GNU-Guix-Kochbuch

Anleitungen und Beispiele, wie man den funktionalen Paketmanager GNU Guix benutzt

Die Entwickler von GNU Guix

Inhaltsverzeichnis

1 Anleitungen zu Scheme

GNU Guix ist in Scheme geschrieben, einer für alle Anwendungszwecke geeigneten Programmiersprache, und viele Funktionalitäten von Guix können programmatisch angesteuert und verändert werden. Sie können Scheme benutzen, um Paketdefinitionen zu erzeugen, abzuändern, ganze Betriebssysteme einzuspielen etc.

Wenn man die Grundzüge kennt, wie man in Scheme programmiert, bekommt man Zugang zu vielen der fortgeschrittenen Funktionen von Guix – und Sie müssen dazu nicht einmal ein erfahrener Programmierer sein!

Legen wir los!

1.1 Ein Schnellkurs in Scheme

Die von Guix benutzte Scheme-Implementierung nennt sich Guile. Um mit der Sprache herumspielen zu können, installieren Sie Guile mit `guix install guile` und starten eine interaktive Programmierumgebung (englisch Read-Eval-Print-Loop (`https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop`), kurz REPL), indem Sie `guile` auf der Befehlszeile ausführen.

Alternativ können Sie auch den Befehl `guix shell -- guile` ausführen, wenn Sie Guile lieber *nicht* in Ihr Nutzerprofil installieren wollen.

In den folgenden Beispielen stehen die Zeilen dafür, was Sie auf der REPL eintippen; wenn eine Zeile mit „⇒“ beginnt, zeigt sie das Ergebnis einer Auswertung, während Zeilen, die mit „ ⊣ “ beginnen, für eine angezeigte Ausgabe stehen. Siehe Abschnitt “Using Guile Interactively” in *das Referenzhandbuch zu GNU Guile* für mehr Details zur REPL.

- Die Scheme-Syntax ist an sich ein Baum von Ausdrücken (Lisp-Programmierer nennen sie *symbolische Ausdrücke*, kurz *S-Ausdrücke* bzw. englisch *s-expression*). Ein solcher Ausdruck kann ein Literal sein, wie z.B. Zahlen oder Zeichenketten, oder er kann ein zusammengesetzter Ausdruck sein, d.h. eine geklammerte Liste von zusammengesetzten und literalen Ausdrücken. Dabei stehen `#true` und `#false` (abgekürzt auch `#t` und `#f`) jeweils für die Booleschen Werte „wahr“ und „falsch“.

  Beispiele für gültige Ausdrücke

  ```
  "Hallo Welt!"
  ⇒ "Hallo Welt!"

  17
  ⇒ 17

  (display (string-append "Hallo " "Guix" "\n"))
  ⊣ Hallo Guix!
  ⇒ #<unspecified>
  ```

- Das letzte Beispiel eben ist der Aufruf einer Funktion innerhalb eines anderen Funktionsaufrufs. Wenn ein geklammerter Ausdruck ausgewertet wird, ist der erste Term die Funktion und der Rest sind die Argumente, die an die Funktion übergeben werden. Jede Funktion liefert ihren zuletzt ausgewerteten Ausdruck als ihren Rückgabewert.

- Anonyme Funktionen – bzw. Prozeduren, wie man in Scheme sagt – werden mit dem `lambda`-Term deklariert:

  ```
  (lambda (x) (* x x))
  ⇒ #<procedure 120e348 at <unknown port>:24:0 (x)>
  ```

  Die obige Prozedur liefert das Quadrat ihres Arguments. Weil alles ein Ausdruck ist, liefert der Ausdruck `lambda` eine anonyme Prozedur, die wiederum auf ein Argument angewandt werden kann:

  ```
  ((lambda (x) (* x x)) 3)
  ⇒ 9
  ```

  Prozeduren sind normale Werte wie Zahlen, Zeichenketten, Boolesche Ausdrücke und dererlei auch.

- Allem kann mit `define` ein globaler Name zugewiesen werden:

  ```
  (define a 3)
  (define quadrat (lambda (x) (* x x)))
  (quadrat a)
  ⇒ 9
  ```

- Prozeduren können auch kürzer mit der folgenden Syntax definiert werden:

  ```
  (define (quadrat x) (* x x))
  ```

- Eine Listenstruktur kann mit der `list`-Prozedur erzeugt werden:

  ```
  (list 2 a 5 7)
  ⇒ (2 3 5 7)
  ```

- Für die Erzeugung und Verarbeitung von Listen stehen im Modul `(srfi srfi-1)` Standardprozeduren zur Verfügung (siehe Abschnitt “SRFI-1” in *Referenzhandbuch zu GNU Guile*). Hier ein kurzer Überblick:

  ```
  (use-modules (srfi srfi-1))  ;Prozeduren zur Listenverarbeitung importieren

  (append (list 1 2) (list 3 4))
  ⇒ (1 2 3 4)

  (map (lambda (x) (* x x)) (list 1 2 3 4))
  ⇒ (1 4 9 16)

  (delete 3 (list 1 2 3 4))        ⇒ (1 2 4)
  (filter odd? (list 1 2 3 4))     ⇒ (1 3)
  (remove even? (list 1 2 3 4))    ⇒ (1 3)
  (find number? (list "a" 42 "b")) ⇒ 42
  ```

  Beachten Sie, dass das erste Argument an `map`, `filter`, `remove` und `find` eine Prozedur ist!

- Mit dem *quote*-Zeichen wird das Auswerten eines geklammerten Ausdrucks abgeschaltet (man spricht dann von einem symbolischen Ausdruck, engl. „S-Expression“ bzw. „S-exp“): Der erste Term wird *nicht* auf den anderen Termen aufgerufen (siehe Abschnitt “Expression Syntax” in *Referenzhandbuch zu GNU Guile*). Folglich liefert es quasi eine Liste von Termen.

  ```
  '(display (string-append "Hallo " "Guix" "\n"))
  ```

```
⇒ (display (string-append "Hallo " "Guix" "\n"))

'(2 a 5 7)
⇒ (2 a 5 7)
```

- Mit einem `quasiquote`-Zeichen (`` ` ``) wird die Auswertung eines geklammerten Ausdrucks so lange abgeschaltet, bis ein `unquote` (ein Komma) sie wieder aktiviert. Wir können damit genau steuern, was ausgewertet wird und was nicht.

  ```
  `(2 a 5 7 (2 ,a 5 ,(+ a 4)))
  ⇒ (2 a 5 7 (2 3 5 7))
  ```

  Beachten Sie, dass obiges Ergebnis eine Liste verschiedenartiger Elemente ist: Zahlen, Symbole (in diesem Fall `a`) und als letztes Element selbst wieder eine Liste.

- In Guix gibt es außerdem eine aufgeputschte Form von S-Ausdrücken: die *G-Ausdrücke* oder „Gexps“, die ihre eigene Art von `quasiquote` und `unquote` haben: `#~` (oder `gexp`) und `#$` (oder `ungexp`). Mit Ihnen kennzeichnen Sie, welcher Code *erst später ausgeführt* werden soll („Code-Staging“).

  G-Ausdrücke begegnen Ihnen etwa in manchen Paketdefinitionen, wo sie eingesetzt werden, damit Code erst bei der Erstellung des Pakets abläuft. Das sieht so aus:

  ```
  (use-modules (guix gexp)           ;für's Schreiben von G-Ausdrücken
               (gnu packages base))  ;für 'coreutils'

  ;; Folgender G-Ausdruck enthält später ausgeführten Code.
  #~(begin
      ;; 'ls' aufrufen aus dem Paket, das in der Variablen 'coreutils'
      ;; definiert ist.
      (system* #$(file-append coreutils "/bin/ls") "-l")

      ;; Das Ausgabeverzeichnis des aktuell definierten Pakets anlegen.
      (mkdir #$output))
  ```

  Siehe Abschnitt “G-Ausdrücke” in *Referenzhandbuch zu GNU Guix* für Details zu G-Ausdrücken.

- Mehrere Variable können in einer lokalen Umgebung mit Bezeichnern versehen werden, indem Sie `let` benutzen (siehe Abschnitt “Local Bindings” in *Referenzhandbuch zu GNU Guile*):

  ```
  (define x 10)
  (let ((x 2)
        (y 3))
    (list x y))
  ⇒ (2 3)

  x
  ⇒ 10

  y
  error  In procedure module-lookup: Unbound variable: y
  ```

  Benutzen Sie `let*`, damit spätere Variablendeklarationen auf frühere verweisen können.

```
(let* ((x 2)
       (y (* x 3)))
  (list x y))
⇒ (2 6)
```

- Mit *Schlüsselwörtern* bestimmen wir normalerweise diejenigen Parameter einer Prozedur, die einen Namen haben sollen. Ihnen wird `#:` (Doppelkreuz und Doppelpunkt) vorangestellt, gefolgt von alphanumerischen Zeichen: `#:etwa-so`. Siehe Abschnitt "Keywords" in *Referenzhandbuch zu GNU Guile*.
- Das Prozentzeichen `%` wird in der Regel für globale Variable auf Erstellungsebene benutzt, auf die nur lesend zugegriffen werden soll. Beachten Sie, dass es sich dabei nur um eine Konvention handelt, ähnlich wie `_` in C. Scheme behandelt `%` genau wie jedes andere Zeichen.
- Module werden mit Hilfe von `define-module` erzeugt (siehe Abschnitt "Creating Guile Modules" in *Referenzhandbuch zu GNU Guile*). Zum Beispiel definiert man mit

  ```
  (define-module (guix build-system ruby)
    #:use-module (guix store)
    #:export (ruby-build
              ruby-build-system))
  ```

  das Modul `guix build-system ruby`, das sich unter dem Pfad `guix/build-system/ruby.scm` innerhalb irgendeines Verzeichnisses im Guile-Ladepfad befinden muss. Es hat eine Abhängigkeit auf das Modul `(guix store)` und exportiert zwei seiner Variablen, `ruby-build` und `ruby-build-system`.

  Siehe Abschnitt "Paketmodule" in *Referenzhandbuch zu GNU Guix* für Informationen zu Modulen, in denen Pakete definiert werden.

  Vertiefung: Die Sprache Scheme wurde oft eingesetzt, um Programmieren beizubringen, daher gibt es ein breites Angebot an Lehrmaterialien, die sie als Transportmittel einsetzen. Hier sehen Sie eine Auswahl solcher Dokumente, mit denen Sie mehr über Scheme erfahren können:

  - *Ein Scheme-Primer* (`https://spritely.institute/news/ein-scheme-primer.html`), von Christine Lemmer-Webber und dem Spritely Institute, auf Deutsch übersetzt von Florian Pelz.
  - *Scheme at a Glance* (`http://www.troubleshooters.com/codecorn/scheme_guile/hello.htm`) von Steve Litt.
  - Eines der Referenzbücher zu Scheme ist das einflussreiche „Structure and Interpretation of Computer Programs“, von Harold Abelson und Gerald Jay Sussman, mit Julie Sussman. Eine deutsche Übersetzung „Struktur und Interpretation von Computerprogrammen“ hat Susanne Daniels-Herold verfasst. Vom englischen Original finden Sie eine kostenlose Ausgabe online (`https://sarabander.github.io/sicp/`). Viele kennen es unter dem Akronym SICP.

    Das Buch können Sie auch installieren und von Ihrem Rechner aus lesen:

    ```
    guix install sicp info-reader
    info sicp
    ```

Sie finden noch mehr Bücher, Anleitungen und andere Ressourcen auf `https://schemers.org/`.

2 Pakete schreiben

In diesem Kapitel bringen wir Ihnen bei, wie Sie Pakete zur mit GNU Guix ausgelieferten Paketsammlung beitragen. Dazu gehört, Paketdefinitionen in Guile Scheme zu schreiben, sie in Paketmodulen zu organisieren und sie zu erstellen.

2.1 Anleitung zum Paketeschreiben

GNU Guix zeichnet sich in erster Linie deswegen als das *hackbare* Paketverwaltungswerkzeug aus, weil es mit GNU Guile (`https://www.gnu.org/software/guile/`) arbeitet, einer mächtigen, hochsprachlichen Programmiersprache, die einen der Dialekte von Scheme (`https://de.wikipedia.org/wiki/Scheme`) darstellt. Scheme wiederum gehört zur Lisp-Familie von Programmiersprachen (`https://de.wikipedia.org/wiki/Lisp`).

Paketdefinitionen werden ebenso in Scheme geschrieben, wodurch Guix auf sehr einzigartige Weise mächtiger wird als die meisten anderen Paketverwaltungssysteme, die Shell-Skripte oder einfache Sprachen benutzen.

- Sie können sich Funktionen, Strukturen, Makros und all die Ausdrucksstärke von Scheme für Ihre Paketdefinitionen zu Nutze machen.
- Durch Vererbung können Sie ohne viel Aufwand ein Paket anpassen, indem Sie von ihm erben lassen und nur das Nötige abändern.
- Stapelverarbeitung („batch mode“) wird möglich; die ganze Paketsammlung kann analysiert, gefiltert und verarbeitet werden. Versuchen Sie, ein Serversystem ohne Bildschirm („headless“) auch tatsächlich von allen Grafikschnittstellen zu befreien? Das ist möglich. Möchten Sie alles von Neuem aus seinem Quellcode erstellen, aber mit eingeschalteten besonderen Compileroptimierungen? Übergeben Sie einfach das passende `#:make-flags "..."`-Argument an die Paketliste. Es wäre nicht übertrieben, hier an die USE-Optionen von Gentoo (`https://wiki.gentoo.org/wiki/USE_flag`) zu denken, aber das hier übertrifft sie: Der Paketautor muss vorher gar nicht darüber nachgedacht haben, der Nutzer kann sie selbst *programmieren*!

Die folgende Anleitung erklärt alles Grundlegende über das Schreiben von Paketen mit Guix. Dabei setzen wir kein großes Wissen über das Guix-System oder die Lisp-Sprache voraus. Vom Leser wird nur erwartet, dass er mit der Befehlszeile vertraut ist und über grundlegende Programmierkenntnisse verfügt.

2.1.1 Ein Hallo-Welt-Paket

Der Abschnitt „Pakete definieren“ im Handbuch führt in die Grundlagen des Paketschreibens für Guix ein (siehe Abschnitt “Pakete definieren” in *Referenzhandbuch zu GNU Guix*). Im folgenden Abschnitt werden wir diese Grundlagen teilweise rekapitulieren.

GNU Hello ist ein Projekt, das uns als Stellvertreter für „richtige“ Projekte und allgemeines Beispiel für das Schreiben von Paketen dient. Es verwendet das GNU-Erstellungssystem (`./configure && make && make install`). Guix stellt uns schon eine Paketdefinition zur Verfügung, die uns einen perfekten Ausgangspunkt bietet. Sie können sich ihre Deklaration anschauen, indem Sie `guix edit hello` von der Befehlszeile ausführen. Schauen wir sie uns an:

```
(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)
  (synopsis "Hello, GNU world: An example GNU package")
  (description
   "GNU Hello prints the message \"Hello, world!\" and then exits.  It
serves as an example of standard GNU coding practices.  As such, it supports
command-line arguments, multiple languages, and so on.")
  (home-page "https://www.gnu.org/software/hello/")
  (license gpl3+)))
```

Wie Sie sehen können, ist das meiste klar strukturiert. Aber sehen wir uns die Felder zusammen an:

‘`name`’ Der Name des Projekts. Wir halten uns an die Konventionen von Scheme und bevorzugen deshalb Kleinschreibung ohne Unterstriche, sondern mit Bindestrichen zwischen den Wörtern.

‘`source`’ Dieses Feld enthält eine Beschreibung, was der Ursprung des Quellcodes ist. Das `origin`-Verbundsobjekt enthält diese Felder:

1. Die Methode. Wir verwenden hier `url-fetch`, um über HTTP/FTP herunterzuladen, aber es gibt auch andere Methoden wie `git-fetch` für Git-Repositorys.
2. Die URI, welche bei `url-fetch` normalerweise eine Ortsangabe mit `https://` ist. In diesem Fall verweist die besondere URI ‚mirror://gnu‘ auf eine von mehreren wohlbekannten Ortsangaben, von denen Guix jede durchprobieren kann, um den Quellcode herunterzuladen, wenn es bei manchen davon nicht klappt.
3. Die `sha256`-Prüfsumme der angefragten Datei. Sie ist notwendig, damit sichergestellt werden kann, dass der Quellcode nicht beschädigt ist. Beachten Sie, dass Guix mit Zeichenketten in Base32-Kodierung arbeitet, weshalb wir die `base32`-Funktion aufrufen.

‘`build-system`’
Hier glänzt Schemes Fähigkeit zur Abstraktion: In diesem Fall abstrahiert `gnu-build-system` die berühmten Schritte `./configure && make && make install`, die sonst in der Shell aufgerufen würden. Zu den anderen Erstellungssystemen gehören das `trivial-build-system`, das nichts tut und dem Paketautoren das Schreiben sämtlicher Erstellungsschritte abverlangt, das `python-build-system`, das `emacs-build-system`, und viele mehr (siehe Abschnitt “Erstellungssysteme” in *Referenzhandbuch zu GNU Guix*).

‘`synopsis`’
: Die Zusammenfassung. Sie sollte eine knappe Beschreibung sein, was das Paket tut. Für viele Pakete findet sich auf der Homepage ein Einzeiler, der als Zusammenfassung benutzt werden kann.

‘`description`’
: Genau wie bei der Zusammenfassung ist es in Ordnung, die Beschreibung des Projekts für das Paket wiederzuverwenden. Beachten Sie, dass Guix dafür Texinfo-Syntax verlangt.

‘`home-page`’
: Hier soll möglichst HTTPS benutzt werden.

‘`license`’ Siehe die vollständige Liste verfügbarer Lizenzen in `guix/licenses.scm` im Guix-Quellcode.

Es wird Zeit, unser erstes Paket zu schreiben! Aber noch nichts tolles, wir bleiben bei einem Paket `my-hello` stelltvertretend für „richtige“ Software; es ist eine Kopie obiger Deklaration.

Genau wie beim Ritual, Neulinge in Programmiersprachen „Hallo Welt“ schreiben zu lassen, fangen wir mit der vielleicht „arbeitsintensivsten“ Herangehensweise ans Paketeschreiben an. Wir kümmern uns später darum, wie man am besten an Paketen arbeitet; erst einmal nehmen wir den einfachsten Weg.

Speichern Sie den folgenden Code in eine Datei `my-hello.scm`.

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

(package
  (name "my-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, Guix world: An example custom Guix package")
  (description
   "GNU Hello prints the message \"Hello, world!\" and then exits.  It
serves as an example of standard GNU coding practices.  As such, it supports
command-line arguments, multiple languages, and so on.")
  (home-page "https://www.gnu.org/software/hello/")
  (license gpl3+))
```

Wir erklären den zusätzlichen Code in Kürze.

Spielen Sie ruhig mit unterschiedlichen Werten für die verschiedenen Felder herum. Wenn Sie den Quellort (die „source“) ändern, müssen Sie die Prüfsumme aktualisieren. Tatsächlich weigert sich Guix, etwas zu erstellen, wenn die angegebene Prüfsumme nicht zu der berechneten Prüfsumme des Quellcodes passt. Um die richtige Prüfsumme für die Paketdeklaration zu finden, müssen wir den Quellcode herunterladen, die SHA256-Summe davon berechnen und sie in Base32 umwandeln.

Glücklicherweise kann Guix diese Aufgabe automatisieren; wir müssen lediglich die URI übergeben.

```
$ guix download mirror://gnu/hello/hello-2.10.tar.gz

Starting download of /tmp/guix-file.JLYgL7
From https://ftpmirror.gnu.org/gnu/hello/hello-2.10.tar.gz...
following redirection to `https://mirror.ibcp.fr/pub/gnu/hello/hello-2.10.tar.gz'...
 ...10.tar.gz  709KiB                                  2.5MiB/s 00:00 [################
/gnu/store/hbdalsf5lpf01x4dcknwx6xbn6n5km6k-hello-2.10.tar.gz
0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i
```

In diesem speziellen Fall sagt uns die Ausgabe, welcher Spiegelserver ausgewählt wurde. Wenn das Ergebnis des obigen Befehls nicht dasselbe ist wie im Codeschnipsel, dann aktualisieren Sie Ihre `my-hello`-Deklaration entsprechend.

Beachten Sie, dass Tarball-Archive von GNU-Paketen mit einer OpenPGP-Signatur ausgeliefert werden, deshalb sollten Sie mit Sicherheit die Signatur dieses Tarballs mit „gpg“ überprüfen, um ihn zu authentifizieren, bevor Sie weitermachen.

```
$ guix download mirror://gnu/hello/hello-2.10.tar.gz.sig

Starting download of /tmp/guix-file.03tFfb
From https://ftpmirror.gnu.org/gnu/hello/hello-2.10.tar.gz.sig...
following redirection to `https://ftp.igh.cnrs.fr/pub/gnu/hello/hello-2.10.tar.gz.sig'
 ....tar.gz.sig  819B
/gnu/store/rzs8wba9ka7grrmgcpfyxvs58mly0sx6-hello-2.10.tar.gz.sig
0q0v86n3y38z17rl146gdakw9xc4mcscpk8dscs412j22glrv9jf
$ gpg --verify /gnu/store/rzs8wba9ka7grrmgcpfyxvs58mly0sx6-hello-2.10.tar.gz.sig /gnu/
gpg: Signatur vom So 16 Nov 2014 13:08:37 CET
gpg:                mittels RSA-Schlüssel A9553245FDE9B739
gpg: Korrekte Signatur von "Sami Kerola (https://www.iki.fi/kerolasa/) <kerolasa@iki.f
gpg: WARNUNG: Dieser Schlüssel trägt keine vertrauenswürdige Signatur!
gpg:          Es gibt keinen Hinweis, daß die Signatur wirklich dem vorgeblichen Besit
Haupt-Fingerabdruck  = 8ED3 96E3 7E38 D471 A005  30D3 A955 3245 FDE9 B739
```

Sie können dann unbesorgt das hier ausführen:

```
$ guix package --install-from-file=my-hello.scm
```

Nun sollte `my-hello` in Ihrem Profil enthalten sein!

