core packages such as libc or Bash, since basically the whole distribution would need to be rebuilt. Using pre-built binaries helps (see Section 3.3 [Substitutes], page 20), but deployment may still take more time than desired.

To address that, Guix implements *grafts*, a mechanism that allows for fast deployment of critical updates without the costs associated with a whole-distribution rebuild. The idea is to rebuild only the package that needs to be patched, and then to "graft" it onto packages explicitly installed by the user and that were previously referring to the original package. The cost of grafting is typically very low, and order of magnitudes lower than a full rebuild of the dependency chain.

For instance, suppose a security update needs to be applied to Bash. Guix developers will provide a package definition for the "fixed" Bash, say *bash-fixed*, in the usual way (see Section 5.1 [Defining Packages], page 37). Then, the original package definition is augmented with a `replacement` field pointing to the package containing the bug fix:

```
(define bash
  (package
    (name "bash")
    ;; ...
    (replacement bash-fixed)))
```

From there on, any package depending directly or indirectly on Bash that is installed will automatically be "rewritten" to refer to *bash-fixed* instead of *bash*. This grafting process takes time proportional to the size of the package, but expect less than a minute for an "average" package on a recent machine.

Currently, the graft and the package it replaces (*bash-fixed* and *bash* in the example above) must have the exact same `name` and `version` fields. This restriction mostly comes from the fact that grafting works by patching files, including binary files, directly. Other restrictions may apply: for instance, when adding a graft to a package providing a shared library, the original shared library and its replacement must have the same `SONAME` and be binary-compatible.

7.5 Package Modules

From a programming viewpoint, the package definitions of the GNU distribution are provided by Guile modules in the `(gnu packages ...)` name space[9] (see Section "Modules" in *GNU Guile Reference Manual*). For instance, the `(gnu packages emacs)` module exports a variable named `emacs`, which is bound to a `<package>` object (see Section 5.1 [Defining Packages], page 37).

The `(gnu packages ...)` module name space is automatically scanned for packages by the command-line tools. For instance, when running `guix package -i emacs`, all the `(gnu packages ...)` modules are scanned until one that exports a package object whose name is `emacs` is found. This package search facility is implemented in the `(gnu packages)` module.

Users can store package definitions in modules with different names—e.g., `(my-packages emacs)`[10]. These package definitions will not be visible by default. Thus, users can invoke commands such as `guix package` and `guix build` have to be used with the `-e` option so that they know where to find the package. Better yet, they can use the `-L` option of these commands to make those modules visible (see Section 6.1 [Invoking guix build], page 60), or define the `GUIX_PACKAGE_PATH` environment variable. This environment variable makes it easy to extend or customize the distribution and is honored by all the user interfaces.

---

[9] Note that packages under the `(gnu packages ...)` module name space are not necessarily "GNU packages". This module naming scheme follows the usual Guile module naming convention: `gnu` means that these modules are distributed as part of the GNU system, and `packages` identifies modules that define packages.

[10] Note that the file name and module name must match. For instance, the `(my-packages emacs)` module must be stored in a `my-packages/emacs.scm` file relative to the load path specified with `--load-path` or `GUIX_PACKAGE_PATH`. See Section "Modules and the File System" in *GNU Guile Reference Manual*, for details.

`GUIX_PACKAGE_PATH` [Environment Variable]
: This is a colon-separated list of directories to search for package modules. Directories listed in this variable take precedence over the distribution's own modules.

The distribution is fully *bootstrapped* and *self-contained*: each package is built based solely on other packages in the distribution. The root of this dependency graph is a small set of *bootstrap binaries*, provided by the `(gnu packages bootstrap)` module. For more information on bootstrapping, see Section 7.7 [Bootstrapping], page 131.

7.6 Packaging Guidelines

The GNU distribution is nascent and may well lack some of your favorite packages. This section describes how you can help make the distribution grow. See Chapter 8 [Contributing], page 135, for additional information on how you can help.

Free software packages are usually distributed in the form of *source code tarballs*—typically `tar.gz` files that contain all the source files. Adding a package to the distribution means essentially two things: adding a *recipe* that describes how to build the package, including a list of other packages required to build it, and adding *package meta-data* along with that recipe, such as a description and licensing information.