```
$ guix package --list-installed=my-hello
my-hello 2.10 out
/gnu/store/f1db2mfm8syb8qvc357c53slbvf1g9m9-my-hello-2.10
```

Wir sind so weit gekommen, wie es ohne Scheme-Kenntnisse möglich ist. Bevor wir mit komplexeren Paketen weitermachen, ist jetzt der Zeitpunkt gekommen, Ihr Wissen über

Scheme zu entstauben. Siehe Abschnitt 1.1 [Ein Schnellkurs in Scheme], Seite 1, für eine Auffrischung.

2.1.2 Herangehensweisen

Im Rest dieses Kapitels setzen wir ein paar grundlegende Scheme-Programmierkenntnisse voraus. Wir wollen uns nun verschiedene mögliche Herangehensweisen anschauen, wie man an Guix-Paketen arbeiten kann.

Es gibt mehrere Arten, eine Umgebung zum Paketeschreiben aufzusetzen.

Unsere Empfehlung ist, dass Sie direkt am Checkout des Guix-Quellcodes arbeiten, weil es dann für alle einfacher ist, zu Guix beizutragen.

Werfen wir aber zunächst einen Blick auf andere Möglichkeiten.

2.1.2.1 Lokale Datei

Diese Methode haben wir zuletzt für '`my-hello`' benutzt. Jetzt nachdem wir uns mit den Scheme-Grundlagen befasst haben, können wir uns den Code am Anfang erklären. `guix package --help` sagt uns:

```
-f, --install-from-file=DATEI
                        das Paket installieren, zu dem der Code in der DATEI
                          ausgewertet wird
```

Daher *muss* der letzte Ausdruck ein Paket liefern, was im vorherigen Beispiel der Fall ist.

Der Ausdruck `use-modules` sagt aus, welche Module in der Datei gebraucht werden. Module sind eine Sammlung aus Werten und Prozeduren. In anderen Programmiersprachen werden sie oft „Bibliotheken“ oder „Pakete“ genannt.

2.1.2.2 Kanäle

Guix und seine Paketsammlung können durch *Kanäle* (englisch: Channels) erweitert werden. Ein Kanal ist ein Git-Repository, öffentlich oder nicht, das `.scm`-Dateien enthält, die Pakete (siehe Abschnitt “Pakete definieren” in *Referenzhandbuch zu GNU Guix*) oder Dienste (siehe Abschnitt “Dienste definieren” in *Referenzhandbuch zu GNU Guix*) bereitstellen.

Wie würden Sie einen Kanal erstellen? Erstellen Sie zunächst ein Verzeichnis, das Ihre `.scm`-Dateien enthalten wird, beispielsweise `~/my-channel`:

```
mkdir ~/my-channel
```

Angenommen, Sie möchten das Paket '`my-hello`', das wir zuvor gesehen haben, hinzufügen; es bedarf zunächst einiger Anpassungen:

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

(define-public my-hello
  (package
    (name "my-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, Guix world: An example custom Guix package")
    (description
     "GNU Hello prints the message \"Hello, world!\" and then exits.  It
serves as an example of standard GNU coding practices.  As such, it supports
command-line arguments, multiple languages, and so on.")
    (home-page "https://www.gnu.org/software/hello/")
    (license gpl3+)))
```

Beachten Sie, dass wir den Paketwert einer exportierten Variablen mit `define-public` zugewiesen haben. Das bedeutet, das Paket wird einer Variablen `my-hello` zugewiesen, damit darauf verwiesen werden kann. Unter anderem kann es dadurch als Abhängigkeit anderer Pakete verwendet werden.

Wenn Sie `guix package --install-from-file=my-hello.scm` auf der obigen Datei aufrufen, geht es schief, weil der letzte Ausdruck, `define-public`, kein Paket zurückliefert. Wenn Sie trotzdem `define-public` für jene Herangehensweise verwenden möchten, stellen Sie sicher, dass am Ende der Datei eine Auswertung von `my-hello` steht:

```
;; ...
(define-public my-hello
  ;; ...
  )

my-hello
```

Meistens tut man das aber nicht.

Wie machen Sie nun dieses Paket für `guix`-Befehle sichtbar, damit Sie Ihre Pakete testen können? Sie müssen das Verzeichnis mit der Befehlszeilenoption `-L` zum Suchpfad hinzufügen, wie in diesen Beispielen:

```
guix show -L ~/my-channel my-hello
guix build -L ~/my-channel my-hello
```

Der letzte Schritt besteht darin, `~/my-channel` in einen echten Kanal zu verwandeln, so dass Ihre Paketsammlung nahtlos mit jedem `guix`-Befehl zur Verfügung steht. Dazu müssen Sie es zunächst zu einem Git-Repository machen:

```
cd ~/my-channel
git init
git add my-hello.scm
git commit -m "Erster Commit meines Kanals."
```

Und das war's, Sie haben einen Kanal! Von nun an können Sie diesen Kanal zu Ihrer Kanalkonfiguration in `~/.config/guix/channels.scm` hinzufügen (siehe Abschnitt "Wei-

tere Kanäle angeben" in *Referenzhandbuch zu GNU Guix*). Unter der Annahme, dass Sie Ihren Kanal vorerst lokal halten, würde die `channels.scm` etwa so aussehen:

```
(append (list (channel
                (name 'my-channel)
                (url (string-append "file://" (getenv "HOME")
                                    "/my-channel"))))
        %default-channels)
```

Wenn Sie das nächste Mal `guix pull` ausführen, wird Ihr Kanal aufgenommen und die darin definierten Pakete stehen allen `guix`-Befehlen zur Verfügung, auch wenn Sie `-L` nicht übergeben. Der Befehl `guix describe` wird zeigen, dass Guix tatsächlich sowohl den Kanal `my-channel` als auch den `guix`-Kanal benutzt.

Siehe Abschnitt "Einen Kanal erstellen" in *Referenzhandbuch zu GNU Guix* für Details.

2.1.2.3 Direkt am Checkout hacken

Es wird empfohlen, direkt am Code des Guix-Projekts zu arbeiten, weil Ihre Änderungen dann später mit weniger Schwierigkeiten bei uns eingereicht werden können, damit Ihre harte Arbeit der Gemeinschaft nützt!

Anders als die meisten Software-Distributionen werden bei Guix sowohl Werkzeuge (einschließlich des Paketverwaltungsprogramms) als auch die Paketdefinitionen in einem Repository gespeichert. Der Grund für diese Entscheidung war, dass Entwickler die Freiheit haben sollten, die Programmierschnittstelle (API) zu ändern, ohne Inkompatibilitäten einzuführen, indem alle Pakete gleichzeitig mit der API aktualisiert werden. Dadurch wird die Entwicklung weniger träge.

Legen Sie ein Checkout des offiziellen Git-Repositorys (`https://git-scm.com/`) an:

```
$ git clone https://git.savannah.gnu.org/git/guix.git
```

Im Rest dieses Artikels schreiben wir '`$GUIX_CHECKOUT`', wenn wir den Ort meinen, an dem das Checkout gespeichert ist.

Folgen Sie den Anweisungen im Handbuch (siehe (Abschnitt "Mitwirken" in *Referenzhandbuch zu GNU Guix*), um die nötige Umgebung für die Nutzung des Repositorys herzustellen.

Sobald sie hergestellt wurde, sollten Sie die Paketdefinitionen aus der Repository-Umgebung benutzen können.

Versuchen Sie sich ruhig daran, die Paketdefinitionen zu editieren, die Sie in '`$GUIX_CHECKOUT/gnu/packages`' finden.

Das Skript '`$GUIX_CHECKOUT/pre-inst-env`' ermöglicht es Ihnen, '`guix`' auf der Paketsammlung des Repositorys aufzurufen (siehe Abschnitt "Guix vor der Installation ausführen" in *Referenzhandbuch zu GNU Guix*).

- So suchen Sie Pakete, z.B. Ruby:

```
$ cd $GUIX_CHECKOUT
$ ./pre-inst-env guix package --list-available=ruby
   ruby    1.8.7-p374      out     gnu/packages/ruby.scm:119:2
   ruby    2.1.6   out     gnu/packages/ruby.scm:91:2
   ruby    2.2.2   out     gnu/packages/ruby.scm:39:2
```

- Erstellen Sie ein Paket, z.B. Ruby in Version 2.1:

```
$ ./pre-inst-env guix build --keep-failed ruby@2.1
/gnu/store/c13v73jxmj2nir2xjqaz5259zywsa9zi-ruby-2.1.6
```

- Installieren Sie es in Ihr Profil:

```
$ ./pre-inst-env guix package --install ruby@2.1
```

- Prüfen Sie auf häufige Fehler:

```
$ ./pre-inst-env guix lint ruby@2.1
```

Guix ist bestrebt, einen hohen Standard an seine Pakete anzusetzen. Wenn Sie Beiträge zum Guix-Projekt leisten,

- schreiben Sie Ihren Code im Stil von Guix (siehe Abschnitt “Programmierstil” in *Referenzhandbuch zu GNU Guix*)
- und schauen Sie sich die Kontrollliste aus dem Handbuch (siehe Abschnitt “Einreichen von Patches” in *Referenzhandbuch zu GNU Guix*) noch einmal an.

Sobald Sie mit dem Ergebnis zufrieden sind, freuen wir uns, wenn Sie Ihren Beitrag an uns schicken, damit wir ihn in Guix aufnehmen. Dieser Prozess wird auch im Handbuch beschrieben (siehe Abschnitt “Mitwirken” in *Referenzhandbuch zu GNU Guix*)<.

Es handelt sich um eine gemeinschaftliche Arbeit, je mehr also mitmachen, desto besser wird Guix!

2.1.3 Erweitertes Beispiel

Einfacher als obiges Hallo-Welt-Beispiel wird es nicht. Pakete können auch komplexer als das sein und Guix eignet sich für fortgeschrittenere Szenarien. Schauen wir uns ein anderes, umfangreicheres Paket an (leicht modifiziert gegenüber Guix’ Quellcode):

```
(define-module (gnu packages version-control)
  #:use-module ((guix licenses) #:prefix license:)
  #:use-module (guix utils)
  #:use-module (guix packages)
  #:use-module (guix git-download)
  #:use-module (guix build-system cmake)
  #:use-module (gnu packages compression)
  #:use-module (gnu packages pkg-config)
  #:use-module (gnu packages python)
  #:use-module (gnu packages ssh)
  #:use-module (gnu packages tls)
  #:use-module (gnu packages web))

(define-public my-libgit2
  (let ((commit "e98d0a37c93574d2c6107bf7f31140b548c6a7bf")
        (revision "1"))
    (package
      (name "my-libgit2")
      (version (git-version "0.26.6" revision commit))
      (source (origin
                (method git-fetch)
```

```
                (uri (git-reference
                      (url "https://github.com/libgit2/libgit2/")
                      (commit commit)))
                (file-name (git-file-name name version))
                (sha256
                 (base32
                  "17pjvprmdrx4h6bb1hhc98w9qi6ki7yl57f090n9kbhswxqfs7s3"))
                (patches (search-patches "libgit2-mtime-0.patch"))
                (modules '((guix build utils)))
                ;; Gebündelte Software entfernen wir.
                (snippet '(delete-file-recursively "deps"))))
        (build-system cmake-build-system)
        (outputs '("out" "debug"))
        (arguments
         `(#:tests? #true                          ;Testkatalog ausführen (wie ohnehin vo
           #:configure-flags '("-DUSE_SHA1DC=ON") ;Kollisionserkennung für SHA-1 aktivie
           #:phases
           (modify-phases %standard-phases
             (add-after 'unpack 'fix-hardcoded-paths
               (lambda _
                 (substitute* "tests/repo/init.c"
                   (("#!/bin/sh") (string-append "#!" (which "sh"))))
                 (substitute* "tests/clar/fs.h"
                   (("/bin/cp") (which "cp"))
                   (("/bin/rm") (which "rm")))))
             ;; Ausführliche Testausgaben einschalten.
             (replace 'check
               (lambda* (#:key tests? #:allow-other-keys)
                 (when tests?
                   (invoke "./libgit2_clar" "-v" "-Q"))))
             (add-after 'unpack 'make-files-writable-for-tests
               (lambda _ (for-each make-file-writable (find-files "."))))))
        (inputs
         (list libssh2 http-parser python-wrapper))
        (native-inputs
         (list pkg-config))
        (propagated-inputs
         ;; Weil diese zwei Bibliotheken unter 'Requires.private' in libgit2.pc stehen.
         (list openssl zlib))
        (home-page "https://libgit2.github.com/")
        (synopsis "Library providing Git core methods")
        (description
         "Libgit2 is a portable, pure C implementation of the Git core methods
provided as a re-entrant linkable library with a solid API, allowing you to
write native speed custom Git applications in any language with bindings.")
        ;; Wir schreiben als Kommentar, dass es eigentlich die GPL2 mit Ausnahmen ist:
        ;; GPLv2 with linking exception
```

```
    (license license:gpl2))))
```

(In solchen Fällen, wo Sie nur ein paar wenige Felder einer Paketdefinition abändern wollen, wäre es wirklich besser, wenn Sie Vererbung einsetzen würden, statt alles abzuschreiben. Siehe unten.)

Reden wir über diese Felder im Detail.

2.1.3.1 `git-fetch`-Methode

Anders als die `url-fetch`-Methode erwartet `git-fetch` eine `git-reference`, welche ein Git-Repository und einen Commit entgegennimmt. Der Commit kann eine beliebige Art von Git-Referenz sein, z.B. ein Tag. Wenn die `version` also mit einem Tag versehen ist, kann sie einfach benutzt werden. Manchmal ist dem Tag ein Präfix `v` vorangestellt. In diesem Fall würden Sie `(commit (string-append "v" version))` schreiben.

Um sicherzustellen, dass der Quellcode aus dem Git-Repository in einem Verzeichnis mit nachvollziehbarem Namen landet, schreiben wir `(file-name (git-file-name name version))`.

Mit der Prozedur `git-version` kann die Version beim Paketieren eines bestimmten Commits eines Programms entsprechend den Richtlinien für Beiträge zu Guix (siehe Abschnitt "Versionsnummern" in *Referenzhandbuch zu GNU Guix*) abgeleitet werden.

Sie fragen, woher man den darin angegebenen `sha256`-Hash bekommt? Indem Sie `guix hash` auf einem Checkout des gewünschten Commits aufrufen, ungefähr so:

```
git clone https://github.com/libgit2/libgit2/
cd libgit2
git checkout v0.26.6
guix hash -rx .
```

`guix hash -rx` berechnet einen SHA256-Hash des gesamten Verzeichnisses, abgesehen vom `.git`-Unterverzeichnis (siehe Abschnitt "Aufruf von guix hash" in *Referenzhandbuch zu GNU Guix*).

In Zukunft wird `guix download` diese Schritte hoffentlich für Sie erledigen können, genau wie es das für normales Herunterladen macht.

2.1.3.2 Schnipsel

„Snippets", deutsch Schnipsel, sind mit z.B. `quote`-Zeichen maskierte, also nicht ausgewertete, Stücke Scheme-Code, mit denen der Quellcode gepatcht wird. Sie sind eine guixige Alternative zu traditionellen `.patch`-Dateien. Wegen der Maskierung werden sie erst dann ausgewertet, wenn sie an den Guix-Daemon zum Erstellen übergeben werden. Es kann so viele Schnipsel geben wie nötig.

In Schnipseln könnten zusätzliche Guile-Module benötigt werden. Diese können importiert werden, indem man sie im Feld `modules` angibt.

2.1.3.3 Eingaben

Es gibt 3 verschiedene Arten von Eingaben. Kurz gefasst:

native-inputs
: Sie werden zum Erstellen gebraucht, aber *nicht* zur Laufzeit – wenn Sie ein Paket als Substitut installieren, werden diese Eingaben nirgendwo installiert.

inputs Sie werden in den Store installiert, aber nicht in das Profil, und sie stehen beim Erstellen zur Verfügung.

propagated-inputs
Sie werden sowohl in den Store als auch ins Profil installiert und sind auch beim Erstellen verfügbar.

Siehe Abschnitt "„package"-Referenz" in *Referenzhandbuch zu GNU Guix* für mehr Details.

Der Unterschied zwischen den verschiedenen Eingaben ist wichtig: Wenn eine Abhängigkeit als `input` statt als `propagated-input` ausreicht, dann sollte sie auch so eingeordnet werden, sonst „verschmutzt" sie das Profil des Benutzers ohne guten Grund.

Wenn eine Nutzerin beispielsweise ein grafisches Programm installiert, das von einem Befehlszeilenwerkzeug abhängt, sie sich aber nur für den grafischen Teil interessiert, dann sollten wir sie nicht zur Installation des Befehlszeilenwerkzeugs in ihr Benutzerprofil zwingen. Um die Abhängigkeit sollte sich das Paket kümmern, nicht seine Benutzerin. Mit *Inputs* können wir Abhängigkeiten verwenden, wo sie gebraucht werden, ohne Nutzer zu belästigen, indem wir ausführbare Dateien (oder Bibliotheken) in deren Profil installieren.

Das Gleiche gilt für *native-inputs*: Wenn das Programm einmal installiert ist, können Abhängigkeiten zur Erstellungszeit gefahrlos dem Müllsammler anvertraut werden. Sie sind auch besser, wenn ein Substitut verfügbar ist, so dass nur die `inputs` und `propagated-inputs` heruntergeladen werden; `native-inputs` braucht niemand, der das Paket aus einem Substitut heraus installiert.

> **Anmerkung:** Vielleicht bemerken Sie hier und da Schnipsel, deren Paketeingaben recht anders geschrieben wurden, etwa so:
>
> ```
> ;; Der „alte Stil" von Eingaben.
> (inputs
>  `(("libssh2" ,libssh2)
>    ("http-parser" ,http-parser)
>    ("python" ,python-wrapper)))
> ```
>
> Früher hatte man Eingaben in diesem „alten Stil" geschrieben, wo jede Eingabe ausdrücklich mit einer Bezeichnung (als Zeichenkette) assoziiert wurde. Er wird immer noch unterstützt, aber wir empfehlen, dass Sie stattdessen im weiter oben gezeigten Stil schreiben. Siehe Abschnitt "„package"-Referenz" in *Referenzhandbuch zu GNU Guix* für mehr Informationen.

2.1.3.4 Ausgaben

Genau wie ein Paket mehrere Eingaben haben kann, kann es auch mehrere Ausgaben haben.

Jede Ausgabe entspricht einem anderen Verzeichnis im Store.

Die Benutzerin kann sich entscheiden, welche Ausgabe sie installieren will; so spart sie Platz auf dem Datenträger und verschmutzt ihr Benutzerprofil nicht mit unerwünschten ausführbaren Dateien oder Bibliotheken.

Nach Ausgaben zu trennen ist optional. Wenn Sie kein `outputs`-Feld schreiben, heißt die standardmäßige und einzige Ausgabe (also das ganze Paket) schlicht `"out"`.

Typische Namen für getrennte Ausgaben sind `debug` und `doc`.

Es wird empfohlen, getrennte Ausgaben nur dann anzubieten, wenn Sie gezeigt haben, dass es sich lohnt, d.h. wenn die Ausgabengröße signifikant ist (vergleichen Sie sie mittels `guix size`) oder das Paket modular aufgebaut ist.

2.1.3.5 Argumente ans Erstellungssystem

`arguments` ist eine Liste aus Schlüsselwort-Wert-Paaren (eine „keyword-value list“), mit denen der Erstellungsprozess konfiguriert wird.

Das einfachste Argument `#:tests?` kann man benutzen, um den Testkatalog bei der Erstellung des Pakets nicht zu prüfen. Das braucht man meistens dann, wenn das Paket überhaupt keinen Testkatalog hat. Wir empfehlen sehr, den Testkatalog zu benutzen, wenn es einen gibt.

Ein anderes häufiges Argument ist `:make-flags`, was eine Liste an den `make`-Aufruf anzuhängender Befehlszeilenargumente festlegt, so wie Sie sie auf der Befehlszeile angeben würden. Zum Beispiel werden die folgenden `:make-flags`

```
#:make-flags (list (string-append "prefix=" (assoc-ref %outputs "out"))
                   "CC=gcc")
```

übersetzt zu

```
$ make CC=gcc prefix=/gnu/store/...-<out>
```

Dadurch wird als C-Compiler `gcc` verwendet und als `prefix`-Variable (das Installationsverzeichnis in der Sprechweise von Make) wird `(assoc-ref %outputs "out")` verwendet, also eine globale Variable der Erstellungsschicht, die auf das Zielverzeichnis im Store verweist (so etwas wie `/gnu/store/...-my-libgit2-20180408`).

Auf gleiche Art kann man auch die Befehlszeilenoptionen für configure festlegen:

```
#:configure-flags '("-DUSE_SHA1DC=ON")
```

Die Variable `%build-inputs` wird auch in diesem Sichtbarkeitsbereich erzeugt. Es handelt sich um eine assoziative Liste, die von den Namen der Eingaben auf ihre Verzeichnisse im Store abbildet.

Das `phases`-Schlüsselwort listet der Reihe nach die vom Erstellungssystem durchgeführten Schritte auf. Zu den üblichen Phasen gehören `unpack`, `configure`, `build`, `install` und `check`. Um mehr über diese Phasen zu lernen, müssen Sie sich die Definition des zugehörigen Erstellungssystems in ‘`$GUIX_CHECKOUT/guix/build/gnu-build-system.scm`’ anschauen:

```
(define %standard-phases
  ;; Standard build phases, as a list of symbol/procedure pairs.
  (let-syntax ((phases (syntax-rules ()
                         ((_ p ...) `((p . ,p) ...)))))
    (phases set-SOURCE-DATE-EPOCH set-paths install-locale unpack
            bootstrap
            patch-usr-bin-file
            patch-source-shebangs configure patch-generated-file-shebangs
            build check install
            patch-shebangs strip
            validate-runpath
            validate-documentation-location
```

```
            delete-info-dir-file
            patch-dot-desktop-files
            install-license-files
            reset-gzip-timestamps
            compress-documentation)))
```

Alternativ auf einer REPL:

```
(add-to-load-path "/path/to/guix/checkout")
,use (guix build gnu-build-system)
(map first %standard-phases)
⇒ (set-SOURCE-DATE-EPOCH set-paths install-locale unpack bootstrap patch-usr-bin-file
```

Wenn Sie mehr darüber wissen wollen, was in diesen Phasen passiert, schauen Sie in den jeweiligen Prozeduren.

Beispielsweise sieht momentan, als dies hier geschrieben wurde, die Definition von `unpack` für das GNU-Erstellungssystem so aus:

```
(define* (unpack #:key source #:allow-other-keys)
  "Unpack SOURCE in the working directory, and change directory within the
source.  When SOURCE is a directory, copy it in a sub-directory of the current
working directory."
  (if (file-is-directory? source)
      (begin
        (mkdir "source")
        (chdir "source")

        ;; Preserve timestamps (set to the Epoch) on the copied tree so that
        ;; things work deterministically.
        (copy-recursively source "."
                          #:keep-mtime? #true))
      (begin
        (if (string-suffix? ".zip" source)
            (invoke "unzip" source)
            (invoke "tar" "xvf" source))
        (chdir (first-subdirectory ".")))))
  #true)
```

Beachten Sie den Aufruf von `chdir`: Damit wird das Arbeitsverzeichnis zu demjenigen gewechselt, wohin die Quelldateien entpackt wurden. In jeder Phase nach `unpack` dient also das Verzeichnis mit den Quelldateien als Arbeitsverzeichnis. Deswegen können wir direkt mit den Quelldateien arbeiten, zumindest solange keine spätere Phase das Arbeitsverzeichnis woandershin wechselt.

Die Liste der `%standard-phases` des Erstellungssystems ändern wir mit Hilfe des `modify-phases`-Makros über eine Liste von Änderungen. Sie kann folgende Formen haben:

- (`add-before` *`Phase neue-Phase Prozedur`*): *Prozedur* unter dem Namen *neue-Phase* vor *Phase* ausführen.
- (`add-after` *`Phase neue-Phase Prozedur`*): Genauso, aber danach.
- (`replace` *`Phase Prozedur`*).

- `(delete Phase)`.

Die *Prozedur* unterstützt die Schlüsselwortargumente `inputs` und `outputs`. Jede Eingabe (ob sie *native*, *propagated* oder nichts davon ist) und jedes Ausgabeverzeichnis ist in diesen Variablen mit dem jeweiligen Namen assoziiert. `(assoc-ref outputs "out")` ist also das Store-Verzeichnis der Hauptausgabe des Pakets. Eine Phasenprozedur kann so aussehen:

```
(lambda* (#:key inputs outputs #:allow-other-keys)
  (let ((bash-directory (assoc-ref inputs "bash"))
        (output-directory (assoc-ref outputs "out"))
        (doc-directory (assoc-ref outputs "doc")))
    ;; ...
    #true))
```

Die Prozedur muss bei Erfolg `#true` zurückliefern. Auf den Rückgabewert des letzten Ausdrucks, mit dem die Phase angepasst wurde, kann man sich nicht verlassen, weil es keine Garantie gibt, dass der Rückgabewert `#true` sein wird. Deswegen schreiben wir dahinter `#true`, damit bei erfolgreicher Ausführung der richtige Wert geliefert wird.

2.1.3.6 Code-Staging

Aufmerksame Leser könnten die Syntax mit `quasiquote` und Komma im Argumentefeld bemerkt haben. Tatsächlich sollte der Erstellungscode in der Paketdeklaration *nicht* auf Client-Seite ausgeführt werden, sondern erst, wenn er an den Guix-Daemon übergeben worden ist. Der Mechanismus, über den Code zwischen zwei laufenden Prozessen weitergegeben wird, nennen wir Code-Staging (`https://arxiv.org/abs/1709.00833`).

2.1.3.7 Hilfsfunktionen

Beim Anpassen der `phases` müssen wir oft Code schreiben, der analog zu den äquivalenten Systemaufrufen funktioniert (`make`, `mkdir`, `cp`, etc.), welche in regulären „Unix-artigen“ Installationen oft benutzt werden.

Manche, wie `chmod`, sind Teil von Guile. Siehe das *Referenzhandbuch zu Guile* für eine vollständige Liste.

Guix stellt zusätzliche Hilfsfunktionen zur Verfügung, die bei der Paketverwaltung besonders praktisch sind.

Manche dieser Funktionalitäten finden Sie in '`$GUIX_CHECKOUT/guix/guix/build/utils.scm`'. Die meisten spiegeln das Verhalten traditioneller Unix-Systembefehle wider:

`which` Das Gleiche wie der '`which`'-Systembefehl.

`find-files`
: Wie der '`find`' Systembefehl.

`mkdir-p` Wie '`mkdir -p`', das Elternverzeichnisse erzeugt, wenn nötig.

`install-file`
: Ähnlich wie '`install`' beim Installieren einer Datei in ein (nicht unbedingt existierendes) Verzeichnis. Guile kennt `copy-file`, das wie '`cp`' funktioniert.

`copy-recursively`
: Wie '`cp -r`'.

`delete-file-recursively`
: Wie '`rm -rf`'.

`invoke`
: Eine ausführbare Datei ausführen. Man sollte es benutzen und nicht `system*`.

`with-directory-excursion`
: Den Rumpf in einem anderen Arbeitsverzeichnis ausführen und danach wieder in das vorherige Arbeitsverzeichnis wechseln.

`substitute*`
: Eine „`sed`-artige“ Funktion.

Siehe Abschnitt "Werkzeuge zur Erstellung" in *Referenzhandbuch zu GNU Guix* für mehr Informationen zu diesen Werkzeugen.

2.1.3.8 Modulpräfix

Die Lizenz in unserem letzten Beispiel braucht ein Präfix. Der Grund liegt darin, wie das `license`-Modul importiert worden ist, nämlich `#:use-module ((guix licenses) #:prefix license:)`. Der Importmechanismus von Guile-Modulen (siehe Abschnitt "Using Guile Modules" in *Referenzhandbuch zu Guile*) gibt Benutzern die volle Kontrolle über Namensräume. Man braucht sie, um Konflikte zu lösen, z.B. zwischen der '`zlib`'-Variablen aus '`licenses.scm`' (dieser Wert repräsentiert eine *Softwarelizenz*) und der '`zlib`'-Variablen aus '`compression.scm`' (ein Wert, der für ein *Paket* steht).

2.1.4 Andere Erstellungssysteme

Was wir bisher gesehen haben reicht für die meisten Pakete aus, die als Erstellungssystem etwas anderes als `trivial-build-system` verwenden. Letzteres automatisiert gar nichts und überlässt es Ihnen, alles zur Erstellung manuell festzulegen. Das kann einen noch mehr beanspruchen und wir beschreiben es hier zurzeit nicht, aber glücklicherweise braucht man dieses System auch nur in seltenen Fällen.

Bei anderen Erstellungssystemen wie ASDF, Emacs, Perl, Ruby und vielen anderen ist der Prozess sehr ähnlich zum GNU-Erstellungssystem abgesehen von ein paar speziellen Argumenten.

Siehe Abschnitt "Erstellungssysteme" in *Referenzhandbuch zu GNU Guix*, für mehr Informationen über Erstellungssysteme, oder den Quellcode in den Verzeichnissen '`$GUIX_CHECKOUT/guix/build`' und '`$GUIX_CHECKOUT/guix/build-system`'.

2.1.5 Programmierbare und automatisierte Paketdefinition

Wir können es nicht oft genug wiederholen: Eine Allzweck-Programmiersprache zur Hand zu haben macht Dinge möglich, die traditionelle Paketverwaltung weit übersteigen.

Wir können uns das anhand Guix' großartiger Funktionalitäten klarmachen!

2.1.5.1 Rekursive Importer

Sie könnten feststellen, dass manche Erstellungssysteme gut genug sind und nichts weiter zu tun bleibt, um ein Paket zu verfassen. Das Paketeschreiben kann so monoton werden und man wird dessen bald überdrüssig. Eine Daseinsberechtigung von Rechnern ist, Menschen bei solch langweiligen Aufgaben zu ersetzen. Lasst uns also Guix die Sache erledigen: Wir

lassen uns die Paketdefinition eines R-Pakets mit den Informationen aus CRAN holen (was zu anderen ausgegeben wird, haben wir im Folgenden weggelassen):

```
$ guix import cran --recursive walrus

(define-public r-mc2d
    ; ...
    (license gpl2+)))

(define-public r-jmvcore
    ; ...
    (license gpl2+)))

(define-public r-wrs2
    ; ...
    (license gpl3)))

(define-public r-walrus
  (package
    (name "r-walrus")
    (version "1.0.3")
    (source
      (origin
        (method url-fetch)
        (uri (cran-uri "walrus" version))
        (sha256
          (base32
            "1nk2glcvy4hyksl5ipq2mz8jy4fss90hx6cq98m3w96kzjni6jjj"))))
    (build-system r-build-system)
    (propagated-inputs
      (list r-ggplot2 r-jmvcore r-r6 r-wrs2))
    (home-page "https://github.com/jamovi/walrus")
    (synopsis "Robust Statistical Methods")
    (description
      "This package provides a toolbox of common robust statistical
tests, including robust descriptives, robust t-tests, and robust ANOVA.
It is also available as a module for 'jamovi' (see
<https://www.jamovi.org> for more information).  Walrus is based on the
WRS2 package by Patrick Mair, which is in turn based on the scripts and
work of Rand Wilcox.  These analyses are described in depth in the book
'Introduction to Robust Estimation & Hypothesis Testing'.")
    (license gpl3)))
```

Der rekursive Importer wird keine Pakete importieren, für die es in Guix bereits eine Paketdefinition gibt, außer dem Paket, mit dem er anfängt.

Nicht für alle Anwendungen können auf diesem Weg Pakete erzeugt werden, nur für jene, die auf ausgewählten Systemen aufbauen. Im Handbuch können Sie Informationen über die

vollständige Liste aller Importer bekommen (siehe Abschnitt “Aufruf von guix import” in *Referenzhandbuch zu GNU Guix*).

2.1.5.2 Automatisch aktualisieren

Guix ist klug genug, um verfügbare Aktualisierungen auf bekannten Systemen zu erkennen. Es kann über veraltete Paketdefinitionen Bericht erstatten, wenn man dies eingibt:

```
$ guix refresh hello
```

In den meisten Fällen muss man zur Aktualisierung auf eine neuere Version wenig mehr tun, als die Versionsnummer und die Prüfsumme ändern. Auch das kann mit Guix automatisiert werden:

```
$ guix refresh hello --update
```

2.1.5.3 Vererbung

Wenn Sie anfangen, bestehende Paketdefinitionen anzuschauen, könnte es Ihnen auffallen, dass viele von ihnen über ein `inherit`-Feld verfügen.

```
(define-public adwaita-icon-theme
  (package (inherit gnome-icon-theme)
    (name "adwaita-icon-theme")
    (version "3.26.1")
    (source (origin
              (method url-fetch)
              (uri (string-append "mirror://gnome/sources/" name "/"
                                  (version-major+minor version) "/"
                                  name "-" version ".tar.xz"))
              (sha256
               (base32
                "17fpahgh5dyckgz7rwqvzgnhx53cx9kr2xw0szprc6bnqy977fi8"))))
    (native-inputs (list `(,gtk+ \"bin\")))))
```

Alle *nicht* aufgeführten Felder werden vom Elternpaket geerbt. Das ist ziemlich praktisch, um alternative Pakete zu erzeugen, zum Beispiel solche mit geänderten Quellorten, Versionen oder Kompilierungsoptionen.

2.1.6 Hilfe bekommen

Leider ist es für manche Anwendungen schwierig, Pakete zu schreiben. Manchmal brauchen sie einen Patch, um mit vom Standard abweichenden Dateisystemhierarchien klarzukommen, wie sie der Store erforderlich macht. Manchmal funktionieren die Tests nicht richtig. (Man kann sie überspringen, aber man sollte nicht.) Ein andermal ist das sich ergebende Paket nicht reproduzierbar.

Wenn Sie nicht weiterkommen, weil Sie nicht wissen, wie Sie ein Problem beim Paketschreiben lösen können, dann zögern Sie nicht, die Gemeinde um Hilfe zu bitten.

Siehe die Homepage von Guix (`https://www.gnu.org/software/guix/contact/`) für Informationen, welche Mailing-Listen, IRC-Kanäle etc. bereitstehen.

2.1.7 Schlusswort

Diese Anleitung hat einen Überblick über die fortgeschrittene Paketverwaltung gegeben, die Guix vorweist. Zu diesem Zeitpunkt haben wir diese Einführung größtenteils auf das `gnu-build-system` eingeschränkt, was eine zentrale Abstraktionsschicht darstellt, auf der weitere Abstraktionen aufbauen.

Wie geht es nun weiter? Als Nächstes müssten wir das Erstellungssystem in seine Bestandteile zerlegen, um einen Einblick ganz ohne Abstraktionen zu bekommen. Das bedeutet, wir müssten das `trivial-build-system` analysieren. Dadurch sollte ein gründliches Verständnis des Prozesses vermittelt werden, bevor wir höher entwickelte Paketierungstechniken und Randfälle untersuchen.

Andere Funktionalitäten, die es wert sind, erkundet zu werden, sind Guix' Funktionalitäten zum interaktiven Editieren und zur Fehlersuche, die die REPL von Guile darbietet.

Diese eindrucksvollen Funktionalitäten sind völlig optional und können warten; jetzt ist die Zeit für eine wohlverdiente Pause. Mit dem Wissen, in das wir hier eingeführt haben, sollten Sie für das Paketieren vieler Programme gut gerüstet sein. Sie können gleich anfangen und hoffentlich bekommen wir bald Ihre Beiträge zu sehen!

2.1.8 Literaturverzeichnis

- Die Paketreferenz im Handbuch (`https://guix.gnu.org/manual/de/html_node/Pakete-definieren.html`)
- Pjotr's Hacking-Leitfaden für GNU Guix (`https://gitlab.com/pjotrp/guix-notes/blob/master/HACKING.org`)
- „GNU Guix: Package without a scheme!“ (`https://www.gnu.org/software/guix/guix-ghm-andreas-20130823.pdf`) von Andreas Enge

3 Systemkonfiguration

Guix stellt eine flexible Sprache bereit, um Ihr „Guix System“ auf deklarative Weise zu konfigurieren. Diese Flexibilität kann einen manchmal überwältigen. Dieses Kapitel hat den Zweck, einige fortgeschrittene Konfigurationskonzepte vorzuzeigen.

Siehe Abschnitt “Systemkonfiguration” in *Referenzhandbuch zu GNU Guix* für eine vollständige Referenz.

3.1 Automatisch an virtueller Konsole anmelden

Im Guix-Handbuch wird erklärt, wie man ein Benutzerkonto automatisch auf *allen* TTYs anmelden lassen kann (siehe Abschnitt “auto-login to TTY” in *Referenzhandbuch zu GNU Guix*), aber vielleicht wäre es Ihnen lieber, ein Benutzerkonto an genau einem TTY anzumelden und die anderen TTYs so zu konfigurieren, dass entweder andere Benutzer oder gar niemand angemeldet wird. Beachten Sie, dass man auf jedem TTY automatisch jemanden anmelden kann, aber meistens will man `tty1` in Ruhe lassen, weil dorthin nach Voreinstellung Warnungs- und Fehlerprotokolle ausgegeben werden.

Um eine Benutzerin auf einem einzelnen TTY automatisch anzumelden, schreibt man:

```
(define (auto-login-to-tty config tty user)
  (if (string=? tty (mingetty-configuration-tty config))
        (mingetty-configuration
         (inherit config)
         (auto-login user))
        config))

(define %my-services
  (modify-services %base-services
    ;; ...
    (mingetty-service-type config =>
                           (auto-login-to-tty
                            config "tty3" "alice"))))

(operating-system
  ;; ...
  (services %my-services))
```

Mit Hilfe von `compose` (siehe Abschnitt “Higher-Order Functions” in *das Referenzhandbuch zu GNU Guile*) kann man etwas wie `auto-login-to-tty` mehrfach angeben, um mehrere Nutzerkonten auf verschiedenen TTYs anzumelden.

Zum Schluss aber noch eine Warnung. Wenn Sie jemanden auf einem TTY automatisch anmelden lassen, kann jeder einfach Ihren Rechner anschalten und dann Befehle in deren Namen ausführen. Haben Sie Ihr Wurzeldateisystem auf einer verschlüsselten Partition, müsste man dafür erst einmal das Passwort eingeben, wenn das System startet. Dann wäre automatisches Anmelden vielleicht bequem.

3.2 Den Kernel anpassen

Im Kern ist Guix eine quellcodebasierte Distribution mit Substituten (siehe Abschnitt "Substitute" in *Referenzhandbuch zu GNU Guix*), daher ist das Erstellen von Paketen aus ihrem Quellcode heraus genauso vorgesehen wie die normale Installation und Aktualisierung von Paketen. Von diesem Standpunkt ist es sinnvoll, zu versuchen, den Zeitaufwand für das Kompilieren von Paketen zu senken, und kürzliche Neuerungen sowie Verbesserungen beim Erstellen und Verteilen von Substituten bleiben ein Diskussionsthema innerhalb von Guix.

Der Kernel braucht zwar keine übermäßigen Mengen an Arbeitsspeicher beim Erstellen, jedoch jede Menge Zeit auf einer durchschnittlichen Maschine. Die offizielle Konfiguration des Kernels umfasst, wie bei anderen GNU/Linux-Distributionen auch, besser zu viel als zu wenig. Das ist der eigentliche Grund, warum seine Erstellung so lange dauert, wenn man den Kernel aus dem Quellcode heraus erstellt.

Man kann den Linux-Kernel jedoch auch als ganz normales Paket beschreiben, das genau wie jedes andere Paket angepasst werden kann. Der Vorgang ist ein klein wenig anders, aber das liegt hauptsächlich an der Art, wie die Paketdefinition geschrieben ist.

Die `linux-libre`-Kernelpaketdefinition ist tatsächlich eine Prozedur, die ein Paket liefert.

```
(define* (make-linux-libre* version gnu-revision source supported-systems
                            #:key
                            (extra-version #f)
                            ;; A function that takes an arch and a variant.
                            ;; See kernel-config for an example.
                            (configuration-file #f)
                            (defconfig "defconfig")
                            (extra-options %default-extra-linux-options))
  ...)
```

Das momentane `linux-libre`-Paket zielt ab auf die 5.15.x-Serie und ist wie folgt deklariert:

```
(define-public linux-libre-5.15
  (make-linux-libre* linux-libre-5.15-version
                     linux-libre-5.15-gnu-revision
                     linux-libre-5.15-source
                     '("x86_64-linux" "i686-linux" "armhf-linux"
                       "aarch64-linux" "riscv64-linux")
                     #:configuration-file kernel-config))
```

Alle Schlüssel, denen kein Wert zugewiesen wird, erben ihren Vorgabewert von der Definition von `make-linux-libre`. Wenn Sie die beiden Schnipsel oben vergleichen, ist anzumerken, dass sich der Code-Kommentar in ersterem auf `#:configuration-file` bezieht. Deswegen ist es nicht so leicht, aus der Definition heraus eine eigene Kernel-Konfiguration anhand der Definition zu schreiben, aber keine Sorge, es gibt andere Möglichkeiten, um mit dem zu arbeiten, was uns gegeben wurde.

Es gibt zwei Möglichkeiten, einen Kernel mit eigener Kernel-Konfiguration zu erzeugen. Die erste ist, eine normale `.config`-Datei als native Eingabe zu unserem angepassten Kernel hinzuzufügen. Im Folgenden sehen Sie ein Schnipsel aus der angepassten `'configure`-Phase der `make-linux-libre`-Paketdefinition:

```
(let ((build  (assoc-ref %standard-phases 'build))
      (config (assoc-ref (or native-inputs inputs) "kconfig")))

  ;; Use a custom kernel configuration file or a default
  ;; configuration file.
  (if config
      (begin
        (copy-file config ".config")
        (chmod ".config" #o666))
      (invoke "make" ,defconfig)))
```

Nun folgt ein Beispiel-Kernel-Paket. Das `linux-libre`-Paket ist nicht anders als andere Pakete und man kann von ihm erben und seine Felder ersetzen wie bei jedem anderen Paket.

```
(define-public linux-libre/E2140
  (package
    (inherit linux-libre)
    (native-inputs
     `(("kconfig" ,(local-file "E2140.config"))
      ,@(alist-delete "kconfig"
                      (package-native-inputs linux-libre))))))
```

Im selben Verzeichnis wie die Datei, die `linux-libre-E2140` definiert, befindet sich noch eine Datei namens `E2140.config`, bei der es sich um eine richtige Kernel-Konfigurationsdatei handelt. Das Schlüsselwort `defconfig` von `make-linux-libre` wird hier leer gelassen, so dass die einzige Kernel-Konfiguration im Paket die im `native-inputs`-Feld ist.

Die zweite Möglichkeit, einen eigenen Kernel zu erzeugen, ist, einen neuen Wert an das `extra-options`-Schlüsselwort der `make-linux-libre`-Prozedur zu übergeben. Das `extra-options`-Schlüsselwort wird zusammen mit einer anderen, direkt darunter definierten Funktion benutzt:

```
(define %default-extra-linux-options
  `(;; https://lists.gnu.org/archive/html/guix-devel/2014-04/msg00039.html
    ("CONFIG_DEVPTS_MULTIPLE_INSTANCES" . #true)
    ;; Modules required for initrd:
    ("CONFIG_NET_9P" . m)
    ("CONFIG_NET_9P_VIRTIO" . m)
    ("CONFIG_VIRTIO_BLK" . m)
    ("CONFIG_VIRTIO_NET" . m)
    ("CONFIG_VIRTIO_PCI" . m)
    ("CONFIG_VIRTIO_BALLOON" . m)
    ("CONFIG_VIRTIO_MMIO" . m)
    ("CONFIG_FUSE_FS" . m)
    ("CONFIG_CIFS" . m)
    ("CONFIG_9P_FS" . m)))

(define (config->string options)
  (string-join (map (match-lambda
                      ((option . 'm)
```

```
                        (string-append option "=m"))
                       ((option . #true)
                        (string-append option "=y"))
                       ((option . #false)
                        (string-append option "=n")))
                     options)
                "\n"))
```

Und im eigenen configure-Skript des „make-linux-libre“-Pakets:

```
;; Appending works even when the option wasn't in the
;; file.  The last one prevails if duplicated.
(let ((port (open-file ".config" "a"))
      (extra-configuration ,(config->string extra-options)))
  (display extra-configuration port)
  (close-port port))

(invoke "make" "oldconfig")
```

Indem wir also kein „configuration-file“ mitgeben, ist `.config` anfangs leer und danach schreiben wir dort die Sammlung der gewünschten Optionen („Flags“) hinein. Hier ist noch ein eigener Kernel:

```
(define %macbook41-full-config
  (append %macbook41-config-options
          %file-systems
          %efi-support
          %emulation
          (@@ (gnu packages linux) %default-extra-linux-options)))

(define-public linux-libre-macbook41
  ;; XXX: Auf die interne 'make-linux-libre*'-Prozedur zugreifen, welche privat
  ;; ist und nicht exportiert, des Weiteren kann sie sich in Zukunft ändern.
  ((@@ (gnu packages linux) make-linux-libre*)
   (@@ (gnu packages linux) linux-libre-version)
   (@@ (gnu packages linux) linux-libre-gnu-revision)
   (@@ (gnu packages linux) linux-libre-source)
   '("x86_64-linux")
   #:extra-version "macbook41"
   #:extra-options %macbook41-config-options))
```

Im obigen Beispiel ist `%file-systems` eine Sammlung solcher „Flags“, mit denen Unterstützung für verschiedene Dateisysteme aktiviert wird, `%efi-support` aktiviert Unterstützung für EFI und `%emulation` ermöglicht es einer x86_64-linux-Maschine, auch im 32-Bit-Modus zu arbeiten. Die `%default-extra-linux-options` sind die oben zitierten, die wieder hinzugefügt werden mussten, weil sie durch das `extra-options`-Schlüsselwort ersetzt worden waren.