In Guix all this information is embodied in *package definitions*. Package definitions provide a high-level view of the package. They are written using the syntax of the Scheme programming language; in fact, for each package we define a variable bound to the package definition, and export that variable from a module (see Section 7.5 [Package Modules], page 126). However, in-depth Scheme knowledge is *not* a prerequisite for creating packages. For more information on package definitions, see Section 5.1 [Defining Packages], page 37.

Once a package definition is in place, stored in a file in the Guix source tree, it can be tested using the `guix build` command (see Section 6.1 [Invoking guix build], page 60). For example, assuming the new package is called `gnew`, you may run this command from the Guix build tree (see Section 8.2 [Running Guix Before It Is Installed], page 135):

```
./pre-inst-env guix build gnew --keep-failed
```

Using `--keep-failed` makes it easier to debug build failures since it provides access to the failed build tree. Another useful command-line option when debugging is `--log-file`, to access the build log.

If the package is unknown to the `guix` command, it may be that the source file contains a syntax error, or lacks a `define-public` clause to export the package variable. To figure it out, you may load the module from Guile to get more information about the actual error:

```
./pre-inst-env guile -c '(use-modules (gnu packages gnew))'
```

Once your package builds correctly, please send us a patch (see Chapter 8 [Contributing], page 135). Well, if you need help, we will be happy to help you too. Once the patch is committed in the Guix repository, the new package automatically gets built on the supported platforms by our continuous integration system.

Users can obtain the new package definition simply by running `guix pull` (see Section 3.6 [Invoking guix pull], page 24). When `hydra.gnu.org` is done building the package, installing the package automatically downloads binaries from there (see Section 3.3 [Substitutes], page 20). The only place where human intervention is needed is to review and apply the patch.

7.6.1 Software Freedom

The GNU operating system has been developed so that users can have freedom in their computing. GNU is *free software*, meaning that users have the four essential freedoms: to run the program, to study and change the program in source code form, to redistribute exact copies, and to distribute modified versions. Packages found in the GNU distribution provide only software that conveys these four freedoms.

In addition, the GNU distribution follow the free software distribution guidelines. Among other things, these guidelines reject non-free firmware, recommendations of non-free software, and discuss ways to deal with trademarks and patents.

Some packages contain a small and optional subset that violates the above guidelines, for instance because this subset is itself non-free code. When that happens, the offending items are removed with appropriate patches or code snippets in the package definition's `origin` form (see Section 5.1 [Defining Packages], page 37). That way, `guix build --source` returns the "freed" source rather than the unmodified upstream source.

7.6.2 Package Naming

A package has actually two names associated with it: First, there is the name of the *Scheme variable*, the one following `define-public`. By this name, the package can be made known in the Scheme code, for instance as input to another package. Second, there is the string in the `name` field of a package definition. This name is used by package management commands such as `guix package` and `guix build`.

Both are usually the same and correspond to the lowercase conversion of the project name chosen upstream, with underscores replaced with hyphens. For instance, GNUnet is available as `gnunet`, and SDL_net as `sdl-net`.

We do not add `lib` prefixes for library packages, unless these are already part of the official project name. But see Section 7.6.5 [Python Modules], page 130 and Section 7.6.6 [Perl Modules], page 130 for special rules concerning modules for the Python and Perl languages.

Font package names are handled differently, see Section 7.6.7 [Fonts], page 130.

7.6.3 Version Numbers

We usually package only the latest version of a given free software project. But sometimes, for instance for incompatible library versions, two (or more) versions of the same package are needed. These require different Scheme variable names. We use the name as defined in Section 7.6.2 [Package Naming], page 128 for the most recent version; previous versions use the same name, suffixed by `-` and the smallest prefix of the version number that may distinguish the two versions.

The name inside the package definition is the same for all versions of a package and does not contain any version number.

For instance, the versions 2.24.20 and 3.9.12 of GTK+ may be packaged as follows:

```
(define-public gtk+
  (package
    (name "gtk+")
    (version "3.9.12")
```

```
    ...))
(define-public gtk+-2
  (package
    (name "gtk+")
    (version "2.24.20")
    ...))
```

If we also wanted GTK+ 3.8.2, this would be packaged as

```
(define-public gtk+-3.8
  (package
    (name "gtk+")
    (version "3.8.2")
    ...))
```

7.6.4 Synopses and Descriptions

As we have seen before, each package in GNU Guix includes a synopsis and a description (see Section 5.1 [Defining Packages], page 37). Synopses and descriptions are important: They are what `guix package --search` searches, and a crucial piece of information to help users determine whether a given package suits their needs. Consequently, packagers should pay attention to what goes into them.