All das klingt machbar, aber woher weiß man überhaupt, welche Module für ein bestimmtes System nötig sind? Es gibt zwei hilfreiche Anlaufstellen, zum einen das Gentoo-Handbuch (`https://wiki.gentoo.org/wiki/Handbook:AMD64/Installation/`

`Kernel`), zum anderen die Dokumentation des Kernels selbst (`https://www.kernel.org/doc/html/latest/admin-guide/README.html?highlight=localmodconfig`). Aus der Kernel-Dokumentation erfahren wir, dass `make localmodconfig` der Befehl sein könnte, den wir wollen.

Um `make localmodconfig` auch tatsächlich ausführen zu können, müssen wir zunächst den Quellcode des Kernels holen und entpacken:

```
tar xf $(guix build linux-libre --source)
```

Sobald wir im Verzeichnis mit dem Quellcode sind, führen Sie `touch .config` aus, um mit einer ersten, leeren `.config` anzufangen. `make localmodconfig` funktioniert so, dass angeschaut wird, was bereits in Ihrer `.config` steht, und Ihnen mitgeteilt wird, was Ihnen noch fehlt. Wenn die Datei leer bleibt, fehlt eben alles. Der nächste Schritt ist, das hier auszuführen:

```
guix shell -D linux-libre -- make localmodconfig
```

und uns die Ausgabe davon anzuschauen. Beachten Sie, dass die `.config`-Datei noch immer leer ist. Die Ausgabe enthält im Allgemeinen zwei Arten von Warnungen. Am Anfang der ersten steht „WARNING“ und in unserem Fall können wir sie tatsächlich ignorieren. Bei der zweiten heißt es:

```
module pcspkr did not have configs CONFIG_INPUT_PCSPKR
```

Für jede solche Zeile kopieren Sie den `CONFIG_XXXX_XXXX`-Teil in die `.config` im selben Verzeichnis und hängen `=m` an, damit es am Ende so aussieht:

```
CONFIG_INPUT_PCSPKR=m
CONFIG_VIRTIO=m
```

Nachdem Sie alle Konfigurationsoptionen kopiert haben, führen Sie noch einmal `make localmodconfig` aus, um sicherzugehen, dass es keine Ausgaben mehr gibt, deren erstes Wort „module“ ist. Zusätzlich zu diesen maschinenspezifischen Modulen gibt es noch ein paar mehr, die Sie auch brauchen. `CONFIG_MODULES` brauchen Sie, damit Sie Module getrennt erstellen und laden können und nicht alles im Kernel eingebaut sein muss. Sie brauchen auch `CONFIG_BLK_DEV_SD`, um von Festplatten lesen zu können. Möglicherweise gibt es auch sonst noch Module, die Sie brauchen werden.

Die Absicht hinter dem hier Niedergeschriebenen ist *nicht*, eine Anleitung zum Konfigurieren eines eigenen Kernels zu sein. Wenn Sie also vorhaben, den Kernel an Ihre ganz eigenen Bedürfnisse anzupassen, werden Sie in anderen Anleitungen fündig.

Die zweite Möglichkeit, die Kernel-Konfiguration einzurichten, benutzt mehr von Guix' Funktionalitäten und sie ermöglicht es Ihnen, Gemeinsamkeiten von Konfigurationen zwischen verschiedenen Kernels zu teilen. Zum Beispiel wird eine Reihe von EFI-Konfigurationsoptionen von allen Maschinen, die EFI benutzen, benötigt. Wahrscheinlich haben all diese Kernels eine Schnittmenge zu unterstützender Dateisysteme. Indem Sie Variable benutzen, können Sie leicht auf einen Schlag sehen, welche Funktionalitäten aktiviert sind, und gleichzeitig sicherstellen, dass Ihnen nicht Funktionalitäten des einen Kernels im anderen fehlen.

Was wir hierbei nicht erläutert haben, ist, wie Guix' initrd und dessen Anpassung funktioniert. Wahrscheinlich werden Sie auf einer Maschine mit eigenem Kernel die initrd verändern müssen, weil sonst versucht wird, bestimmte Module in die initrd einzubinden, die Sie gar nicht erstellen haben lassen.

3.3 Die Image-Schnittstelle von Guix System

In der Vergangenheit drehte sich in Guix System alles um eine `operating-system`-Struktur. So eine Struktur enthält vielerlei Felder vom Bootloader und der Deklaration des Kernels bis hin zu den Diensten, die installiert werden sollen.

Aber je nach Zielmaschine – diese kann alles von einer normalen `x86_64`-Maschine sein bis zu einem kleinen ARM-Einplatinenrechner wie dem Pine64 –, können die für ein Abbild geltenden Einschränkungen sehr unterschiedlich sein. Die Hardwarehersteller ordnen verschiedene Abbildformate an mit unterschiedlich versetzten unterschiedlich großen Partitionen.

Um für jede dieser Arten von Maschinen geeignete Abbilder zu erzeugen, brauchen wir eine neue Abstraktion. Dieses Ziel verfolgen wir mit dem `image`-Verbund. Ein Verbundsobjekt enthält alle nötigen Informationen, um daraus ein eigenständiges Abbild auf die Zielmaschine zu bringen. Es ist direkt startfähig von jeder solchen Zielmaschine.

```
(define-record-type* <image>
  image make-image
  image?
  (name               image-name ;Symbol
                      (default #f))
  (format             image-format) ;Symbol
  (target             image-target
                      (default #f))
  (size               image-size  ;Größe in Bytes als ganze Zahl
                      (default 'guess)) ;Vorgabe: automatisch bestimmen
  (operating-system   image-operating-system  ;<operating-system>
                      (default #f))
  (partitions         image-partitions ;Liste von <partition>
                      (default '()))
  (compression?       image-compression? ;Boolescher Ausdruck
                      (default #t))
  (volatile-root?     image-volatile-root? ;Boolescher Ausdruck
                      (default #t))
  (substitutable?     image-substitutable? ;Boolescher Ausdruck
                      (default #t)))
```

In einem Verbundsobjekt davon steht auch das zu instanziierende Betriebssystem (in `operating-system`). Im Feld `format` steht, was der Abbildtyp („image type“) ist, zum Beispiel `efi-raw`, `qcow2` oder `iso9660`. In Zukunft könnten die Möglichkeiten auf `docker` oder andere Abbildtypen erweitert werden.

Ein neues Verzeichnis im Guix-Quellbaum wurde Abbilddefinitionen gewidmet. Zurzeit gibt es vier Dateien darin:

- `gnu/system/images/hurd.scm`
- `gnu/system/images/pine64.scm`
- `gnu/system/images/novena.scm`
- `gnu/system/images/pinebook-pro.scm`

Schauen wir uns `pine64.scm` an. Es enthält die Variable `pine64-barebones-os`, bei der es sich um eine minimale Definition eines Betriebssystems handelt, die auf Platinen der Art **Pine A64 LTS** ausgerichtet ist.

```
(define pine64-barebones-os
  (operating-system
   (host-name "vignemale")
   (timezone "Europe/Paris")
   (locale "en_US.utf8")
   (bootloader (bootloader-configuration
                (bootloader u-boot-pine64-lts-bootloader)
                (targets '("/dev/vda"))))
   (initrd-modules '())
   (kernel linux-libre-arm64-generic)
   (file-systems (cons (file-system
                        (device (file-system-label "my-root"))
                        (mount-point "/")
                        (type "ext4"))
                       %base-file-systems))
   (services (cons (service agetty-service-type
                            (agetty-configuration
                             (extra-options '("-L")) ;kein Carrier Detect
                             (baud-rate "115200")
                             (term "vt100")
                             (tty "ttyS0")))
                   %base-services))))
```

Die Felder `kernel` und `bootloader` verweisen auf platinenspezifische Pakete.

Direkt darunter wird auch die Variable `pine64-image-type` definiert.

```
(define pine64-image-type
  (image-type
   (name 'pine64-raw)
   (constructor (cut image-with-os arm64-disk-image <>))))
```

Sie benutzt einen Verbundstyp, über den wir noch nicht gesprochen haben, den `image-type`-Verbund. Er ist wie folgt definiert:

```
(define-record-type* <image-type>
  image-type make-image-type
  image-type?
  (name           image-type-name) ;Symbol
  (constructor    image-type-constructor)) ;<operating-system> -> <image>
```

Der Hauptzweck dieses Verbunds ist, einer Prozedur, die ein `operating-system` in ein `image`-Abbild umwandelt, einen Namen zu geben. Um den Bedarf dafür nachzuvollziehen, schauen wir uns den Befehl an, mit dem ein Abbild aus einer `operating-system`-Konfigurationsdatei erzeugt wird:

```
guix system image my-os.scm
```

Dieser Befehl erwartet eine `operating-system`-Konfiguration, doch wie geben wir an, dass wir ein Abbild für einen Pine64-Rechner möchten? Wir müssen zusätzliche Informa-

tionen mitgeben, nämlich den Abbildtyp, `image-type`, indem wir die Befehlszeilenoption `--image-type` oder `-t` übergeben, und zwar so:

```
guix system image --image-type=pine64-raw my-os.scm
```

Der Parameter `image-type` verweist auf den oben definierten `pine64-image-type`. Dadurch wird die Prozedur `(cut image-with-os arm64-disk-image <>)` auf das in `my-os.scm` deklarierte `operating-system` angewandt und macht es zu einem `image`-Abbild.

Es ergibt sich ein Abbild wie dieses:

```
(image
 (format 'disk-image)
 (target "aarch64-linux-gnu")
 (operating-system my-os)
 (partitions
  (list (partition
         (inherit root-partition)
         (offset root-offset)))))
```

Das ist das Aggregat aus dem in `my-os.scm` definierten `operating-system` und dem `arm64-disk-image`-Verbundsobjekt.

Aber genug vom Scheme-Wahnsinn. Was nützt die Image-Schnittstelle dem Nutzer von Guix?

Sie können das ausführen:

```
mathieu@cervin:~$ guix system --list-image-types
Die verfügbaren Abbildtypen sind:

   - unmatched-raw
   - rock64-raw
   - pinebook-pro-raw
   - pine64-raw
   - novena-raw
   - hurd-raw
   - hurd-qcow2
   - qcow2
   - iso9660
   - uncompressed-iso9660
   - tarball
   - efi-raw
   - mbr-raw
   - docker
   - wsl2
   - raw-with-offset
   - efi32-raw
```

und indem Sie eine Betriebssystemkonfigurationsdatei mit einem auf `pine64-barebones-os` aufbauenden `operating-system` schreiben, können Sie Ihr Abbild nach Ihren Wünschen anpassen in einer Datei, sagen wir `my-pine-os.scm`:

```
(use-modules (gnu services linux)
```

```
              (gnu system images pine64))

(let ((base-os pine64-barebones-os))
  (operating-system
    (inherit base-os)
    (timezone "America/Indiana/Indianapolis")
    (services
     (cons
      (service earlyoom-service-type
               (earlyoom-configuration
                (prefer-regexp "icecat|chromium")))
      (operating-system-user-services base-os)))))
```

Führen Sie aus:

```
guix system image --image-type=pine64-raw my-pine-os.scm
```

oder

```
guix system image --image-type=hurd-raw my-hurd-os.scm
```

und Sie bekommen ein Abbild, das Sie direkt auf eine Festplatte kopieren und starten können.

Ohne irgendetwas an `my-hurd-os.scm` zu ändern, bewirkt ein Aufruf

```
guix system image --image-type=hurd-qcow2 my-hurd-os.scm
```

dass stattdessen ein Hurd-Abbild für QEMU erzeugt wird.

3.4 Sicherheitsschlüssel verwenden

Indem Sie Sicherheitsschlüssel einsetzen, können Sie sich besser absichern. Diese stellen eine zweite Bestätigung Ihrer Identität dar, die nicht leicht gestohlen oder kopiert werden kann, zumindest wenn der Angreifer nur Fernzugriff ergattert hat. Sicherheitsschlüssel sind etwas, was Sie haben, dagegen ist ein Geheimnis (eine Passphrase) etwas, was Sie wissen. So sinkt das Risiko, dass sich jemand Fremdes als Sie ausgibt.

Mit der nachfolgenden Beispielkonfiguration wird vorgeführt, wie Sie mit kleinstem Konfigurationsaufwand auf Ihrem Guix System einen Yubico-Sicherheitsschlüssel einsetzen können. Wir hoffen, die Konfiguration klappt auch bei anderen Sicherheitsschlüsseln, mit geringen Anpassungen.

3.4.1 Einrichten einer Zwei-Faktor-Authentisierung (2FA)

Dazu müssen Sie die udev-Regeln des Systems um auf Ihren Sicherheitsschlüssel abgestimmte Regeln erweitern. Im Folgenden wird gezeigt, wie Sie Ihre udev-Regeln um die Regeldatei `lib/udev/rules.d/70-u2f.rules` aus dem `libfido2`-Paket im Modul `(gnu packages security-token)` erweitern und Ihren Benutzer zur zugehörigen Gruppe ‘`"plugdev"`’ hinzufügen:

```
(use-package-modules ... security-token ...)
...
(operating-system
 ...
 (users (cons* (user-account
```

```
                (name "ihr-benutzer")
                (group "users")
                (supplementary-groups
'("wheel" "netdev" "audio" "video"
                   "plugdev"))           ;<- Systemgruppe hinzufügen
                (home-directory "/home/ihr-benutzer"))
               %base-user-accounts))
 ...
 (services
  (cons*
   ...
   (udev-rules-service 'fido2 libfido2 #:groups '("plugdev")))))
```

Rekonfigurieren Sie Ihr System. Danach melden Sie sich neu an zu einer grafischen Sitzung, damit sich die neue Gruppe Ihres Benutzers auswirken kann. Sie können jetzt prüfen, ob Ihr Sicherheitsschlüssel einsatzbereit ist, indem Sie dies starten:

```
guix shell ungoogled-chromium -- chromium chrome://settings/securityKeys
```

und nachsehen, ob Sie den Sicherheitsschlüssel mit dem Menüeintrag „Sicherheitsschlüssel zurücksetzen“ neu machen können. Wenn es klappt, gratulieren wir zu Ihrem nutzungsbereiten Sicherheitsschlüssel, mit dem Sie nun ihn unterstützende Anwendungen für Zwei-Faktor-Authentisierung (2FA) einrichten können.

3.4.2 Abschalten der OTP-Code-Erzeugung beim Yubikey

Wenn Sie einen Yubikey-Sicherheitsschlüssel verwenden und es Sie stört, dass er unsinnige OTP-Codes erzeugt, wenn Sie ihn unabsichtlich anrühren (und Sie damit am Ende den ‘`#guix`’-Kanal zumüllen, während Sie mit Ihrem Lieblings-IRC-Client an der Diskussion teilnehmen!), dann deaktivieren Sie es einfach mit dem folgenden `ykman`-Befehl:

```
guix shell python-yubikey-manager -- ykman config usb --force --disable OTP
```

Alternativ können Sie auch die Oberfläche mit dem `ykman-gui`-Befehl aus dem Paket `yubikey-manager-qt` starten und entweder gleich die ganze OTP-Anwendung für die USB-Schnittstelle abschalten oder, unter ‘`Applications -> OTP`’, die Konfiguration für Slot 1 löschen, die bei der Yubico-OTP-Anwendung voreingestellt ist.

3.4.3 Yubikey verlangen, um eine KeePassXC-Datenbank zu entsperren

Die Anwendung KeePassXC zur Verwaltung von Passwörtern kann mit Yubikeys zusammenarbeiten, aber erst, wenn die udev-Regeln dafür auf Ihrem Guix System vorliegen und nachdem Sie passende Einstellungen in der Yubico-OTP-Anwendung zum Schlüssel vorgenommen haben.

Die erforderliche udev-Regeldatei liegt im Paket `yubikey-personalization` und kann hiermit installiert werden:

```
(use-package-modules ... security-token ...)
...
(operating-system
 ...
 (services
```

```
(cons*
 ...
 (udev-rules-service 'yubikey yubikey-personalization))))
```

Sobald Sie Ihr System rekonfiguriert haben (und Ihren Yubikey neu verbunden haben), sollten Sie anschließend die OTP-Challenge-Response-Anwendung für Ihren Yubikey auf dessen Slot 2 einrichten, die von KeePassXC benutzt wird. Das ist problemlos möglich mit dem grafischen Konfigurationsprogramm Yubikey Manager, das Sie so aufrufen können:

```
guix shell yubikey-manager-qt -- ykman-gui
```

Stellen Sie zunächst sicher, dass '`OTP`' im Karteireiter '`Interfaces`' aktiviert ist. Begeben Sie sich dann zu '`Applications -> OTP`' und klicken Sie auf den Knopf '`Configure`' im Abschnitt '`Long Touch (Slot 2)`'. Wählen Sie '`Challenge-response`' und geben Sie einen geheimen Schlüssel entweder ein oder lassen Sie ihn erzeugen. Anschließend klicken Sie auf den Knopf '`Finish`'. Wenn Sie noch einen zweiten Yubikey haben, den Sie als Ersatz einsetzen können möchten, konfigurieren Sie ihn auf die gleiche Weise mit *demselben* geheimen Schlüssel.

Jetzt sollte Ihr Yubikey von KeePassXC erkannt werden. Er kann für eine Datenbank hinzugefügt werden, indem Sie in KeePassXC im Menü '`Datenbank -> Datenbank-Sicherheit...`' öffnen und dort auf den Knopf '`Zusätzlichen Schutz hinzufügen ...`' klicken, dann auf '`Challenge-Response hinzufügen`' klicken und den Sicherheitsschlüssel aus der Auswahlliste wählen und den '`OK`'-Knopf anklicken, um die Einrichtung abzuschließen.

3.5 Dynamisches DNS als mcron-Auftrag

Wenn Sie von Ihrem Internetdienstanbieter nur dynamische IP-Adressen bekommen, aber einen statischen Rechnernamen möchten, können Sie einen Dienst für dynamisches DNS (Domain Name System) einrichten (auch bekannt unter der Bezeichnung DDNS (Dynamic DNS)). Dieser ordnet einem festen, „statischen“ Rechnernamen eine öffentliche, aber dynamische (oft wechselnde) IP-Adresse zu. Es gibt davon mehrere Anbieter; im folgenden mcron-Auftrag verwenden wir DuckDNS (`https://duckdns.org`). Er sollte auch mit anderen Anbietern von dynamischem DNS funktionieren, die eine ähnliche Schnittstelle zum Wechsel der IP-Adresse haben, etwa `https://freedns.afraid.org/`, wenn Sie Kleinigkeiten anpassen.

So sieht der mcron-Auftrag dafür aus. Statt *DOMAIN* schreiben Sie Ihre eigene Domain und den von DuckDNS der *DOMAIN* zugeordneten Token in die Datei `/etc/duckdns/`*DOMAIN*`.token`.

```
(define duckdns-job
  ;; IP der eigenen Domain alle 5 Minuten aktualisieren.
  #~(job '(next-minute (range 0 60 5))
 #$(program-file
            "duckdns-update"
            (with-extensions (list guile-gnutls) ;nötig für (web client)
              #~(begin
                  (use-modules (ice-9 textual-ports)
                               (web client))
                  (let ((token (string-trim-both
```

```
                                    (call-with-input-file "/etc/duckdns/DOMAIN.token"
                                      get-string-all)))
                                (query-template (string-append "https://www.duckdns.org/"
                                                               "update?domains=DOMAIN"
                                                               "&token=~a&ip=")))
                           (http-get (format #f query-template token))))))
                  "duckdns-update"
                  #:user "nobody"))
```

Der Auftrag muss in die Liste der mcron-Aufträge für Ihr System eingetragen werden, etwa so:

```
(operating-system
 (services
  (cons* (service mcron-service-type
           (mcron-configuration
             (jobs (list duckdns-job ...))))
         ...
         %base-services)))
```

3.6 Verbinden mit Wireguard VPN

Damit Sie sich mit einem Wireguard-VPN-Server verbinden können, müssen Sie dafür sorgen, dass die dafür nötigen Kernel-Module in den Speicher eingeladen werden und ein Paket bereitsteht, dass die unterstützenden Netzwerkwerkzeuge dazu enthält (z.B. `wireguard-tools` oder `network-manager`).

Hier ist ein Beispiel für eine Konfiguration für Linux-Libre < 5.6, wo sich das Modul noch nicht im Kernel-Quellbaum befindet („out of tree“) und daher von Hand geladen werden muss – in nachfolgenden Kernelversionen ist das Modul bereits eingebaut und *keine* derartige Konfiguration ist nötig.

```
(use-modules (gnu))
(use-service-modules desktop)
(use-package-modules vpn)

(operating-system
  ;; ...
  (services (cons (simple-service 'wireguard-module
                                  kernel-module-loader-service-type
                                  '("wireguard"))
                  %desktop-services))
  (packages (cons wireguard-tools %base-packages))
  (kernel-loadable-modules (list wireguard-linux-compat)))
```

Nachdem Sie Ihr System rekonfiguriert haben, können Sie dann entweder die Wireguard-Werkzeuge („Wireguard Tools“) oder NetworkManager benutzen, um sich mit einem VPN-Server zu verbinden.

3.6.1 Die Wireguard Tools benutzen

Um Ihre Wireguard-Konfiguration zu testen, bietet es sich an, `wg-quick` zu benutzen. Übergeben Sie einfach eine Konfigurationsdatei `wg-quick up ./wg0.conf`. Sie können auch die Konfigurationsdatei in `/etc/wireguard` platzieren und dann stattdessen `wg-quick up wg0` ausführen.