Synopses must start with a capital letter and must not end with a period. They must not start with "a" or "the", which usually does not bring anything; for instance, prefer "File-frobbing tool" over "A tool that frobs files". The synopsis should say what the package is—e.g., "Core GNU utilities (file, text, shell)"—or what it is used for—e.g., the synopsis for GNU grep is "Print lines matching a pattern".

Keep in mind that the synopsis must be meaningful for a very wide audience. For example, "Manipulate alignments in the SAM format" might make sense for a seasoned bioinformatics researcher, but might be fairly unhelpful or even misleading to a non-specialized audience. It is a good idea to come up with a synopsis that gives an idea of the application domain of the package. In this example, this might give something like "Manipulate nucleotide sequence alignments", which hopefully gives the user a better idea of whether this is what they are looking for.

Descriptions should take between five and ten lines. Use full sentences, and avoid using acronyms without first introducing them. Descriptions can include Texinfo markup, which is useful to introduce ornaments such as `@code` or `@dfn`, bullet lists, or hyperlinks (see Section "Overview" in *GNU Texinfo*). However you should be careful when using some characters for example '`@`' and curly braces which are the basic special characters in Texinfo (see Section "Special Characters" in *GNU Texinfo*). User interfaces such as `guix package --show` take care of rendering it appropriately.

Synopses and descriptions are translated by volunteers at the Translation Project so that as many users as possible can read them in their native language. User interfaces search them and display them in the language specified by the current locale.

Translation is a lot of work so, as a packager, please pay even more attention to your synopses and descriptions as every change may entail additional work for translators. In order to help them, it is possible to make recommendations or instructions visible to them by inserting special comments like this (see Section "xgettext Invocation" in *GNU Gettext*):

```
;; TRANSLATORS: "X11 resize-and-rotate" should not be translated.
(description "ARandR is designed to provide a simple visual front end
for the X11 resize-and-rotate (RandR) extension. ...")
```

7.6.5 Python Modules

We currently package Python 2 and Python 3, under the Scheme variable names `python-2` and `python` as explained in Section 7.6.3 [Version Numbers], page 128. To avoid confusion and naming clashes with other programming languages, it seems desirable that the name of a package for a Python module contains the word `python`.

Some modules are compatible with only one version of Python, others with both. If the package Foo compiles only with Python 3, we name it `python-foo`; if it compiles only with Python 2, we name it `python2-foo`. If it is compatible with both versions, we create two packages with the corresponding names.

If a project already contains the word `python`, we drop this; for instance, the module python-dateutil is packaged under the names `python-dateutil` and `python2-dateutil`.

7.6.6 Perl Modules

Perl programs standing for themselves are named as any other package, using the lowercase upstream name. For Perl packages containing a single class, we use the lowercase class name, replace all occurrences of `::` by dashes and prepend the prefix `perl-`. So the class `XML::Parser` becomes `perl-xml-parser`. Modules containing several classes keep their lowercase upstream name and are also prepended by `perl-`. Such modules tend to have the word `perl` somewhere in their name, which gets dropped in favor of the prefix. For instance, `libwww-perl` becomes `perl-libwww`.

7.6.7 Fonts

For fonts that are in general not installed by a user for typesetting purposes, or that are distributed as part of a larger software package, we rely on the general packaging rules for software; for instance, this applies to the fonts delivered as part of the X.Org system or fonts that are part of TeX Live.

To make it easier for a user to search for fonts, names for other packages containing only fonts are constructed as follows, independently of the upstream package name.

The name of a package containing only one font family starts with `font-`; it is followed by the foundry name and a dash `-` if the foundry is known, and the font family name, in which spaces are replaced by dashes (and as usual, all upper case letters are transformed to lower case). For example, the Gentium font family by SIL is packaged under the name `font-sil-gentium`.

For a package containing several font families, the name of the collection is used in the place of the font family name. For instance, the Liberation fonts consist of three families, Liberation Sans, Liberation Serif and Liberation Mono. These could be packaged separately under the names `font-liberation-sans` and so on; but as they are distributed together under a common name, we prefer to package them together as `font-liberation`.

In the case where several formats of the same font family or font collection are packaged separately, a short form of the format, prepended by a dash, is added to the package name. We use `-ttf` for TrueType fonts, `-otf` for OpenType fonts and `-type1` for PostScript Type 1 fonts.

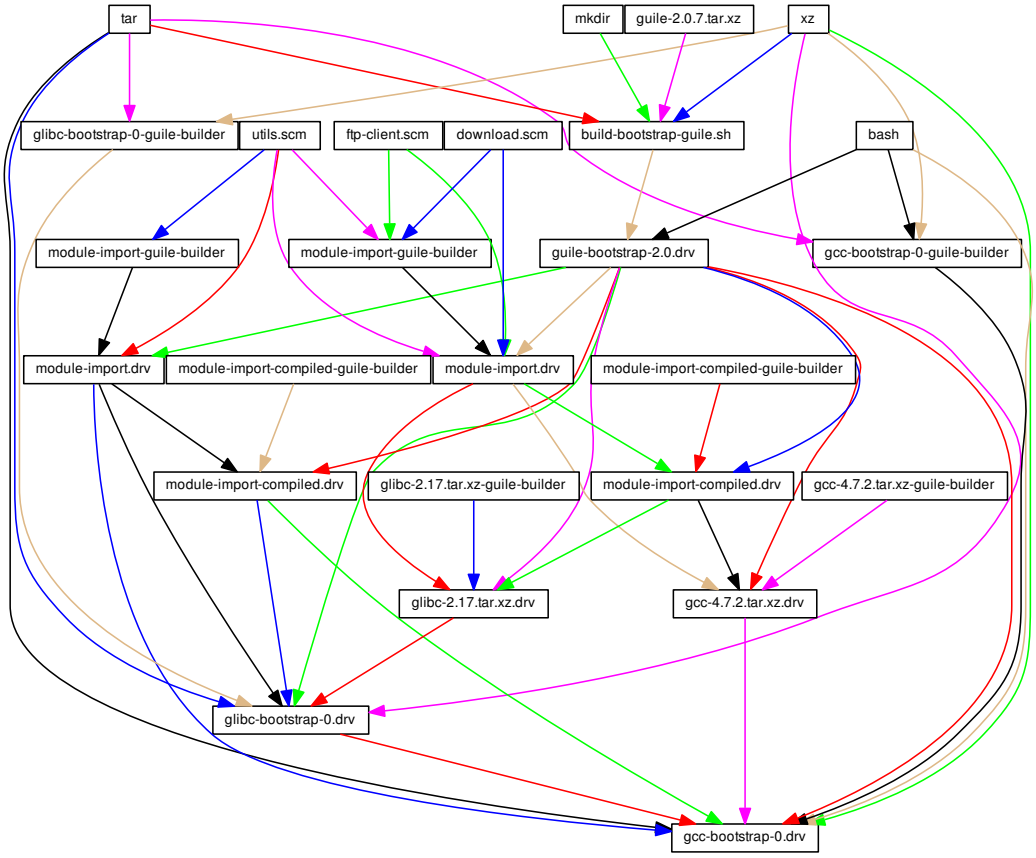
7.7 Bootstrapping

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 (see Chapter 1 [Introduction], page 2). So there's an obvious chicken-and-egg problem: how does the first package get built? How does the first compiler get compiled? Note that this is a question of interest only to the curious hacker, not to the regular user, so you can shamelessly skip this section if you consider yourself a "regular user".

The GNU system is primarily made of C code, with libc at its core. The GNU build system itself assumes the availability of a Bourne shell and command-line tools provided by GNU Coreutils, Awk, Findutils, 'sed', and 'grep'. Furthermore, build programs—programs that run `./configure`, `make`, etc.—are written in Guile Scheme (see Section 5.4 [Derivations], page 47). Consequently, to be able to build anything at all, from scratch, Guix relies on pre-built binaries of Guile, GCC, Binutils, libc, and the other packages mentioned above—the *bootstrap binaries*.

These bootstrap binaries are "taken for granted", though we can also re-create them if needed (more on that later).

Preparing to Use the Bootstrap Binaries



The figure above shows the very beginning of the dependency graph of the distribution, corresponding to the package definitions of the `(gnu packages bootstrap)` module. At this level of detail, things are slightly 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` (see Section 5.3 [The Store], page 46).

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. The `module-import-compiled.drv` derivations compile those modules, and write them in an output directory with the right layout. This corresponds to the `#:modules` argument of `build-expression->derivation` (see Section 5.4 [Derivations], page 47).