> **Anmerkung:** Seien Sie gewarnt, dass sein Autor diesen Befehl als ein schnell und unsauber geschriebenes Bash-Skript bezeichnet hat.

3.6.2 NetworkManager benutzen

Dank der Unterstützung für Wireguard durch den NetworkManager können wir über den Befehl `nmcli` eine Verbindung mit unserem VPN herstellen. Bislang treffen wir in dieser Anleitung die Annahme, dass Sie den Network-Manager-Dienst aus den `%desktop-services` benutzen. Wenn Sie eine abweichende Konfiguration verwenden, müssen Sie unter Umständen Ihre `services`-Liste abändern, damit ein `network-manager-service-type` geladen wird, und Ihr Guix-System rekonfigurieren.

Benutzen Sie den nmcli-Import-Befehl, um Ihre VPN-Konfiguration zu importieren.

```
# nmcli connection import type wireguard file wg0.conf
Verbindung »wg0« (edbee261-aa5a-42db-b032-6c7757c60fde) erfolgreich hinzugefügt.
```

Dadurch wird eine Konfigurationsdatei in `/etc/NetworkManager/wg0.nmconnection` angelegt. Anschließend können Sie sich mit dem Wireguard-Server verbinden:

```
$ nmcli connection up wg0
Verbindung wurde erfolgreich aktiviert (aktiver D-Bus-Pfad: /org/freedesktop/NetworkMa
```

Nach Voreinstellung wird sich NetworkManager automatisch beim Systemstart verbinden. Um dieses Verhalten zu ändern, müssen Sie Ihre Konfiguration bearbeiten:

```
# nmcli connection modify wg0 connection.autoconnect no
```

Für Informationen speziell zu NetworkManager und Wireguard siehe diesen Blogeintrag von thaller (`https://blogs.gnome.org/thaller/2019/03/15/wireguard-in-networkmanager/`).

3.7 Fensterverwalter (Window Manager) anpassen

3.7.1 StumpWM

Sie können StumpWM mit einem Guix-System installieren, indem Sie die Pakete `stumpwm` und optional auch `` `(,stumpwm "lib") `` in eine Systemkonfigurationsdatei, z.B. `/etc/config.scm`, eintragen.

Eine Beispielkonfiguration kann so aussehen:

```
(use-modules (gnu))
(use-package-modules wm)

(operating-system
  ;; ...
  (packages (append (list sbcl stumpwm `(,stumpwm "lib"))
                    %base-packages)))
```

Nach Voreinstellung benutzt StumpWM die Schriftarten von X11, die auf Ihrem System klein oder verpixelt erscheinen mögen. Sie können das Problem beheben, indem Sie das Lisp-Modul `sbcl-ttf-fonts` aus den Beiträgen zu StumpWM („StumpWM Contrib“) als Systempaket installieren:

```
(use-modules (gnu))
(use-package-modules fonts wm)

(operating-system
  ;; ...
  (packages (append (list sbcl stumpwm `(,stumpwm "lib"))
                    sbcl-ttf-fonts font-dejavu %base-packages)))
```

Den folgenden Code fügen Sie in die Konfigurationsdatei von StumpWM `~/.stumpwm.d/init.lisp` ein:

```
(require :ttf-fonts)
(setf xft:*font-dirs* '("/run/current-system/profile/share/fonts/"))
(setf clx-truetype:+font-cache-filename+ (concat (getenv "HOME")
                                                 "/.fonts/font-cache.sexp"))
(xft:cache-fonts)
(set-font (make-instance 'xft:font :family "DejaVu Sans Mono"
                                   :subfamily "Book" :size 11))
```

3.7.2 Sitzungen sperren

Abhängig von Ihrer Arbeitsumgebung ist das Sperren Ihres Bildschirms vielleicht bereits eingebaut. Wenn nicht, müssen Sie es selbst einrichten. Wenn Sie eine Arbeitsumgebung wie GNOME oder KDE benutzen, ist Sperren normalerweise bereits möglich. Wenn Sie einen einfachen Fensterverwalter („Window Manager“) wie StumpWM oder EXWM benutzen, müssen Sie es vielleicht selbst einrichten.

3.7.2.1 Xorg

Sofern Sie Xorg benutzen, können Sie mit dem Programm xss-lock (`https://www.mankier.com/1/xss-lock`) den Bildschirm für Ihre Sitzung sperren. xss-lock wird durch DPMS ausgelöst, was seit Xorg 1.8 automatisch aktiv ist, wenn zur Laufzeit des Kernels ACPI verfügbar ist.

Um xss-lock zu benutzen, können Sie es einfach ausführen und in den Hintergrund versetzen, bevor Sie Ihren Fensterverwalter z.B. aus Ihrer `~/.xsession` heraus starten:

```
xss-lock -- slock &
exec stumpwm
```

In diesem Beispiel benutzt xss-lock das Programm `slock`, um die eigentliche Sperrung des Bildschirms durchzuführen, wenn es den Zeitpunkt dafür gekommen sieht, weil Sie z.B. Ihr Gerät in den Energiesparmodus versetzen.

Damit slock aber überhaupt die Berechtigung dafür erteilt bekommt, Bildschirme grafischer Sitzungen zu sperren, muss es als setuid-root eingestellt sein, wodurch es Benutzer authentifizieren kann, außerdem braucht es Einstellungen im PAM-Dienst. Um das bereitzustellen, tragen Sie den folgenden Dienst in Ihre `config.scm` ein:

```
(service screen-locker-services-type
```

```
(screen-locker-configuration
 (name "slock")
 (program (file-append slock "/bin/slock"))))
```

Wenn Sie Ihren Bildschirm manuell sperren, z.B. indem Sie slock direkt aufrufen, wenn Sie Ihren Bildschirm sperren wollen, ohne in den Energiesparmodus zu wechseln, dann ist es eine gute Idee, das auch xss-lock mitzuteilen, indem Sie `xset s activate` direkt vor slock ausführen.

3.8 Guix auf einem Linode-Server nutzen

Um Guix auf einem durch Linode (`https://www.linode.com`) bereitgestellten, „gehosteten“ Server zu benutzen, richten Sie zunächst einen dort empfohlenen Debian-Server ein. Unsere Empfehlung ist, zum Wechsel auf Guix mit der voreingestellten Distribution anzufangen. Erzeugen Sie Ihre SSH-Schlüssel.

```
ssh-keygen
```

Stellen Sie sicher, dass Ihr SSH-Schlüssel zur leichten Anmeldung auf dem entfernten Server eingerichtet ist. Das können Sie leicht mit Linodes grafischer Oberfläche zum Hinzufügen von SSH-Schlüsseln bewerkstelligen. Gehen Sie dazu in Ihr Profil und klicken Sie auf die Funktion zum Hinzufügen eines SSH-Schlüssels. Kopieren Sie dort hinein die Ausgabe von:

```
cat ~/.ssh/<benutzername>_rsa.pub
```

Fahren Sie den Linode-Knoten herunter.

Im Karteireiter für „Storage“ bei Linode verkleinern Sie das Laufwerk für Debian. Empfohlen werden 30 GB freier Speicher. Klicken Sie anschließend auf „Add a disk“ und tragen Sie Folgendes in das Formular ein:

- Label: "Guix"
- Filesystem: ext4
- Wählen Sie als Größe den übrigen Speicherplatz.

Drücken Sie im Karteireiter „Configurations“ auf „Edit“. Unter „Block Device Assignment“ klicken Sie auf „Add a Device“. Dort müsste `/dev/sdc` stehen; wählen Sie das Laufwerk „Guix“. Speichern Sie die Änderungen.

Fügen Sie eine Konfiguration hinzu („Add a Configuration“) mit folgenden Eigenschaften:

- Label: Guix
- Kernel: GRUB 2 (Das steht ganz unten! Dieser Schritt ist **WICHTIG!**)
- Block device assignment:
- `/dev/sda`: Guix
- `/dev/sdb`: swap
- Root device: `/dev/sda`
- Schalten Sie alle Dateisystem-/Boot-Helfer ab.

Starten Sie den Knoten jetzt wieder mit der Debian-Konfiguration. Sobald er wieder läuft, verbinden Sie sich mittels SSH zu Ihrem Server über `ssh root@<IP-Adresse-Ihres-Servers>`. (Die IP-Adresse Ihres Servers finden Sie in Linodes Übersichtsseite bei „Summa-

ry“.) Nun können Sie mit den Schritten aus dem Abschnitt “Aus Binärdatei installieren” in *Referenzhandbuch zu GNU Guix* weitermachen:

```
sudo apt-get install gpg
wget https://sv.gnu.org/people/viewgpg.php?user_id=15145 -qO - | gpg --import -
wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
chmod +x guix-install.sh
./guix-install.sh
guix pull
```

Nun wird es Zeit, eine Konfiguration für den Server anzulegen. Die wichtigsten Informationen finden Sie hierunter. Speichern Sie die Konfiguration als `guix-config.scm`.

```
(use-modules (gnu)
             (guix modules))
(use-service-modules networking
                     ssh)
(use-package-modules admin
                     package-management
                     ssh
                     tls)

(operating-system
  (host-name "my-server")
  (timezone "America/New_York")
  (locale "en_US.UTF-8")
  ;; Dieser komisch aussehende Code wird eine grub.cfg
  ;; anlegen ohne den GRUB-Bootloader auf die
  ;; Platte zu installieren.
  (bootloader (bootloader-configuration
               (bootloader
                (bootloader
                 (inherit grub-bootloader)
                 (installer #~(const #true))))))
  (file-systems (cons (file-system
                        (device "/dev/sda")
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))


  (swap-devices (list "/dev/sdb"))


  (initrd-modules (cons "virtio_scsi"    ; Um die Platte zu finden
                        %base-initrd-modules))

  (users (cons (user-account
                (name "janedoe")
```

```
                (group "users")
                ;; Durch Hinzufügen zur "wheel"-Gruppe
                ;; wird das Konto zum Sudoer.
                (supplementary-groups '("wheel"))
                (home-directory "/home/janedoe"))
               %base-user-accounts))

  (packages (cons* openssh-sans-x
                   %base-packages))

  (services (cons*
             (service dhcp-client-service-type)
             (service openssh-service-type
                      (openssh-configuration
                       (openssh openssh-sans-x)
                       (password-authentication? #false)
                       (authorized-keys
                        `(("janedoe" ,(local-file "janedoe_rsa.pub"))
                          ("root" ,(local-file "janedoe_rsa.pub"))))))
             %base-services)))
```

Ersetzen Sie in der obigen Konfiguration aber folgende Felder:

```
(host-name "my-server")       ; hier sollte Ihr Servername stehen
; Wenn Sie sich einen Linode-Server außerhalb der USA ausgesucht
; haben, finden Sie mit tzselect die richtige Zeitzonenangabe.
(timezone "America/New_York") ; Zeitzone ersetzen wenn nötig
(name "janedoe")              ; Ersetzen durch Ihren Benutzernamen
("janedoe" ,(local-file "janedoe_rsa.pub")) ; Ersetzen durch Ihren SSH-Schlüssel
("root" ,(local-file "janedoe_rsa.pub")) ; Ersetzen durch Ihren SSH-Schlüssel
```

Durch die letzte Zeile im obigen Beispiel können Sie sich als Administratornutzer root auf dem Server anmelden und das anfängliche Passwort für root festlegen (lesen Sie den Hinweis zur Anmeldung als root am Ende dieses Rezepts). Nachdem das erledigt ist, können Sie die Zeile aus Ihrer Konfiguration löschen und Ihr System rekonfigurieren, damit eine Anmeldung als root nicht mehr möglich ist.

Kopieren Sie Ihren öffentlichen SSH-Schlüssel (z.B. `~/.ssh/id_rsa.pub`) in die Datei `<`*`Ihr-Benutzername`*`>_rsa.pub` und Ihre `guix-config.scm` ins selbe Verzeichnis. Führen Sie dann diese Befehle in einem neuen Terminal aus:

```
sftp root@<IP-Adresse-des-entfernten-Servers>
put /pfad/mit/dateien/<Benutzername>_rsa.pub .
put /pfad/mit/dateien/guix-config.scm .
```

Binden Sie mit Ihrem ersten Terminal das Guix-Laufwerk ein:

```
mkdir /mnt/guix
mount /dev/sdc /mnt/guix
```

Aufgrund der Art und Weise, wie wir den Bootloader-Abschnitt von `guix-config.scm` oben festgelegt haben, installieren wir GRUB nicht vollständig. Stattdessen installieren wir

nur unsere GRUB-Konfigurationsdatei. Daher müssen wir ein bisschen von den anderen GRUB-Einstellungen vom Debian-System kopieren:

```
mkdir -p /mnt/guix/boot/grub
cp -r /boot/grub/* /mnt/guix/boot/grub/
```

Initialisieren Sie nun die Guix-Installation:

```
guix system init guix-config.scm /mnt/guix
```

OK, fahren Sie Ihn jetzt herunter! Von der Linode-Konsole wählen Sie Booten aus und wählen „Guix“.

Sobald es gestartet ist, sollten Sie sich über SSH anmelden können! (Allerdings wird sich die Serverkonfiguration geändert haben.) Ihnen könnte eine Fehlermeldung wie diese hier gezeigt werden:

```
$ ssh root@<server ip address>
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:0B+wp33w57AnKQuHCvQP0+ZdKaqYrI/kyU7CfVbS7R4.
Please contact your system administrator.
Add correct host key in /home/joshua/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /home/joshua/.ssh/known_hosts:3
ECDSA host key for 198.58.98.76 has changed and you have requested strict checking.
Host key verification failed.
```

Löschen Sie entweder die ganze Datei `~/.ssh/known_hosts` oder nur die ungültig gewordene Zeile, die mit der IP-Adresse Ihres Servers beginnt.

Denken Sie daran, Ihr Passwort und das Passwort des Administratornutzers root festzulegen.

```
ssh root@<IP-Adresse-des-entfernten-Servers>
passwd  ; für das root-Passswort
passwd <benutzername> ; für das Passwort des normalen Benutzers
```

Es kann sein, dass die obigen Befehle noch nicht funktionieren. Wenn Sie Probleme haben, sich aus der Ferne über SSH bei Ihrer Linode-Kiste anzumelden, dann müssen Sie vielleicht Ihr anfängliches Passwort für root und das normale Benutzerkonto festlegen, indem Sie auf die „Launch Console“-Option in Ihrer Linode klicken. Wählen Sie „Glish“ statt „Weblish“. Jetzt sollten Sie über SSH in Ihre Maschine 'reinkommen.

Hurra! Nun können Sie den Server herunterfahren, die Debian-Platte löschen und Guix auf den gesamten verfügbaren Speicher erweitern. Herzlichen Glückwunsch!

Übrigens, wenn Sie das Ergebnis jetzt als „Disk Image“ speichern, können Sie neue Guix-Abbilder von da an leicht einrichten! Vielleicht müssen Sie die Größe des Guix-Abbilds auf 6144MB verkleinern, um es als Abbild speichern zu können. Danach können Sie es wieder auf die Maximalgröße vergrößern.

3.9 Guix auf einem Kimsufi-Server nutzen

Um Guix auf einem von Kimsufi (`https://www.kimsufi.com/`) gemieteten Server einzusetzen, klicken Sie auf den Karteireiter zu Netboot und wählen Sie dann rescue64-pro aus und starten Sie neu.

OVH wird Ihnen eine E-Mail mit den Anmeldedaten schicken, damit Sie sich über SSH auf ein Debian-System verbinden können.

Folgen Sie nun den Anweisungen zur Installation von Guix aus einer Binärdatei (siehe Abschnitt “Aus Binärdatei installieren” in *Referenzhandbuch zu GNU Guix*):

```
wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
chmod +x guix-install.sh
./guix-install.sh
guix pull
```

Partitionieren Sie nun die Laufwerke und formatieren Sie sie, aber erst nachdem Sie das RAID-Array angehalten haben:

```
mdadm --stop /dev/md127
mdadm --zero-superblock /dev/sda2 /dev/sdb2
```

Löschen Sie die Laufwerksinhalte und richten Sie die Partitionen ein; wir erzeugen ein RAID-1-Array.

```
wipefs -a /dev/sda
wipefs -a /dev/sdb

parted /dev/sda --align=opt -s -m -- mklabel gpt
parted /dev/sda --align=opt -s -m -- \
 mkpart bios_grub 1049kb 512MiB \
 set 1 bios_grub on
parted /dev/sda --align=opt -s -m -- \
 mkpart primary 512MiB -512MiB
 set 2 raid on
parted /dev/sda --align=opt -s -m -- mkpart primary linux-swap 512MiB 100%

parted /dev/sdb --align=opt -s -m -- mklabel gpt
parted /dev/sdb --align=opt -s -m -- \
     mkpart bios_grub 1049kb 512MiB \
     set 1 bios_grub on
parted /dev/sdb --align=opt -s -m -- \
     mkpart primary 512MiB -512MiB \
     set 2 raid on
parted /dev/sdb --align=opt -s -m -- mkpart primary linux-swap 512MiB 100%
```

Legen Sie das Array an:

```
mdadm --create /dev/md127 --level=1 --raid-disks=2 \
  --metadata=0.90 /dev/sda2 /dev/sdb2
```

Legen Sie nun Dateisysteme auf diesen Partitionen an, angefangen mit der Boot-Partition:

```
mkfs.ext4  /dev/sda1
```

```
mkfs.ext4  /dev/sdb1
```

Dann die Wurzelpartition:

```
mkfs.ext4 /dev/md127
```

Initialisieren Sie die Swap-Partitionen:

```
mkswap /dev/sda3
swapon /dev/sda3
mkswap /dev/sdb3
swapon /dev/sdb3
```

Binden Sie das Guix-Laufwerk ein:

```
mkdir /mnt/guix
mount /dev/md127 /mnt/guix
```

Der nächste Schritt ist, in eine Datei `os.scm` eine Betriebssystemdeklaration zu schreiben, zum Beispiel:

```
(use-modules (gnu) (guix))
(use-service-modules networking ssh vpn virtualization sysctl admin mcron)
(use-package-modules ssh tls tmux vpn virtualization)

(operating-system
  (host-name "kimsufi")

  (bootloader (bootloader-configuration
       (bootloader grub-bootloader)
       (targets (list "/dev/sda" "/dev/sdb"))
       (terminal-outputs '(console))))

  ;; Wir brauchen ein Kernel-Modul für RAID-1 (d.h. "spiegeln").
  (initrd-modules (cons* "raid1"  %base-initrd-modules))

  (mapped-devices
   (list (mapped-device
          (source (list "/dev/sda2" "/dev/sdb2"))
          (target "/dev/md127")
          (type raid-device-mapping))))

  (swap-devices
   (list (swap-space
          (target "/dev/sda3"))
         (swap-space
          (target "/dev/sdb3"))))

  (issue
   ;; Voreingestellte Botschaft für /etc/issue.
   "\
Hier ist das GNU-System bei Kimsufi.  Willkommen.\n")
```

```
  (file-systems (cons* (file-system
         (mount-point "/")
         (device "/dev/md127")
         (type "ext4")
         (dependencies mapped-devices))
       %base-file-systems))

  (users (cons (user-account
        (name "guix")
        (comment "guix")
        (group "users")
        (supplementary-groups '("wheel"))
        (home-directory "/home/guix"))
       %base-user-accounts))

  (sudoers-file
   (plain-file "sudoers" "\
root ALL=(ALL) ALL
%wheel ALL=(ALL) ALL
guix ALL=(ALL) NOPASSWD:ALL\n"))

  ;; Global installierte Pakete.
  (packages (cons* tmux gnutls wireguard-tools %base-packages))
  (services
   (cons*
    (service static-networking-service-type
     (list (static-networking
    (addresses (list (network-address
      (device "enp3s0")
      (value "Server-IP-Adresse/24"))))
    (routes (list (network-route
   (destination "default")
   (gateway "Server-Netzwerkzugang"))))
    (name-servers '("213.186.33.99")))))

    (service unattended-upgrade-service-type)

    (service openssh-service-type
     (openssh-configuration
      (openssh openssh-sans-x)
      (permit-root-login #f)
      (authorized-keys
       `(("guix" ,(plain-file "SSH-Name-des-Schlüssels.pub"
                              "SSH-Inhalt-öffentl-Schlüssel"))))))
    (modify-services %base-services
      (sysctl-service-type
       config =>
```

```
       (sysctl-configuration
  (settings (append '(("net.ipv6.conf.all.autoconf" . "0")
     ("net.ipv6.conf.all.accept_ra" . "0"))
    %default-sysctl-settings))))))))
```

Vergessen Sie nicht, anstelle der Variablen *Server-IP-Adresse*, *Server-Netzwerkzugang*, *SSH-Name-des-Schlüssels* und *SSH-Inhalt-öffentl-Schlüssel* die für Sie zutreffenden Werte zu schreiben.

Der Netzwerkzugang (Gateway) ist die letzte nutzbare IP in Ihrem Block. Wenn Sie z.B. einen Server mit IP '`37.187.79.10`' haben, dann ist sein Gateway '`37.187.79.254`'.

Übertragen Sie Ihre Datei mit Betriebssystemdeklaration `os.scm` auf den Server mittels des Befehls `scp` oder `sftp`.

Bleibt nur noch, Guix mit `guix system init` zu installieren und neu zu starten.

Aber das klappt nur, wenn wir vorher ein Chroot aufsetzen, weil die Wurzelpartition des laufenden Rettungssystems („rescue“) auf einer aufs-Partition eingebunden ist und, wenn Sie es versuchen, die Installation von Guix beim GRUB-Schritt fehlschlägt mit der Meldung, der kanonische Pfad von „aufs“ sei nicht ermittelbar.

Installieren Sie die Pakete, die im Chroot benutzt werden:

```
guix install bash-static parted util-linux-with-udev coreutils guix
```

Dann legt folgender Befehl die Verzeichnisse für das Chroot an:

```
cd /mnt && \
mkdir -p bin etc gnu/store root/.guix-profile/ root/.config/guix/current \
  var/guix proc sys dev
```

Kopieren Sie die resolv.conf des Wirtssystems in die Chroot:

```
cp /etc/resolv.conf etc/
```

Binden Sie die blockorientierten Speichermedien, den Store, seine Datenbank sowie die aktuelle Guix-Konfiguration ein:

```
mount --rbind /proc /mnt/proc
mount --rbind /sys /mnt/sys
mount --rbind /dev /mnt/dev
mount --rbind /var/guix/ var/guix/
mount --rbind /gnu/store gnu/store/
mount --rbind /root/.config/ root/.config/
mount --rbind /root/.guix-profile/bin/ bin
mount --rbind /root/.guix-profile root/.guix-profile/
```

Betreten Sie ein Chroot in /mnt und installieren Sie das System:

```
chroot /mnt/ /bin/bash

guix system init /root/os.scm /guix
```

Schlussendlich können Sie in der Web-Oberfläche von '`netboot`' auf '`boot to disk`' wechseln und neu starten (auch das tun Sie über die Web-Oberfläche).