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 tool chain.

Building the Build Tools

Bootstrapping is complete when we have a full tool chain that does not depend on the pre-built bootstrap tools discussed above. This no-dependency requirement is verified by checking whether the files of the final tool chain contain references to the `/gnu/store` directories of the bootstrap inputs. The process that leads to this "final" tool chain is described by the package definitions found in the `(gnu packages commencement)` module.

The first tool that gets built with the bootstrap binaries is GNU Make, which is a prerequisite for all the following packages. From there Findutils and Diffutils get built.

Then come the first-stage Binutils and GCC, built as pseudo cross tools—i.e., with `--target` equal to `--host`. They are used to build libc. Thanks to this cross-build trick, this libc is guaranteed not to hold any reference to the initial tool chain.

From there the final Binutils and GCC are built. GCC uses `ld` from the final Binutils, and links programs against the just-built libc. This tool chain is used to build the other packages used by Guix and by the GNU Build System: Guile, Bash, Coreutils, etc.

And voilà! At this point we have the complete set of build tools that the GNU Build System expects. These are in the `%final-inputs` variable of the `(gnu packages commencement)` module, and are implicitly used by any package that uses `gnu-build-system` (see Section 5.2 [Build Systems], page 42).

Building the Bootstrap Binaries

Because the final tool chain does not depend on the bootstrap binaries, those rarely need to be updated. Nevertheless, it is useful to have an automated way to produce them, should an update occur, and this is what the `(gnu packages make-bootstrap)` module provides.

The following command builds the tarballs containing the bootstrap binaries (Guile, Binutils, GCC, libc, and a tarball containing a mixture of Coreutils and other basic command-line tools):

```
guix build bootstrap-tarballs
```

The generated tarballs are those that should be referred to in the `(gnu packages bootstrap)` module mentioned at the beginning of this section.

Still here? Then perhaps by now you've started to wonder: when do we reach a fixed point? That is an interesting question! The answer is unknown, but if you would like to investigate further (and have significant computational and storage resources to do so), then let us know.

7.8 Porting to a New Platform

As discussed above, the GNU distribution is self-contained, and self-containment is achieved by relying on pre-built "bootstrap binaries" (see Section 7.7 [Bootstrapping], page 131). These binaries are specific to an operating system kernel, CPU architecture, and application binary interface (ABI). Thus, to port the distribution to a platform that is not yet supported, one must build those bootstrap binaries, and update the `(gnu packages bootstrap)` module to use them on that platform.

Fortunately, Guix can *cross compile* those bootstrap binaries. When everything goes well, and assuming the GNU tool chain supports the target platform, this can be as simple as running a command like this one:

```
guix build --target=armv5tel-linux-gnueabi bootstrap-tarballs
```

For this to work, the `glibc-dynamic-linker` procedure in `(gnu packages bootstrap)` must be augmented to return the right file name for libc's dynamic linker on that platform; likewise, `system->linux-architecture` in `(gnu packages linux)` must be taught about the new platform.

Once these are built, the `(gnu packages bootstrap)` module needs to be updated to refer to these binaries on the target platform. That is, the hashes and URLs of the bootstrap tarballs for the new platform must be added alongside those of the currently supported platforms. The bootstrap Guile tarball is treated specially: it is expected to be available locally, and `gnu-system.am` has rules do download it for the supported architectures; a rule for the new platform must be added as well.

In practice, there may be some complications. First, it may be that the extended GNU triplet that specifies an ABI (like the `eabi` suffix above) is not recognized by all the GNU tools. Typically, glibc recognizes some of these, whereas GCC uses an extra `--with-abi` configure flag (see `gcc.scm` for examples of how to handle this). Second, some of the required packages could fail to build for that platform. Lastly, the generated binaries could be broken for some reason.

8 Contributing

This project is a cooperative effort, and we need your help to make it grow! Please get in touch with us on guix-devel@gnu.org and #guix on the Freenode IRC network. We welcome ideas, bug reports, patches, and anything that may be helpful to the project. We particularly welcome help on packaging (see Section 7.6 [Packaging Guidelines], page 127).

8.1 Building from Git

If you want to hack Guix itself, it is recommended to use the latest version from the Git repository. When building Guix from a checkout, the following packages are required in addition to those mentioned in the installation instructions (see Section 2.2 [Requirements], page 4).

- GNU Autoconf;
- GNU Automake;
- GNU Gettext;
- GNU Texinfo;
- Graphviz;
- GNU Help2man (optional).

Run `./bootstrap` to download the Nix daemon source code and to generate the build system infrastructure using autoconf. It reports an error if an inappropriate version of the above packages is being used.

If you get an error like this one:

```
configure.ac:46: error: possibly undefined macro: PKG_CHECK_MODULES
```

it probably means that Autoconf couldn't find `pkg.m4`, which is provided by `pkg-config`. Make sure that `pkg.m4` is available. For instance, if you installed Automake in `/usr/local`, it wouldn't look for `.m4` files in `/usr/share`. So you have to invoke the following command in that case

```
export ACLOCAL_PATH=/usr/share/aclocal
```

See see Section "Macro Search Path" in *The GNU Automake Manual* for more information.

Then, run `./configure` as usual.

Finally, you have to invoke `make check` to run tests. If anything fails, take a look at installation instructions (see Chapter 2 [Installation], page 3) or send a message to the mailing list.

8.2 Running Guix Before It Is Installed

In order to keep a sane working environment, you will find it useful to test the changes made in your local source tree checkout without actually installing them. So that you can distinguish between your "end-user" hat and your "motley" costume.

To that end, all the command-line tools can be used even if you have not run `make install`. To do that, prefix each command with `./pre-inst-env` (the `pre-inst-env` script lives in the top build tree of Guix), as in:

```
$ sudo ./pre-inst-env guix-daemon --build-users-group=guixbuild
$ ./pre-inst-env guix build hello
```

Similarly, for a Guile session using the Guix modules:

```
$ ./pre-inst-env guile -c '(use-modules (guix utils)) (pk (%current-system))'

;;; ("x86_64-linux")
```

... and for a REPL (see Section "Using Guile Interactively" in *Guile Reference Manual*):

```
$ ./pre-inst-env guile
scheme@(guile-user)> ,use(guix)
scheme@(guile-user)> ,use(gnu)
scheme@(guile-user)> (define snakes
                       (fold-packages
                         (lambda (package lst)
                           (if (string-prefix? "python"
                                               (package-name package))
                               (cons package lst)
                               lst))
                         '()))
scheme@(guile-user)> (length snakes)
$1 = 361
```

The `pre-inst-env` script sets up all the environment variables necessary to support this, including `PATH` and `GUILE_LOAD_PATH`.

Note that `./pre-inst-env guix pull` does *not* upgrade the local source tree; it simply updates the `~/.config/guix/latest` symlink (see Section 3.6 [Invoking guix pull], page 24). Run `git pull` instead if you want to upgrade your local source tree.

8.3 The Perfect Setup

The Perfect Setup to hack on Guix is basically the perfect setup used for Guile hacking (see Section "Using Guile in Emacs" in *Guile Reference Manual*). First, you need more than an editor, you need Emacs, empowered by the wonderful Geiser.

Geiser allows for interactive and incremental development from within Emacs: code compilation and evaluation from within buffers, access to on-line documentation (docstrings), context-sensitive completion, `M-.` to jump to an object definition, a REPL to try out your code, and more (see Section "Introduction" in *Geiser User Manual*). For convenient Guix development, make sure to augment Guile's load path so that it finds source files from your checkout:

```
;; Assuming the Guix checkout is in ~/src/guix.
(add-to-list 'geiser-guile-load-path "~/src/guix")
```

To actually edit the code, Emacs already has a neat Scheme mode. But in addition to that, you must not miss Paredit. It provides facilities to directly operate on the syntax tree, such as raising an s-expression or wrapping it, swallowing or rejecting the following s-expression, etc.

GNU Guix also comes with a minor mode that provides some additional functionality for Scheme buffers (see Section 4.7 [Emacs Development], page 35).

8.4 Coding Style

In general our code follows the GNU Coding Standards (see *GNU Coding Standards*). However, they do not say much about Scheme, so here are some additional rules.

8.4.1 Programming Paradigm

Scheme code in Guix is written in a purely functional style. One exception is code that involves input/output, and procedures that implement low-level concepts, such as the `memoize` procedure.

8.4.2 Modules

Guile modules that are meant to be used on the builder side must live in the `(guix build ...)` name space. They must not refer to other Guix or GNU modules. However, it is OK for a "host-side" module to use a build-side module.