Nach ein paar Minuten Wartezeit können Sie versuchen, Ihren Server mit SSH zu betreten: `ssh guix@`*`Server-IP-Adresse`* `-i` *`Pfad-zu-Ihrem-SSH-Schlüssel`*

Sie sollten nun ein nutzbares Guix-System auf Kimsufi am Laufen haben; wir gratulieren!

3.10 Bind-Mounts anlegen

Um ein Dateisystem per „bind mount“ einzubinden, braucht man zunächst ein paar Definitionen. Fügen Sie diese noch vor dem `operating-system`-Abschnitt Ihrer Systemdefinition ein. In diesem Beispiel binden wir ein Verzeichnis auf einem Magnetfestplattenlaufwerk an `/tmp`, um die primäre SSD weniger abzunutzen, ohne dass wir extra eine ganze Partition für `/tmp` erzeugen müssen.

Als Erstes sollten wir das Quelllaufwerk definieren, wo wir das Verzeichnis für den Bind-Mount unterbringen. Dann kann der Bind-Mount es als Abhängigkeit benutzen.

```
(define source-drive ;; "source-drive" can be named anything you want.
   (file-system
    (device (uuid "UUID goes here"))
    (mount-point "/path-to-spinning-disk-goes-here")
    (type "ext4"))) ;Make sure to set this to the appropriate type for your drive.
```

Auch das Quellverzeichnis muss so definiert werden, dass Guix es nicht für ein reguläres blockorientiertes Gerät hält, sondern es als Verzeichnis erkennt.

```
;; "source-directory" can be named any valid variable name.
(define (%source-directory) "/path-to-spinning-disk-goes-here/tmp")
```

In der Definition des `file-systems`-Felds müssen wir die Einbindung einfügen.

```
(file-systems (cons*

               ...<hier würden andere Laufwerke stehen>...

               ;; Must match the name you gave the source drive in the earlier defini
               source-drive

               (file-system
                ;; Make sure "source-directory" matches your earlier definition.
                (device (%source-directory))
                (mount-point "/tmp")
                ;; We are mounting a folder, not a partition, so this type needs to b
                (type "none")
                (flags '(bind-mount))
                ;; Ensure "source-drive" matches what you've named the variable for t
                (dependencies (list source-drive))
                )

               ...<hier würden andere Laufwerke stehen>...

               ))
```

3.11 Substitute über Tor beziehen

Der Guix-Daemon kann einen HTTP-Proxy benutzen, wenn er Substitute herunterlädt. Wir wollen ihn hier so konfigurieren, dass Substitute über Tor bezogen werden.

Warnung: *Nicht aller* Datenverkehr des Guix-Daemons wird dadurch über Tor laufen! Nur HTTP und HTTPS durchläuft den Proxy, *nicht* so ist es bei FTP, dem Git-Protokokl, SSH etc., diese laufen weiterhin durch's „Clearnet“. Die Konfiguration ist also *nicht* narrensicher; ein Teil Ihres Datenverkehrs wird gar nicht über Tor geleitet. Verwenden Sie sie nur auf Ihr eigenes Risiko!

Beachten Sie außerdem, dass sich die hier beschriebene Prozedur nur auf Paketsubstitute bezieht. Wenn Sie Ihre Guix-Distribution mit `guix pull` aktualisieren, müssen Sie noch immer `torsocks` benutzen, wenn Sie die Verbindung zu Guix' Git-Repository über Tor leiten wollen.

Der Substitutserver von Guix ist als Onion-Dienst verfügbar, wenn Sie ihn benutzen möchten, um Ihre Substitute über Tor zu laden, dann konfigurieren Sie Ihr System wie folgt:

```
(use-modules (gnu))
(use-service-module base networking)

(operating-system
  ...
  (services
    (cons
      (service tor-service-type
               (tor-configuration
                 (config-file (plain-file "tor-config"
                                          "HTTPTunnelPort 127.0.0.1:9250"))))
      (modify-services %base-services
        (guix-service-type
          config => (guix-configuration
                      (inherit config)
                      ;; ci.guix.gnu.org's Onion service
                      (substitute-urls "\
https://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclcyd.onion")
                      (http-proxy "http://localhost:9250")))))))
```

Dadurch wird ständig ein Tor-Prozess laufen, der einen HTTP-CONNECT-Tunnel für die Nutzung durch den `guix-daemon` bereitstellt. Der Daemon kann andere Protokolle als HTTP(S) benutzen, um entfernte Ressourcen abzurufen, und Anfragen über solche Protokolle werden Tor *nicht* durchlaufen, weil wir hier nur einen HTTP-Tunnel festlegen. Beachten Sie, dass wir für `substitutes-urls` HTTPS statt HTTP benutzen, sonst würde es nicht funktionieren. Dabei handelt es sich um eine Einschränkung von Tors Tunnel; vielleicht möchten Sie stattdessen `privoxy` benutzen, um dieser Einschränkung nicht zu unterliegen.

Wenn Sie Substitute nicht immer, sondern nur manchmal über Tor beziehen wollen, dann überspringen Sie das mit der `guix-configuration`. Führen Sie einfach das hier aus, wenn Sie ein Substitut über den Tor-Tunnel laden möchten:

```
sudo herd set-http-proxy guix-daemon http://localhost:9250
guix build \
  --substitute-urls=https://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclcyd.o
```

3.12 NGINX mit Lua konfigurieren

NGINX lässt sich mit Lua-Skripts erweitern.

Guix stellt einen NGINX-Dienst bereit, mit dem das Lua-Modul und bestimmte Lua-Pakete geladen werden können, so dass Anfragen beantwortet werden, indem Lua-Skripte ausgewertet werden.

Folgendes Beispiel zeigt eine Systemdefinition mit Einstellungen, um das Lua-Skript `index.lua` bei HTTP-Anfragen an den Endpunkt `http://localhost/hello` auszuwerten:

```
local shell = require "resty.shell"

local stdin = ""
local timeout = 1000  -- ms
local max_size = 4096  -- byte

local ok, stdout, stderr, reason, status =
   shell.run([[/run/current-system/profile/bin/ls /tmp]], stdin, timeout, max_size)

ngx.say(stdout)
```

```
(use-modules (gnu))
(use-service-modules #;... web)
(use-package-modules #;... lua)
(operating-system
  ;; ...
  (services
   ;; ...
   (service nginx-service-type
            (nginx-configuration
             (modules
              (list
               (file-append nginx-lua-module "/etc/nginx/modules/ngx_http_lua_module.s
             (lua-package-path (list lua-resty-core
                                     lua-resty-lrucache
                                     lua-resty-signal
                                     lua-tablepool
                                     lua-resty-shell))
             (lua-package-cpath (list lua-resty-signal))
             (server-blocks
              (list (nginx-server-configuration
                     (server-name '("localhost"))
                     (listen '("80"))
                     (root "/etc")
                     (locations (list
                                 (nginx-location-configuration
                                  (uri "/hello")
                                  (body (list #~(format #f "content_by_lua_file ~s;"
                                                        #$(local-file "index.lua")))))
```

3.13 Musik-Server mit Bluetooth-Audio

Bei MPD, dem Music Player Daemon, handelt es sich um eine vielseitige Server-Anwendung, mit der Sie Musik spielen lassen. Client-Programme an verschiedenen Maschinen im Netzwerk – einem Mobiltelefon, Laptop oder einer Desktop-Workstation – können sich damit verbinden und die Wiedergabe der Tondateien steuern, die Ihre eigene Musiksammung enthalten. MPD dekodiert die Tondateien und spielt sie auf einer oder mehreren Ausgaben ab.

MPD ist so voreingestellt, dass auf dem Standardtonausgabegerät abgespielt wird. Im folgenden Beispiel legen wir eine interessantere Architektur fest: einen Musik-Server, der ohne Bildschirm betrieben werden kann. Eine grafische Oberfläche fehlt, ein Pulseaudio-Daemon ist auch nicht da und es gibt keine lokale Tonausgabe. Stattdessen richten wir MPD für zwei Ausgaben ein: zum einen ein Bluetooth-Lautsprecher, zum anderen ein Web-Server, der Audio an Mediaplayer streamen kann.

Bluetooth einzurichten, ist oft eher frustrierend. Sie müssen Ihr Bluetooth-Gerät koppeln und dabei beachten, dass zum Gerät automatisch eine Verbindung aufgebaut sein soll, sobald es an ist. Das können Sie mit dem Bluetooth-Systemdienst umsetzen, der von der Prozedur `bluetooth-service` geliefert wird.

Rekonfigurieren Sie Ihr System mit mindestens den folgenden Diensten und Paketen:

```
(operating-system
  ;; ...
  (packages (cons* bluez bluez-alsa
                   %base-packages))
  (services
   ;; ...
   (dbus-service #:services (list bluez-alsa))
   (bluetooth-service #:auto-enable? #t)))
```

Starten Sie den `bluetooth`-Dienst und benutzen Sie dann `bluetoothctl`, um einen Scan nach Bluetooth-Geräten durchzuführen. Versuchen Sie, Ihren Bluetooth-Lautsprecher unter dem schwedischen Büfett aus anderen Smart-Home-Gerätschaften Ihrer Nachbarn auszumachen, und merken Sie sich dessen Device-ID in der angezeigten Liste. Das einmal zu machen, genügt:

```
$ bluetoothctl
[NEW] Controller 00:11:22:33:95:7F BlueZ 5.40 [default]

[bluetooth]# power on
[bluetooth]# Changing power on succeeded

[bluetooth]# agent on
[bluetooth]# Agent registered

[bluetooth]# default-agent
[bluetooth]# Default agent request successful

[bluetooth]# scan on
[bluetooth]# Discovery started
```

```
[CHG] Controller 00:11:22:33:95:7F Discovering: yes
[NEW] Device AA:BB:CC:A4:AA:CD My Bluetooth Speaker
[NEW] Device 44:44:FF:2A:20:DC My Neighbor's TV
...

[bluetooth]# pair AA:BB:CC:A4:AA:CD
Attempting to pair with AA:BB:CC:A4:AA:CD
[CHG] Device AA:BB:CC:A4:AA:CD Connected: yes

[My Bluetooth Speaker]# [CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110b-0000-1000-8000-
[CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110c-0000-1000-8000-00xxxxxxxxxx
[CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110e-0000-1000-8000-00xxxxxxxxxx
[CHG] Device AA:BB:CC:A4:AA:CD Paired: yes
Pairing successful

[CHG] Device AA:BB:CC:A4:AA:CD Connected: no

[bluetooth]#
[bluetooth]# trust AA:BB:CC:A4:AA:CD
[bluetooth]# [CHG] Device AA:BB:CC:A4:AA:CD Trusted: yes
Changing AA:BB:CC:A4:AA:CD trust succeeded

[bluetooth]#
[bluetooth]# connect AA:BB:CC:A4:AA:CD
Attempting to connect to AA:BB:CC:A4:AA:CD
[bluetooth]# [CHG] Device AA:BB:CC:A4:AA:CD RSSI: -63
[CHG] Device AA:BB:CC:A4:AA:CD Connected: yes
Connection successful

[My Bluetooth Speaker]# scan off
[CHG] Device AA:BB:CC:A4:AA:CD RSSI is nil
Discovery stopped
[CHG] Controller 00:11:22:33:95:7F Discovering: no
```

Wir gratulieren: Sie bekommen jetzt eine automatische Verbindung zu Ihrem Bluetooth-Speaker!

Es wird Zeit, ALSA einzurichten. Dabei müssen Sie es anweisen, das Bluetooth-Modul *bluealsa* zu verwenden, so dass Sie ein ALSA-pcm-Gerät definieren können, das Ihrem Bluetooth-Lautsprecher entspricht. Wenn Ihr Server keine Bedienelemente haben soll, ist es einfacher, wenn Sie *bluealsa* mit einem festen Bluetooth-Gerät verbinden lassen, statt mit Pulseaudio ein Umschalten der Streams einzurichten. ALSA einzurichten, geht so: Wir schreiben eine passende `alsa-configuration` für den `alsa-service-type`. In der Konfiguration deklarieren Sie den `pcm`-Typ `bluealsa` aus dem Modul `bluealsa` des Pakets `bluez-alsa`. Für diesen Typ definieren Sie dann ein `pcm`-Gerät für Ihren Bluetooth-Lautsprecher.

Was bleibt, ist, MPD die Audiodaten an dieses ALSA-Gerät schicken zu lassen. Zusätzlich fügen wir eine zweite MPD-Ausgabe hinzu, womit die aktuell abgespielten Tondateien auch als Stream über einen Web-Server auf Port 8080 dargeboten

werden. Es ermöglicht, dass man mit anderen Geräten im Netzwerk einen tauglichen Mediaplayer die Daten vom HTTP-Server, auf Port 8080, abspielen lässt, egal ob der Bluetooth-Lautsprecher läuft.

Es folgt ein Umriss, wie eine Betriebssystemdeklaration aussehen kann, die die oben genannten Aufgaben erfüllen dürfte:

```
(use-modules (gnu))
(use-service-modules audio dbus sound #;... etc)
(use-package-modules audio linux #;... etc)
(operating-system
  ;; ...
  (packages (cons* bluez bluez-alsa
                   %base-packages))
  (services
   ;; ...
   (service mpd-service-type
            (mpd-configuration
             (user "your-username")
             (music-dir "/path/to/your/music")
             (address "192.168.178.20")
             (outputs (list (mpd-output
                             (type "alsa")
                             (name "MPD")
                             (extra-options
                              ;; Use the same name as in the ALSA
                              ;; configuration below.
                              '((device . "pcm.btspeaker"))))
                            (mpd-output
                             (type "httpd")
                             (name "streaming")
                             (enabled? #false)
                             (always-on? #true)
                             (tags? #true)
                             (mixer-type 'null)
                             (extra-options
                              '((encoder . "vorbis")
                                (port    . "8080")
                                (bind-to-address . "192.168.178.20")
                                (max-clients . "0") ;no limit
                                (quality . "5.0")
                                (format  . "44100:16:1")))))))))
   (dbus-service #:services (list bluez-alsa))
   (bluetooth-service #:auto-enable? #t)
   (service alsa-service-type
            (alsa-configuration
             (pulseaudio? #false) ;we don't need it
             (extra-options
```

```
              #~(string-append "\
# Declare Bluetooth audio device type \"bluealsa\" from bluealsa module
pcm_type.bluealsa {
    lib \""
#$(file-append bluez-alsa "/lib/alsa-lib/libasound_module_pcm_bluealsa.so") "\"
}

# Declare control device type \"bluealsa\" from the same module
ctl_type.bluealsa {
    lib \""
#$(file-append bluez-alsa "/lib/alsa-lib/libasound_module_ctl_bluealsa.so") "\"
}

# Das eigentliche Bluetooth-Audiogerät definieren.
pcm.btspeaker {
    type bluealsa
    device \"AA:BB:CC:A4:AA:CD\" # Unique Device Identifier
    profile \"a2dp\"
}

# Einen zugehörigen Controller definieren.
ctl.btspeaker {
    type bluealsa
}
"))))))
```

Genießen Sie nun, wie die Musik aus dem MPD-Client Ihrer Wahl erschallt, oder einem Mediaplayer, der über HTTP streamen kann!

4 Container

Durch den Linux-Kernel werden den Systemprozessen zahlreiche gemeinsame Ressourcen zur Verfügung gestellt, etwa eine gemeinsame Sicht auf das Dateisystem, auf andere Prozesse, Netzwerkgeräte, Benutzer- und Gruppenidentitäten und noch mehr. Seit Linux 3.19 ist es möglich, dass Benutzer einige dieser Ressourcen trennen und ausgewählten Prozessen (und deren Kindprozessen) eine andere Sicht auf das System geben.

Wenn ein Prozess zum Beispiel einen getrennten `mount`-Namensraum bekommt, hat er eine eigene Sicht auf das Dateisystem – in seiner Welt kann er nur Verzeichnisse sehen, die in seinen `mount`-Namensraum eingebunden worden sind. Wenn ein Prozess über seinen eigenen `proc`-Namensraum verfügt, scheint es, als sei er der einzige Prozess, der auf dem System läuft, und seine Prozesskennung ist die PID 1.

Guix bringt diese Kernelfunktionen für völlig isolierte Umgebungen und sogar Guix System in Containern zum Einsatz – Container sind wie leichtgewichtige virtuelle Maschinen, die denselben Kernel wie das Wirtssystem benutzen. Daraus können Sie besonders dann Ihren Nutzen ziehen, wenn Sie Guix auf einer Fremddistribution verwenden, denn so können Sie ausschließen, dass die systemweit zugänglichen fremden Bibliotheken und Konfigurationsdateien störenden Einfluss hätten.

4.1 Guix-Container

Der einfachste Einstieg ist, `guix shell` mit der Befehlszeilenoption `--container` aufzurufen. Siehe Abschnitt “Aufruf von guix shell” in *Referenzhandbuch zu GNU Guix* für eine Referenz der möglichen Optionen.

Folgende Befehle legen einen minimalen Shell-Prozess an, bei dem die meisten Namensräume vom übrigen System getrennt sind. Das aktuelle Arbeitsverzeichnis bleibt für den Prozess sichtbar, aber das restliche Dateisystem ist unzugänglich. Diese äußerste Isolation kann sehr hilfreich sein, wenn Sie jeglichen Einfluss durch Umgebungsvariable, global installierte Bibliotheken oder Konfigurationsdateien ausschließen möchten.

```
guix shell --container
```

Diese Umgebung ist kahl und leer. Sie haben in der brachen Umgebung nicht einmal die GNU coreutils und müssen zu ihrer Erkundung mit den in die Shell eingebauten Werkzeugen vorliebnehmen. Selbst das Verzeichnis `/gnu/store` hat seine für gewöhnlich gigantischen Ausmaße verloren und ist nur mehr ein Schatten seiner selbst.

```
$ echo /gnu/store/*
/gnu/store/...-gcc-10.3.0-lib
/gnu/store/...-glibc-2.33
/gnu/store/...-bash-static-5.1.8
/gnu/store/...-ncurses-6.2.20210619
/gnu/store/...-bash-5.1.8
/gnu/store/...-profile
/gnu/store/...-readline-8.1.1
```

In einer solchen Umgebung gibt es nichts für Sie zu tun außer die Umgebung wieder zu verlassen. Das geht, indem Sie `^D` drücken oder `exit` aufrufen, womit die eingeschränkte Shell-Umgebung ihr Ende findet.

Sie können zusätzliche Verzeichnisse in der Container-Umgebung zugänglich machen. Verwenden Sie dazu `--expose=VERZEICHNIS` für eine Verzeichniseinbindung nur mit Lesezugriff innerhalb des Containers oder verwenden Sie `--share=VERZEICHNIS` für Schreibzugriff. Mit einem zusätzlichen Argument nach dem Verzeichnisnamen können Sie den Namen festlegen, der dem Verzeichnis innerhalb des Containers zugeordnet wird. Folgendes Beispiel zeigt, wie Sie das Verzeichnis `/etc` aus dem Wirtssystem auf `/das/wirtssystem/etc` innerhalb eines Containers abbilden, in dem die GNU coreutils installiert sind.

```
$ guix shell --container --share=/etc=/das/wirtssystem/etc coreutils
$ ls /das/wirtssystem/etc
```

Gleichermaßen können Sie verhindern, dass das aktuelle Arbeitsverzeichnis eine Zuordnung in den Container bekommt, indem Sie die Befehlszeilenoption `--no-cwd` angeben. Es ist eine gute Idee, ein Verzeichnis anzulegen, das innerhalb des Containers das Persönliche Verzeichnis (auch „Home-Verzeichnis“) ist. Aus diesem heraus lassen Sie die Shell für den Container starten.

Auf einem fremden System ist es möglich, in einer Container-Umgebung Software zu kompilieren, die mit den Bibliotheken des Systems oder mit dessen Compiler-Toolchain inkompatibel ist. In der Forschung kommt es häufiger vor, dass man in einer R-Sitzung Pakete installieren möchte. Ohne Container ist es gut möglich, dass die Compiler-Toolchain des Fremdsystems und dessen inkompatible Bibliotheken Vorrang haben und die Binärdateien *nicht* zusammenpassen und in R *nicht* benutzt werden können. In einer Container-Shell gibt es das Problem *nicht*, denn die Bibliotheken und Programme des äußeren Systems sind schlicht gar nicht da, weil der `mount`-Namensraum getrennt ist.

Schauen wir uns ein umfassendes Manifest für eine komfortable R-Entwicklungsumgebung an:

```
(specifications->manifest
  (list "r-minimal"

        ;; grundlegende Pakete
        "bash-minimal"
        "glibc-locales"
        "nss-certs"

        ;; Befehlszeilenwerkzeuge, die man oft braucht, sonst
        ;; wäre der Container so gut wie leer.
        "coreutils"
        "grep"
        "which"
        "wget"
        "sed"

        ;; R-Programme für Markdown
        "pandoc"

        ;; Toolchain und häufige Bibliotheken für "install.packages"
        "gcc-toolchain@10"
        "gfortran-toolchain"
```

```
        "gawk"
        "tar"
        "gzip"
        "unzip"
        "make"
        "cmake"
        "pkg-config"
        "cairo"
        "libxt"
        "openssl"
        "curl"
        "zlib"))
```

Nehmen wir dieses Manifest und richten uns eine Container-Umgebung ein, in der wir R ausführen. Der Einfachheit halber wollen wir den `net`-Namensraum teilen und bekommen Zugriff auf die Netzwerkschnittstellen des Wirtssystems. Damit können wir R-Pakete auf traditionelle Weise aus ihrem Quellcode erstellen, ohne uns um inkompatible ABIs oder sonstige Inkompatibilitäten sorgen zu müssen.

```
$ guix shell --container --network --manifest=manifest.scm -- R

R version 4.2.1 (2022-06-23) -- "Funny-Looking Kid"
Copyright (C) 2022 The R Foundation for Statistical Computing
...
> e <- Sys.getenv("GUIX_ENVIRONMENT")
> Sys.setenv(GIT_SSL_CAINFO=paste0(e, "/etc/ssl/certs/ca-certificates.crt"))
> Sys.setenv(SSL_CERT_FILE=paste0(e, "/etc/ssl/certs/ca-certificates.crt"))
> Sys.setenv(SSL_CERT_DIR=paste0(e, "/etc/ssl/certs"))
> install.packages("Cairo", lib=paste0(getwd()))
...
* installing *source* package 'Cairo' ...
...
* DONE (Cairo)

The downloaded source packages are in
'/tmp/RtmpCuwdwM/downloaded_packages'
> library("Cairo", lib=getwd())
> # success!
```

Containerisierte Shells einzusetzen ist lustig, aber wenn man mehr will als einen einzelnen interaktiven Prozess, werden sie ein bisschen umständlich. Manche Aufgaben sind einfach leichter zu lösen, wenn Sie sie auf einem ordentlichen Guix-System als festem Fundament aufbauen. Dann bekämen Sie Zugriff auf das reichhaltige Angebot von Systemdiensten. Im nächsten Abschnitt wird Ihnen gezeigt, wie Sie eine vollständige Instanz von Guix System in einem Container starten können.

4.2 Container mit Guix System

Guix System stellt eine breite Palette von ineinandergreifenden Systemdiensten zur Verfügung, welche Sie deklarativ konfigurieren, um ein verlässliches und zustandsloses

GNU-System als Fundament für Ihre weiteren Aufgaben zu haben. Selbst wenn Sie Guix auf einer Fremddistribution verwenden, können Sie von diesem Aufbau profitieren, indem Sie so ein System in einem Container starten. Dabei greifen wir auf die Kernel-Funktionalität getrennter Namensräume zurück, die im vorigen Abschnitt genannt wurde. Durch sie wird die erstellte Guix-System-Instanz vom Wirtssystem isoliert und nur die Orte im Dateisystem freigegeben, die Sie ausdrücklich deklariert haben.

Der Unterschied zwischen einem Guix-System-Container und einem Shell-Prozess, wie ihn `guix shell --container` anlegt, besteht in mehreren wichtigen Einzelheiten. Während in einem Shell-Container der containerisierte Prozess ein Bash-Shell-Prozess ist, wird in einem Guix-System-Container Shepherd als PID 1 ausgeführt. In einem Systemcontainer werden all die Systemdienste (siehe Abschnitt "Dienste" in *Referenzhandbuch zu GNU Guix*) auf dieselbe Weise eingerichtet wie wenn Sie Guix System in einer virtuellen Maschine oder auf echter Hardware booten – auch die Daemons, die durch GNU Shepherd gestartet werden (siehe Abschnitt "Shepherd-Dienste" in *Referenzhandbuch zu GNU Guix*) und andere Arten, das System zu erweitern (siehe Abschnitt "Dienstkompositionen" in *Referenzhandbuch zu GNU Guix*), sind die gleichen.

Dass ein Guix-System-Container eine Zunahme an Komplexität bedeute, lässt sich leicht rechtfertigen, wenn Sie mit komplexeren Anwendungen zu tun haben und deren höheren oder engeren Anforderungen an ihren Ausführungskontext – an Konfigurationsdateien, eigene Benutzerkonten, Verzeichnisse für Zwischenspeicher oder Protokolldateien, etc. In Guix System werden solche Anforderungen einer Software durch das Aufspielen von Systemdiensten erfüllt.

4.2.1 Ein Datenbank-Container

Ein geeignetes Beispiel wäre ein PostgreSQL-Datenbankserver. Die Komplexität, ihn aufzusetzen, können Sie großenteils in einer verführerisch kurzen Dienstdeklaration einfangen:

```
(service postgresql-service-type
         (postgresql-configuration
          (postgresql postgresql-14)))
```

Eine vollumfängliche Betriebssystemdeklaration, die Sie für einen Guix-System-Container verwenden könnten, sähe in etwa so aus:

```
(use-modules (gnu))
(use-package-modules databases)
(use-service-modules databases)

(operating-system
  (host-name "container")
  (timezone "Europe/Berlin")
  (file-systems (cons (file-system
                        (device (file-system-label "ist-egal"))
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))
  (bootloader (bootloader-configuration
               (bootloader grub-bootloader)
               (targets '("/dev/sdX")))))
```

```
  (services
   (cons* (service postgresql-service-type
                   (postgresql-configuration
                    (postgresql postgresql-14)
                    (config-file
                     (postgresql-config-file
                      (log-destination "stderr")
                      (hba-file
                       (plain-file "pg_hba.conf"
                                   "\
local all all trust
host all all 10.0.0.1/32  trust"))
                      (extra-config
                       '(("listen_addresses" "*")
                         ("log_directory"    "/var/log/postgresql")))))))
          (service postgresql-role-service-type
                   (postgresql-role-configuration
                    (roles
                     (list (postgresql-role
                            (name "test")
                            (create-database? #t))))))
          %base-services)))
```

Mit `postgresql-role-service-type` definieren wir eine Rolle namens „test“ und lassen eine zugehörige Datenbank erzeugen, so dass wir gleich mit dem Testen anfangen können ohne weitere manuelle Schritte. Mit den Einstellungen in `postgresql-config-file` wird einem Client mit der IP-Adresse 10.0.0.1 die Berechtigung verliehen, eine Verbindung zur Datenbank aufzubauen, ohne sich authentisieren zu müssen – eigentlich keine gute Idee bei Produktivsystemen, aber für dieses Beispiel geeignet.

Erstellen wir uns ein Skript, das eine Instanz dieses Guix-Systems als Container anlegt. Nachdem Sie obige Betriebssystemdeklaration mit dem `operating-system` in eine Datei `os.scm` speichern, entsteht das Startprogramm für den Container durch einen Aufruf von `guix system container` (siehe Abschnitt “Aufruf von guix system” in *Referenzhandbuch zu GNU Guix*).

```
$ guix system container os.scm
Folgende Ableitungen werden erstellt:
  /gnu/store/...-run-container.drv
  ...
/gnu/store/...-run-container.drv wird erstellt ...
/gnu/store/...-run-container
```

Jetzt, wo unser Startprogramm fertig ist, können wir es ausführen und ein neues System mit einem laufenden PostgreSQL-Dienst entschlüpft. Anmerkung: Aufgrund derzeit noch ungelöster Einschränkungen kann das Startprogramm nur mit Administratorrechten ausgeführt werden, zum Beispiel mit `sudo`.

```
$ sudo /gnu/store/...-run-container
system container is running as PID 5983
...
```

Versetzen Sie den Prozess in den Hintergrund, indem Sie `Strg-z` drücken und danach `bg` eingeben. Achten Sie auf die ausgegebene Prozesskennung (PID); wir brauchen sie später noch, um uns mit dem Container zu verbinden. Wissen Sie was? Schauen wir uns doch gleich an, wie Sie eine Verbindung zum Container herstellen. Dazu benutzen wir `nsenter`, ein Werkzeug, das im Paket `util-linux` zu finden ist:

```
$ guix shell util-linux
$ sudo nsenter -a -t 5983
root@container /# pgrep -a postgres
49 /gnu/store/...-postgresql-14.4/bin/postgres -D /var/lib/postgresql/data --config-fi
51 postgres: checkpointer
52 postgres: background writer
53 postgres: walwriter
54 postgres: autovacuum launcher
55 postgres: stats collector
56 postgres: logical replication launcher
root@container /# exit
```

Wie Sie sehen, läuft der PostgreSQL-Dienst im Container!

4.2.2 Netzwerkverbindungen im Container

Was nützt ein Guix System mit laufendem PostgreSQL-Datenbankdienst in einem Container, wenn es nur mit Prozessen reden kann, die ihren Ursprung in dem Container haben? Viel besser wäre es doch, könnten wir die Datenbank über das Netzwerk ansprechen.

Am einfachsten geht das mit einem Paar verbundener virtueller Ethernet-Geräte (bekannt als `veth`). Wir schieben das eine Gerät (`ceth-test`) in den `net`-Namensraum des Containers und lassen das andere Ende (`veth-test`) der Verbindung auf dem Wirtssystem.

```
pid=5983
ns="guix-test"
host="veth-test"
client="ceth-test"

# Anbringen des neuen net-Namensraums "guix-test" an der PID des Containers.
sudo ip netns attach $ns $pid

# Das Geräte-Paar erzeugen.
sudo ip link add $host type veth peer name $client

# Das Client-Gerät in den net-Namensraum des Containers verschieben.
sudo ip link set $client netns $ns
```

Anschließend konfigurieren wir die Wirtsseite:

```
sudo ip link set $host up
sudo ip addr add 10.0.0.1/24 dev $host
```

... und dann konfigurieren wir die Clientseite:

```
sudo ip netns exec $ns  ip link set lo up
sudo ip netns exec $ns  ip link set $client up
sudo ip netns exec $ns  ip addr add 10.0.0.2/24 dev $client
```

Zu diesem Zeitpunkt kann der Wirt den Container unter der IP-Adresse 10.0.0.2 erreichen und der Container kann seinerseits den Wirt auf IP-Adresse 10.0.0.1 erreichen. Das war alles, um mit dem Datenbankdienst, der im Container steckt, vom außen liegenden Wirtssystem aus reden zu können.

```
$ psql -h 10.0.0.2 -U test
psql (14.4)
Type "help" for help.

test=> CREATE TABLE hallo (wen TEXT NOT NULL);
CREATE TABLE
test=> INSERT INTO hallo (wen) VALUES ('Welt');
INSERT 0 1
test=> SELECT * FROM hallo;
  wen
-------
  Welt
(1 row)
```

Jetzt, wo wir mit der kurzen Demonstration fertig sind, geht es an's Aufräumen:

```
sudo kill $pid
sudo ip netns del $ns
sudo ip link del $host
```

5 Virtuelle Maschinen

Mit Guix können Sie Disk-Images erzeugen (siehe Abschnitt “Aufruf von guix system” in *Referenzhandbuch zu GNU Guix*), um sie in Software für virtuelle Maschinen wie virt-manager, GNOME Boxes oder auch dem schlichten QEMU einzusetzen, nur als Beispiel.

In diesem Kapitel wollen wir direkt nutzbare, praktische Beispiele anschauen, wie man virtuelle Maschinen auf Guix System konfigurieren und nutzen will.

5.1 Netzwerkbrücke für QEMU

In der Voreinstellung wird der QEMU-Netzwerkstapel auf dem Wirtssystem als normaler Nutzer ausgeführt („User-Mode Host Network Back-end“), was den Vorteil hat, dass man nichts weiter konfigurieren muss. Unglücklicherweise kann man damit *nicht* alles machen. In diesem Modus bekommt die Gast-VM (virtuelle Maschine) Zugriff auf das Netzwerk auf dieselbe Weise wie das Wirtssystem, aber vom Wirt aus kann man keine Verbindung zum Gast zustande bringen. Außerdem können, weil in QEMU das User-Mode-Netzwerk über ICMP läuft, *keine* ICMP-basierten Netzwerkprogramme wie `ping` funktionieren. Daher ist es oft ratsam, eine Netzwerkbrücke zu konfigurieren, über die das Gastsystem vollständig am Netzwerk teilhat. So kann man auf dem Gast auch einen Server betreiben.

5.1.1 Eine Netzwerkbrücken-Schnittstelle aufbauen

Es gibt viele Möglichkeiten, wie man eine Netzwerkbrücke herstellen kann. Wenn die Betriebssystemdeklaration Ihres Systems auf einer der Vorlagen für Desktop-Rechner basiert, läuft darauf NetworkManager. Mit folgendem Befehl kann man NetworkManager über dessen befehlszeilenbasiertes Programm `nmcli` („Command Line Interface“, CLI) anweisen, eine Netzwerkbrücke aufzubauen:

```
# nmcli con add type bridge con-name br0 ifname br0
```

Um diese Brücke zu Ihrem Netzwerk hinzuzufügen, müssen Sie Ihre Netzwerkbrücke der Ethernet-Schnittstelle zuordnen, über die Sie sich mit dem Netzwerk verbinden. Wenn Ihre Schnittstelle z.B. die Bezeichnung ‘`enp2s0`’ hat, dann wird die Zuordnung mit folgendem Befehl hergestellt:

```
# nmcli con add type bridge-slave ifname enp2s0 master br0
```

Wichtig: Zu einer Brücke können nur Ethernet-Schnittstellen hinzugefügt werden. Wenn Sie ein Drahtlosnetzwerk benutzen, richten Sie wohl besser Routing für Ihr Netzwerk ein, beschrieben in Abschnitt 5.2 [Netzwerk-Routing anschalten in libvirt], Seite 62.

In der Voreinstellung ermöglicht es die Netzwerkbrücke den Gastsystemen, ihre IP-Adressen mittels DHCP zu beziehen, wenn DHCP auf Ihrem lokalen Netzwerk zur Verfügung gestellt wird. Weil das einfach ist, werden wir es hier benutzen. Um Gastmaschinen im Netzwerk zu finden, können Sie die Gäste so konfigurieren, dass diese ihre Rechnernamen über mDNS mitteilen.

5.1.2 Das QEMU-Bridge-Helper-Skript konfigurieren

Zu QEMU gehört ein Hilfsprogramm, mit dem eine Netzwerkbrücken-Schnittstelle zur Verwendung durch „unprivilegierte“ Nutzer ohne besondere Berechtigungen einfach eingerichtet

wird (siehe Abschnitt “Network options” in *QEMU-Dokumentation*. Das Programm muss setuid-root gesetzt sein, damit es funktioniert. Dazu fügen Sie es zum `setuid-programs`-Feld Ihrer Betriebssystemkonfiguration (auf dem Wirtssystem) hinzu, so wie hier:

```
(setuid-programs
 (cons (file-append qemu "/libexec/qemu-bridge-helper")
       %setuid-programs))
```

Zudem muss in der Datei `/etc/qemu/bridge.conf` eine Erlaubnis für die Netzwerkbrücken-Schnittstelle eingetragen werden, denn nach Voreinstellung wird alles blockiert. Fügen Sie dazu folgenden Dienst zu Ihrer `services`-Liste hinzu:

```
(extra-special-file "/etc/qemu/host.conf" "allow br0\n")
```

5.1.3 QEMU mit den richtigen Befehlszeilenoptionen aufrufen

Wenn Sie QEMU aufrufen, müssen Sie die folgenden Optionen angeben, damit die Netzwerkbrücke benutzt wird, nachdem Sie eine einzigartige MAC-Adresse für den Gast vorgeben.

Wichtig: Nach Voreinstellung wird dieselbe MAC-Adresse für alle Gastsysteme benutzt, wenn Sie keine angeben. Wenn sich die MAC-Adressen mehrerer virtueller Maschinen aber nicht unterscheiden, kommt es zu Netzwerkfehlern, wenn Sie die Netzwerkbrücke benutzen.

```
$ qemu-system-x86_64 [...] \
    -device virtio-net-pci,netdev=user0,mac=XX:XX:XX:XX:XX:XX \
    -netdev bridge,id=user0,br=br0 \
    [...]
```

Um MAC-Adressen mit dem für QEMU registrierten Präfix zu erzeugen, können Sie folgendes Code-Schnipsel eintippen:

```
mac_address="52:54:00:$(dd if=/dev/urandom bs=512 count=1 2>/dev/null \
                           | md5sum \
                           | sed -E 's/^(..)(..)(..).*$/\1:\2:\3/')"
echo $mac_address
```

5.1.4 Durch Docker verursachte Netzwerkprobleme

Wenn Sie auf Ihrer Maschine Docker einsetzen, kann es zu Verbindungsproblemen kommen, wenn Sie eine Netzwerkbrücke zu benutzen versuchen. Der Grund ist, dass auch Docker Netzwerkbrücken mitbringt und seine eigenen Netzwerkregeln einrichtet. Die Lösung liegt im Hinzufügen des folgenden `iptables`-Schnipsels in Ihre Betriebssystemdeklaration:

```
(service iptables-service-type
             (iptables-configuration
              (ipv4-rules (plain-file "iptables.rules" "\
*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A FORWARD -i br0 -o br0 -j ACCEPT
COMMIT
"))
```

5.2 Netzwerk-Routing anschalten in libvirt

Wenn die Maschine, auf der Ihre virtuellen Maschinen laufen sollen, nur eine drahtlose Verbindung ins Netzwerk hat, haben Sie keine Möglichkeit, eine echte Netzwerkbrücke zu deren Anbindung einzurichten, wie wir sie im vorigen Abschnitt beschrieben haben (siehe Abschnitt 5.1 [Netzwerkbrücke für QEMU], Seite 60). Ihnen bleibt die zweitbeste Option, eine *virtuelle* Netzwerkbrücke mit statischem Routing zu benutzen und eine mit libvirt betriebene virtuelle Maschine so einzurichten, dass sie benutzt wird (zum Beispiel mit Hilfe der grafischen Benutzeroberfläche `virt-manager`). Das ist ähnlich zu dem voreingestellten Betriebsmodus von QEMU/libvirt, außer dass wir, statt NAT (Network Address Translation) zum Einsatz zu bringen, mit statischen Routen die IP-Adresse der VM (virtuellen Maschine) ins LAN (Local Area Network) aufnehmen. Damit wird eine beidseitige Verbindung zu und von der virtuellen Maschine möglich, damit alle in Ihrem Netzwerk Zugriff auf die auf der virtuellen Maschine angebotenen Dienste bekommen.

5.2.1 Eine virtuelle Netzwerkbrücke anlegen

Eine virtuelle Netzwerkbrücke besteht aus mehreren Komponenten bzw. Konfigurationen wie einer TUN-Schnittstelle („Netzwerk-Tunnel-Schnittstelle“), DHCP-Server (dnsmasq) und Firewall-Regeln (iptables). Mit dem Befehl `virsh` aus dem Paket `libvirt` lässt sich die virtuelle Bridge sehr einfach anlegen. Zuerst müssen Sie wissen, welches Netzwerk-Subnetz Ihre virtuelle Brücke haben soll. Wenn Ihrem LAN zu Hause das ‘`192.168.1.0/24`’-Netzwerk gehört, könnten Sie z.B. ‘`192.168.2.0/24`’ vergeben. Legen Sie eine XML-Datei an z.B. als `/tmp/virbr0.xml` mit folgendem Inhalt:

```
<network>
  <name>virbr0</name>
  <bridge name="virbr0" />
  <forward mode="route"/>
  <ip address="192.168.2.0" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.2.1" end="192.168.2.254"/>
    </dhcp>
  </ip>
</network>
```

Damit legen Sie die Schnittstelle mit Hilfe des `virsh`-Befehls an, welchen Sie als Administratorbenutzer `root` ausführen:

```
virsh net-define /tmp/virbr0.xml
virsh net-autostart virbr0
virsh net-start virbr0
```

Die Schnittstelle ‘`virbr0`’ müsste jetzt zum Beispiel in der Ausgabe des Befehls ‘`ip address`’ zu sehen sein. Sie wird automatisch gestartet, sobald Ihre virtuelle Maschine mit libvirt gestartet wird.

5.2.2 Statische Routen für Ihre virtuelle Netzwerkbrücke einrichten

Wenn Sie Ihre virtuelle Maschine so eingerichtet haben, dass sie auf der neu geschaffenen Schnittstelle ‘`virbr0`’ für die virtuelle Bridge läuft, sollte sie von sich aus bereits eine IP

wie ‘`192.168.2.15`’ über DHCP beziehen und darüber von dem Wirts-Serverrechner aus erreichbar sein mit ‘`ping 192.168.2.15`’ oder entsprechend. Zum Schluss fehlt noch eine letzte Konfiguration, damit die VM auch das externe Netzwerk erreichen kann: Sie müssen im für das Netzwerk zuständigen Router statische Routen hinzufügen.

In diesem Beispiel gehen wir davon aus, dass das LAN-Netzwerk bei Ihnen den Block ‘`192.168.1.0/24`’ bekommt und die Router-Einstellungswebseite könnte z.B. unter `http://192.168.1.1` verfügbar sein. Wenn auf Ihrem Router die Firmware von libreCMC (`https://librecmc.org/`) läuft, würden Sie nach Netzwerk → Statische Routen navigieren (auf die Seite `https://192.168.1.1/cgi-bin/luci/admin/network/routes`) und dort einen neuen Eintrag für ‘`Statische IPv4-Routen`’ mit den folgenden Informationen anlegen:

‘`Schnittstelle`’
: lan

‘`Ziel`’
: 192.168.2.0

‘`IPv4-Netzmaske`’
: 255.255.255.0

‘`IPv4-Gateway`’
: *Server-IP*

‘`Routen-Typ`’
: unicast

Dabei schreiben Sie statt *Server-IP* die IP-Adresse der Wirtsmaschine, auf der die VM laufen. Diese sollte als statische Adresse eingerichtet sein.

Sobald Sie diese neue statische Route speichern und anwenden, sollten Sie sich von und nach außen mit Ihrer VM verbinden können. Um zu überprüfen, ob es funktioniert, führen Sie z.B. den Befehl ‘`ping gnu.org`’ aus.

6 Fortgeschrittene Paketverwaltung

Guix ist ein funktionales Paketverwaltungsprogramm, das weit mehr Funktionalitäten als traditionelle Paketverwalter anbietet. Für nicht Eingeweihte sind deren Anwendungsfälle nicht sofort ersichtlich. Dieses Kapitel ist dazu da, manche fortgeschrittenen Paketverwaltungskonzepte zu demonstrieren.

Siehe Abschnitt “Paketverwaltung” in *Referenzhandbuch zu GNU Guix* für eine vollständige Referenz.

6.1 Guix-Profile in der Praxis

Guix gibt uns eine sehr nützliche Funktionalität, die Neuankömmlingen sehr fremd sein dürfte: *Profile*. Mit ihnen kann man Paketinstallationen zusammenfassen und jeder Benutzer desselben Systems kann so viele davon anlegen, wie sie oder er möchte.

Ob Sie ein Entwickler sind oder nicht, Sie dürften feststellen, dass mehrere Profile ein mächtiges Werkzeug sind, das Sie flexibler macht. Zwar ist es ein gewisser Paradigmenwechsel verglichen mit *traditioneller Paketverwaltung*, doch sind sie sehr praktisch, sobald man im Umgang mit ihnen den Dreh ’raushat.