Modules that deal with the broader GNU system should be in the `(gnu ...)` name space rather than `(guix ...)`.

8.4.3 Data Types and Pattern Matching

The tendency in classical Lisp is to use lists to represent everything, and then to browse them "by hand" using `car`, `cdr`, `cadr`, and co. There are several problems with that style, notably the fact that it is hard to read, error-prone, and a hindrance to proper type error reports.

Guix code should define appropriate data types (for instance, using `define-record-type*`) rather than abuse lists. In addition, it should use pattern matching, via Guile's `(ice-9 match)` module, especially when matching lists.

8.4.4 Formatting Code

When writing Scheme code, we follow common wisdom among Scheme programmers. In general, we follow the Riastradh's Lisp Style Rules. This document happens to describe the conventions mostly used in Guile's code too. It is very thoughtful and well written, so please do read it.

Some special forms introduced in Guix, such as the `substitute*` macro, have special indentation rules. These are defined in the `.dir-locals.el` file, which Emacs automatically uses. If you do not use Emacs, please make sure to let your editor know the rules.

We require all top-level procedures to carry a docstring. This requirement can be relaxed for simple private procedures in the `(guix build ...)` name space, though.

Procedures should not have more than four positional parameters. Use keyword parameters for procedures that take more than four parameters.

8.5 Submitting Patches

Development is done using the Git distributed version control system. Thus, access to the repository is not strictly necessary. We welcome contributions in the form of patches as produced by `git format-patch` sent to the mailing list. Please write commit logs in the ChangeLog format (see Section "Change Logs" in *GNU Coding Standards*); you can check the commit history for examples.

Before submitting a patch that adds or modifies a package definition, please run through this check list:

1. Take some time to provide an adequate synopsis and description for the package. See Section 7.6.4 [Synopses and Descriptions], page 129, for some guidelines.
2. Run `guix lint` *package*, where *package* is the name of the new or modified package, and fix any errors it reports (see Section 6.7 [Invoking guix lint], page 70).
3. Make sure the package builds on your platform, using `guix build` *package*.
4. Take a look at the profile reported by `guix size` (see Section 6.8 [Invoking guix size], page 71). This will allow you to notice references to other packages unwillingly retained. It may also help determine whether to split the package (see Section 3.4 [Packages with Multiple Outputs], page 21), and which optional dependencies should be used.
5. For important changes, check that dependent package (if applicable) are not affected by the change; `guix refresh --list-dependent` *package* will help you do that (see Section 6.6 [Invoking guix refresh], page 68).
6. Check whether the package's build process is deterministic. This typically means checking whether an independent build of the package yields the exact same result that you obtained, bit for bit.

   A simple way to do that is with `guix challenge` (see Section 6.12 [Invoking guix challenge], page 79). You may run it once the package has been committed and built by `hydra.gnu.org` to check whether it obtains the same result as you did. Better yet: Find another machine that can build it and run `guix publish`.

When posting a patch to the mailing list, use '`[PATCH] ...`' as a subject. You may use your email client or the `git send-mail` command.

9 Acknowledgments

Guix is based on the Nix package manager, which was designed and implemented by Eelco Dolstra, with contributions from other people (see the `nix/AUTHORS` file in Guix.) Nix pioneered functional package management, and promoted unprecedented features, such as transactional package upgrades and rollbacks, per-user profiles, and referentially transparent build processes. Without this work, Guix would not exist.

The Nix-based software distributions, Nixpkgs and NixOS, have also been an inspiration for Guix.

GNU Guix itself is a collective work with contributions from a number of people. See the `AUTHORS` file in Guix for more information on these fine people. The `THANKS` file lists people who have helped by reporting bugs, taking care of the infrastructure, providing artwork and themes, making suggestions, and more—thank you!

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

.

.local, host name lookup 110

A

authorizing, archives 26

B

bag (low-level package representation) 42
boot loader 113
bootstrap binaries 131
bootstrapping 131
build code quoting 53
build environment 8
build hook 6, 9
build phases 43
build system 42
build users 5

C

chroot 6, 8
closure 23, 71
common build options 63
container 77, 80
container, build environment 8
CPAN 67
CRAN 67
cross compilation 54
cross-compilation 39, 62
customization, of packages 126
customization, of services 87

D

daemon 5
daemons 117
DAG 72
debugging files 124
deduplication 10, 24
derivation 37
derivation path 47
derivations 47
development environments 75
device mapping 93
DHCP, networking service 101
digital signatures 20
disk encryption 93

E

elpa 68
Emacs 27

F

file-like objects 57
firmware 89
functional package management 2

G

G-expression 53
garbage collector 22
gem 66
GNU Build System 38
grafts 125
GRUB 113
Guix System Distribution 2, 82
GuixSD 2, 82

H

hackage 67
hosts file 89
HTTPS, certificates 109

I

importing packages 66
incompatibility, of locale data 97
init system 123
initial RAM disk (initrd) 112
initrd (initial RAM disk) 112
integrity checking 23
integrity, of the store 23

L

locale 96
locale definition 96
locale name 97
locales, when not on GuixSD 11
lowering, of high-level objects in gexps 54, 58
LUKS 93

M

mapped devices 93
monad 50
monadic functions 50
monadic values 50
multiple-output packages 21

N

name service cache daemon 99
name service switch 110
network management 101
non-determinism, in package builds 80
normalized codeset in locale names 97
nscd 99
`nss-certs` 109
nss-mdns 110
NSS 110

O

offloading 6

P

package conversion 66
package definition, editing 64
package import 66
package module search path 126
package outputs 21
PAM 91
patches 38
PID 1 123
pluggable authentication modules 91
pre-built binaries 20
profile declaration 16
profile manifest 16
propagated inputs 15, 40
pypi 66

R

read-eval-print loop 29, 136
repairing the store 24
replacements of packages, for grafts 125
REPL 29, 136
reproducibility 13
reproducible build environments 75
reproducible builds 8, 13, 79
roll-back, of the operating system 88

S

search paths 14, 17
security 20
security updates 125
service extensions 117
service type 121
service types 118
services 117
setuid programs 108
signing, archives 25
state monad 52
store 2, 46
store paths 46
strata of code 53
substituter 127
substitutes 9, 14, 20
sudoers file 91
swap devices 90
system configuration 85
system service 118
system services 97

T

Texinfo markup, in package descriptions 129
TLS 109

V

verifiable builds 79
virtual machine 114
VM 114

W

wicd 101

X

X session 103
X.509 certificates 109

Programming Index

#

`#~exp` 55

(

`(gexp` 55

>

`>>=` 51

A

`add-text-to-store` 47
`avahi-service` 103

B

`base-initrd` 112
`bitlbee-service` 102
`build-derivations` 47
`build-expression->derivation` 49

C

`close-connection` 47
`colord-service` 107
`computed-file` 57
`console-keymap-service` 101
`current-state` 52

D

`dbus-service` 105
`derivation` 48
`dhcp-client-service` 101

E

`elogind-service` 105
`expression->initrd` 113

F

`fold-services` 122

G

`geoclue-application` 107
`geoclue-service` 107
`gexp->derivation` 56
`gexp->file` 58
`gexp->script` 57
`gexp?` 56
`guix-publish-service` 101
`guix-service` 101

H

`host-name-service` 98

I

`interned-file` 53

L

`lirc-service` 108
`local-file` 57
`lower-object` 59
`lsh-service` 102

M

`mbegin` 52
`mingetty-service` 98
`mixed-text-file` 58
`mlet` 51
`mlet*` 51
`modify-services` 87, 121

N

`nginx-service` 108
`nscd-service` 99
`ntp-service` 101

O

`open-connection` 47
`operating-system` 85
`operating-system-derivation` 89

P

`package->cross-derivation` 53
`package->derivation` 53
`package-cross-derivation` 39
`package-derivation` 39
`package-file` 53
`packages->manifest` 17
`plain-file` 57
`polkit-service` 107
`postgresql-service` 108
`program-file` 58

R

return 51
run-with-state 52
run-with-store 53

S

scheme-file 58
screen-locker-service 104
service 120
service-extension 122
service-extension? 122
service-kind 120
service-parameters 120
service? 120
set-current-state 52
slim-service 103
state-pop 52
state-push 52
static-networking-service 101
syslog-service 100

T

text-file 53
text-file* 58
tor-service 102

U

udev-service 101
udisks-service 107
upower-service 107

V

valid-path? 47

W

wicd-service 101
with-monad 51

X

xorg-configuration-file 104
xorg-start-command 104