> **Anmerkung:** Die Anleitung in diesem Abschnitt setzt sich meinungsstark für die Nutzung mehrerer Profile ein. Als sie geschrieben wurde, gab es `guix shell` und dessen schnelles Zwischenspeichern der damit angelegten Profile noch nicht (siehe Abschnitt “Aufruf von guix shell” in *Referenzhandbuch zu GNU Guix*).
>
> Oftmals dürfte es für Sie einfacher sein, mit `guix shell` die von Ihnen benötigten Umgebungen dann aufzusetzen, wenn Sie sie brauchen, statt dass Sie sich der Pflege eines dedizierten Profils annehmen müssen. Es ist Ihre Entscheidung!

Wenn Ihnen Pythons ‘`virtualenv`’ vertraut ist, können Sie sich ein Profil als eine Art universelles ‘`virtualenv`’ vorstellen, das jede Art von Software enthalten kann und nicht nur Python-Software. Des Weiteren sind Profile selbstversorgend: Sie schließen alle Laufzeitabhängigkeiten ein und garantieren somit, dass alle Programme innerhalb eines Profils stets zu jeder Zeit funktionieren werden.

Mehrere Profile bieten viele Vorteile:

- Klare semantische Trennung der verschiedenen Pakete, die ein Nutzer für verschiedene Kontexte braucht.
- Mehrere Profile können in der Umgebung verfügbar gemacht werden, entweder beim Anmelden oder in einer eigenen Shell.
- Profile können bei Bedarf geladen werden. Zum Beispiel kann der Nutzer mehrere Unter-Shells benutzen, von denen jede ein anderes Profil ausführt.
- Isolierung: Programme aus dem einen Profil werden keine Programme aus dem anderen benutzen, und der Nutzer kann sogar verschiedene Versionen desselben Programms in die zwei Profile installieren, ohne dass es zu Konflikten kommt.
- Deduplizierung: Profile teilen sich Abhängigkeiten, wenn sie genau gleich sind. Dadurch sind mehrere Profile speichereffizient.

- Reproduzierbar: Wenn man dafür deklarative Manifeste benutzt, kann ein Profil allein durch den bei dessen Einrichtung aktiven Guix-Commit eindeutig spezifiziert werden. Das bedeutet, dass man genau dasselbe Profil jederzeit und überall einrichten kann (`https://guix.gnu.org/blog/2018/multi-dimensional-transactions-and-rollbacks-oh-my/`) und man dafür nur die Commit-Informationen braucht. Siehe den Abschnitt über Abschnitt 6.1.5 [Reproduzierbare Profile], Seite 69.
- Leichtere Aktualisierung und Wartung: Mit mehreren Profilen ist es ein Leichtes, eine Liste von Paketen zur Hand zu haben und Aktualisierungen völlig reibungslos ablaufen zu lassen.

Konkret wären diese hier typische Profile:

- Die Abhängigkeiten des Projekts, an dem Sie arbeiten.
- Die Bibliotheken Ihrer Lieblingsprogrammiersprache.
- Programme nur für Laptops (wie '`powertop`'), für die Sie auf einem „Desktop"-Rechner keine Verwendung haben.
- TEXlive (das kann wirklich praktisch sein, wenn Sie nur ein einziges Paket für dieses eine Dokument installieren müssen, das Ihnen jemand in einer E-Mail geschickt hat).
- Spiele.

Tauchen wir ein in deren Einrichtung!

6.1.1 Grundlegende Einrichtung über Manifeste

Ein Guix-Profil kann über ein *Manifest* eingerichtet werden. Ein Manifest ist ein in Scheme geschriebenes Codeschnipsel, mit dem die Pakete spezifiziert werden, die Sie in Ihrem Profil haben möchten. Das sieht etwa so aus:

```
(specifications->manifest
  '("paket-1"
    ;; Version 1.3 von paket-2.
    "paket-2@1.3"
    ;; Die "lib"-Ausgabe von paket-3.
    "paket-3:lib"
    ; ...
    "paket-N"))
```

Siehe Abschnitt "Manifeste verfassen" in *Referenzhandbuch zu GNU Guix* für mehr Informationen zur Syntax.

Wir können eine Manifestspezifikation für jedes Profil schreiben und es auf diese Weise installieren:

```
GUIX_EXTRA_PROFILES=$HOME/.guix-extra-profiles
mkdir -p "$GUIX_EXTRA_PROFILES"/my-project # if it does not exist yet
guix package --manifest=/path/to/guix-my-project-manifest.scm \
 --profile="$GUIX_EXTRA_PROFILES"/my-project/my-project
```

Hierbei haben wir eine beliebig benannte Variable '`GUIX_EXTRA_PROFILES`' eingerichtet, die auf das Verzeichnis verweist, wo wir unsere Profile für den Rest dieses Artikels speichern wollen.

Wenn Sie all Ihre Profile in ein einzelnes Verzeichnis legen und jedes Profil ein Unterverzeichnis darin bekommt, ist die Organisation etwas verständlicher. Dadurch wird jedes Unterverzeichnis all die symbolischen Verknüpfungen für genau ein Profil enthalten. Außerdem wird es von jeder Programmiersprache aus einfach, eine „Schleife über die Profile“ zu schreiben (z.B. in einem Shell-Skript), indem Sie es einfach die Unterverzeichnisse von '`$GUIX_EXTRA_PROFILES`' in einer Schleife durchlaufen lassen.

Beachten Sie, dass man auch eine Schleife über die Ausgabe von

```
guix package --list-profiles
```

schreiben kann, obwohl Sie dabei wahrscheinlich `~/.config/guix/current` herausfiltern wollen würden.

Um bei der Anmeldung alle Profile zu aktivieren, fügen Sie dies in Ihre `~/.bash_profile` ein (oder etwas Entsprechendes):

```
for i in $GUIX_EXTRA_PROFILES/*; do
  profile=$i/$(basename "$i")
  if [ -f "$profile"/etc/profile ]; then
    GUIX_PROFILE="$profile"
    . "$GUIX_PROFILE"/etc/profile
  fi
  unset profile
done
```

Eine Anmerkung für Nutzer von „Guix System“: Obiger Code entspricht dem, wie Ihr voreingestelltes Profil `~/.guix-profile` durch `/etc/profile` aktiviert wird, was nach Vorgabe durch `~/.bashrc` geladen wird.

Selbstverständlich können Sie sich auch dafür entscheiden, nur eine Teilmenge zu aktivieren:

```
for i in "$GUIX_EXTRA_PROFILES"/my-project-1 "$GUIX_EXTRA_PROFILES"/my-project-2; do
  profile=$i/$(basename "$i")
  if [ -f "$profile"/etc/profile ]; then
    GUIX_PROFILE="$profile"
    . "$GUIX_PROFILE"/etc/profile
  fi
  unset profile
done
```

Wenn ein Profil abgeschaltet ist, lässt es sich mit Leichtigkeit für eine bestimmte Shell aktivieren, ohne die restliche Benutzersitzung zu „verschmutzen“:

```
GUIX_PROFILE="pfad/zu/my-project" ; . "$GUIX_PROFILE"/etc/profile
```

Der Schlüssel dazu, wie man ein Profil aktiviert, ist dessen '`etc/profile`'-Datei mit `source` zu laden. Diese Datei enthält einige Shell-Befehle, um die für das Aktivieren der Software im Profil nötigen Umgebungsvariablen zu exportieren. Die Datei wird durch Guix automatisch erzeugt, um mit `source` eingelesen zu werden. Sie enthält dieselben Variablen, die Sie nach Ausführung dieses Befehls bekämen:

```
guix package --search-paths=prefix --profile=$my_profile"
```

Siehe auch hier das Abschnitt “Aufruf von guix package” in *Referenzhandbuch zu GNU Guix* für die Befehlszeilenoptionen.

Um ein Profil zu aktualisieren, installieren Sie das Manifest einfach nochmal:

```
guix package -m /path/to/guix-my-project-manifest.scm \
 -p "$GUIX_EXTRA_PROFILES"/my-project/my-project
```

Um alle Profile zu aktualisieren, genügt es, sie in einer Schleife durchlaufen zu lassen. Nehmen wir zum Beispiel an, Ihre Manifestspezifikationen befinden sich in `~/.guix-manifests/guix-$profile-manifest.scm`, wobei '`$profile`' der Name des Profils ist (z.B. „projekt1“), dann könnten Sie in der Bourne-Shell Folgendes tun:

```
for profile in "$GUIX_EXTRA_PROFILES"/*; do
  guix package --profile="$profile" \
   --manifest="$HOME/.guix-manifests/guix-$profile-manifest.scm"
done
```

Jedes Profil verfügt über seine eigenen Generationen:

```
guix package -p "$GUIX_EXTRA_PROFILES"/my-project/my-project --list-generations
```

Sie können es auf jede Generation zurücksetzen:

```
guix package -p "$GUIX_EXTRA_PROFILES"/my-project/my-project --switch-generations=17
```

Zu guter Letzt ist es möglich, zu einem Profil zu wechseln ohne die aktuelle Umgebung zu erben, indem Sie es aus einer leeren Shell heraus aktivieren:

```
env -i $(which bash) --login --noprofile --norc
. my-project/etc/profile
```

6.1.2 Die nötigen Pakete

Das Aktivieren eines Profils bedeutet im Grunde, dass eine Menge Umgebungsvariabler exportiert wird. Diese Rolle fällt der '`etc/profile`'-Datei innerhalb des Profils zu.

Anmerkung: Nur diejenigen Umgebungsvariablen der sie gebrauchenden Pakete werden gesetzt.

Zum Beispiel wird kein '`MANPATH`' gesetzt sein, wenn keine Anwendung im Profil diese „Man-Pages“ (Handbuchseiten) gebraucht. Wenn Sie also transparenten Zugriff auf Handbuchseiten brauchen, nachdem das Profil geladen wurde, dann gibt es zwei Möglichkeiten:

- Entweder Sie exportieren die Variablen von Hand, z.B.

  ```
  export MANPATH=/path/to/profile${MANPATH:+:}$MANPATH
  ```

- Oder Sie schreiben '`man-db`' in das Profilmanifest hinein.

Das Gleiche gilt für '`INFOPATH`' (Sie können '`info-reader`' installieren), '`PKG_CONFIG_PATH`' (installieren Sie '`pkg-config`'), etc.

6.1.3 Vorgabeprofil

Was ist mit dem Standardprofil, das Guix in `~/.guix-profile` aufbewahrt?

Sie können ihm die Rolle zuweisen, die Sie wollen. Normalerweise würden Sie das Manifest derjenigen Pakete installieren, die Sie ständig benutzen möchten.

Alternativ können Sie es ohne Manifest für Wegwerfpakete benutzen, die Sie nur ein paar Tage lang benutzen wollen. Das macht es leicht,

```
guix install paket-foo
guix upgrade paket-bar
```

auszuführen ohne den Pfad zu einem Profil festzulegen.

6.1.4 Der Vorteil von Manifesten

Mit Manifesten können Sie *deklarativ* angeben, welche Pakete Sie in Ihrem Profil haben möchten (siehe Abschnitt "Manifeste verfassen" in *Referenzhandbuch zu GNU Guix*). Sie sind eine bequeme Art, Ihre Paketlisten zur Hand zu haben und diese z.B. über mehrere Maschinen hinweg in einem Versionskontrollsystem zu synchronisieren.

Eine oft gehörte Beschwerde über Manifeste ist, dass es lange dauert, sie zu installieren, wenn sie viele Pakete enthalten. Das ist besonders hinderlich, wenn Sie nur ein einziges Paket in ein großes Manifest installieren möchten.

Das ist ein weiteres Argument dafür, mehrere Profile zu benutzen, denn es stellt sich heraus, dass dieses Vorgehen perfekt für das Aufbrechen von Manifesten in mehrere Mengen semantisch verbundener Pakete geeignet ist. Mit mehreren, kleinen Profilen haben Sie mehr Flexibilität und Benutzerfreundlichkeit.

Manifeste haben mehrere Vorteile. Insbesondere erleichtern sie die Wartung.

- Wenn ein Profil aus einem Manifest heraus eingerichtet wird, ist das Manifest selbst genug, um eine Liste der Pakete zur Verfügung zu haben und das Profil später auf einem anderen System zu installieren. Bei *ad-hoc*-Profilen müssten wir hingegen eine Manifestspezifikation von Hand schreiben und uns um die Paketversionen derjenigen Pakete kümmern, die nicht die vorgegebene Version verwenden.
- Bei `guix package --upgrade` wird immer versucht, die Pakete zu aktualisieren, die propagierte Eingaben haben, selbst wenn es nichts zu tun gibt. Mit Guix-Manifesten fällt dieses Problem weg.
- Wenn man nur Teile eines Profils aktualisiert, kann es zu Konflikten kommen (weil die Abhängigkeiten zwischen aktualisierten und nicht aktualisierten Paketen voneinander abweichen), und es kann mühsam sein, diese Konflikte von Hand aufzulösen. Manifeste haben kein solches Problem, weil alle Pakete immer gleichzeitig aktualisiert werden.
- Wie zuvor erwähnt, gewähren einem Manifeste reproduzierbare Profile, während die imperativen `guix install`, `guix upgrade`, etc. das nicht tun, weil sie jedes Mal ein anderes Profil ergeben, obwohl sie dieselben Pakete enthalten. Siehe die dieses Thema betreffende Diskussion (`https://issues.guix.gnu.org/issue/33285`).
- Manifestspezifikationen können von anderen '`guix`'-Befehlen benutzt werden. Zum Beispiel können Sie `guix weather -m manifest.scm` ausführen, um zu sehen, wie viele Substitute verfügbar sind, was Ihnen bei der Entscheidung helfen kann, ob Sie heute schon eine Aktualisierung durchführen oder lieber noch eine Weile warten möchten. Ein anderes Beispiel: Sie können mit `guix pack -m manifest.scm` ein Bündel erzeugen, das alle Pakete im Manifest enthält (mitsamt derer transitiven Referenzen).
- Zuletzt haben Manifeste auch eine Repräsentation in Scheme, nämlich den '`<manifest>`'-Verbundstyp. Sie können in Scheme verarbeitet werden und an die verschiedenen Guix-Programmierschnittstellen (APIs) (`https://de.wikipedia.org/wiki/Programmierschnittstelle`) übergeben werden.

Es ist wichtig, dass Sie verstehen, dass Manifeste zwar benutzt werden können, um Profile zu deklarieren, sie aber nicht ganz dasselbe wie Profile sind: Profile haben Nebenwirkungen. Sie setzen Pakete im Store fest, so dass sie nicht vom Müllsammler geholt werden (siehe Abschnitt "Aufruf von guix gc" in *Referenzhandbuch zu GNU Guix*) und stellen sicher, dass sie auch in Zukunft jederzeit verfügbar sein werden. Wenn Sie den Befehl `guix`

`shell` verwenden, werden damit erzeugte kürzlich verwendete Profile allerdings auch vor dem Müllsammler beschützt; Profile, die länger nicht verwendet werden, können jedoch zusammen mit ihren referenzierten Paketen von ihm gelöscht werden.

Wenn wir uns 100% sicher sein wollen, dass der Müllsammler ein bestimmtes Profil nicht sammelt, müssen wir das Manifest in ein Profil installieren und `GUIX_PROFILE=/das/profil; . "$GUIX_PROFILE"/etc/profile` aufrufen, wie oben erklärt. Dadurch haben wir die Garantie, dass unsere Hacking-Umgebung jederzeit zur Verfügung steht.

Sicherheitswarnung: Obwohl es angenehm sein kann, alte Profile zu behalten, sollten Sie daran denken, dass veraltete Pakete *nicht* über die neuesten Sicherheitsbehebungen verfügen.

6.1.5 Reproduzierbare Profile

Um ein Profil Bit für Bit nachzubilden, brauchen wir zweierlei Informationen:

- ein Manifest (siehe Abschnitt “Manifeste verfassen” in *Referenzhandbuch zu GNU Guix*) und
- eine Kanalspezifikation für Guix (siehe siehe Abschnitt “Guix nachbilden” in *Referenzhandbuch zu GNU Guix*).

Tatsächlich kann es vorkommen, dass ein Manifest allein nicht genug ist: Verschiedene Versionen von Guix (oder andere Kanäle) können beim selben Manifest zu verschiedenen Ausgaben führen.

Sie können sich die Guix-Kanalspezifikationen mit ‘`guix describe --format=channels`’ ausgeben lassen (siehe Abschnitt “Aufruf von guix describe” in *Referenzhandbuch zu GNU Guix*). Speichern Sie sie in eine Datei ab, sagen wir ‘`channel-specs.scm`’.

Auf einem anderen Rechner können Sie die Kanalspezifikationsdatei und das Manifest benutzen, um genau dasselbe Profil zu reproduzieren:

```
GUIX_EXTRA_PROFILES=$HOME/.guix-extra-profiles
GUIX_EXTRA=$HOME/.guix-extra

mkdir -p "$GUIX_EXTRA"/my-project
guix pull --channels=channel-specs.scm --profile="$GUIX_EXTRA/my-project/guix"

mkdir -p "$GUIX_EXTRA_PROFILES/my-project"
"$GUIX_EXTRA"/my-project/guix/bin/guix package \
 --manifest=/path/to/guix-my-project-manifest.scm \
 --profile="$GUIX_EXTRA_PROFILES"/my-project/my-project
```

Es kann nichts Schlimmes passieren, wenn Sie das Guix-Kanalprofil, das Sie eben aus der Kanalspezifikation erstellt haben, löschen, denn das Projektprofil hängt davon nicht ab.

7 Software-Entwicklung

Guix ist ein hilfreiches Werkzeug für Entwickler; besonders `guix shell` versorgt Sie mit einer eigenständigen Entwicklungsumgebung für Ihr Paket, unabhängig von der oder den Sprachen, in denen es programmiert ist (siehe Abschnitt "Aufruf von guix shell" in *Referenzhandbuch zu GNU Guix*). Um Ihren Nutzen daraus zu ziehen, fertigen Sie als Erstes eine Paketdefinition an, die entweder ins eigentliche Guix akzeptiert werden muss oder die Teil eines Kanals oder gleich im Quellbaum Ihres Projekts sein muss in einer Datei `guix.scm`. Letztere Option bietet sich an, weil Entwickler dann nur das Repository des Projekts zu klonen brauchen und `guix shell` ohne Argumente aufrufen können.

Zur Entwicklung gehört jedoch mehr. Wie richten Entwickler eine kontinuierliche Integration ihres Codes in Guix-Erstellungsumgebungen ein? Wie findet eine sofortige Auslieferung ihres Codes an abenteuerlustige Nutzer statt? Dieses Kapitel erklärt, wie Entwickler wenige zusätzliche Dateien ins Repository hinzufügen, um Guix-basierte Entwicklungsumgebungen, kontinuierliche Integration und kontinuierliche Auslieferung umzusetzen – alles auf einmal[1].

7.1 Einstieg

Wie lässt sich also ein Repository „guixifizieren"? Unser erster Schritt ist, wie wir sehen konnten, eine Datei `guix.scm` im obersten Verzeichnis des fraglichen Repositorys unterzubringen. Nehmen wir Guile (`https://www.gnu.org/software/guile`) als Beispiel in diesem Kapitel; es ist in Scheme (zum größten Teil) und in C geschrieben und ist von anderer Software abhängig – einer C-Compiler-Toolchain, C-Bibliotheken, Autoconf und seinen Freunden, LaTeX und so weiter. Daraus ergibt sich eine `guix.scm`, die ziemlich wie andere Paketdefinitionen auch aussieht (siehe Abschnitt "Pakete definieren" in *Referenzhandbuch zu GNU Guix*), nur fehlt das `define-public` am Anfang:

```
;; Die ,guix.scm‘-Datei für Guile, gedacht für ,guix shell‘.

(use-modules (guix)
             (guix build-system gnu)
             ((guix licenses) #:prefix license:)
             (gnu packages autotools)
             (gnu packages base)
             (gnu packages bash)
             (gnu packages bdw-gc)
             (gnu packages compression)
             (gnu packages flex)
             (gnu packages gdb)
             (gnu packages gettext)
             (gnu packages gperf)
             (gnu packages libffi)
             (gnu packages libunistring)
```

[1] Dieses Kapitel ist eine aufgearbeitete Fassung eines Blog-Eintrags (`https://guix.gnu.org/de/blog/2023/from-development-environments-to-continuous-integrationthe-ultimate-guide-to-software-development-with-`), der im Juni 2023 auf Guix' Webauftritt veröffentlicht wurde.

```
            (gnu packages linux)
            (gnu packages pkg-config)
            (gnu packages readline)
            (gnu packages tex)
            (gnu packages texinfo)
            (gnu packages version-control))

(package
  (name "guile")
  (version "3.0.99-git")                          ;komische Versionsnummer
  (source #f)                                     ;keine Quelle
  (build-system gnu-build-system)
  (native-inputs
   (append (list autoconf
                 automake
                 libtool
                 gnu-gettext
                 flex
                 texinfo
                 texlive-base                ;für "make pdf"
                 texlive-epsf
                 gperf
                 git
                 gdb
                 strace
                 readline
                 lzip
                 pkg-config)

           ;; Zum Cross-Kompilieren ist eine native Version desselben
           ;; Guiles notwendig.
           (if (%current-target-system)
               (list this-package)
               '())))
  (inputs
   (list libffi bash-minimal))
  (propagated-inputs
   (list libunistring libgc))

  (native-search-paths
   (list (search-path-specification
          (variable "GUILE_LOAD_PATH")
          (files '("share/guile/site/3.0")))
         (search-path-specification
          (variable "GUILE_LOAD_COMPILED_PATH")
          (files '("lib/guile/3.0/site-ccache")))))
  (synopsis "Scheme implementation intended especially for extensions")
```

```
  (description
   "Guile is the GNU Ubiquitous Intelligent Language for Extensions,
and it's actually a full-blown Scheme implementation!")
  (home-page "https://www.gnu.org/software/guile/")
  (license license:lgpl3+))
```

Das ist schon ein bisschen aufwendig, aber wenn jemand an Guile hacken will, braucht sie bloß noch das hier aufzurufen:

```
guix shell
```

Sie findet sich in einer Shell wieder, wo es alle Abhängigkeiten von Guile gibt: die oben aufgeführten Abhängigkeiten und außerdem *implizite Abhängigkeiten* wie die GCC-Toolchain, GNU Make, sed, grep und so weiter. Siehe Abschnitt "Aufruf von guix shell" in *Referenzhandbuch zu GNU Guix* für weitere Informationen zu `guix shell`.

Empfehlung des Hauses: Wir legen Ihnen nahe, Entwicklungsumgebungen mit diesem Befehl anzulegen:

```
guix shell --container --link-profile
```

... oder kurz:

```
guix shell -CP
```

So erhält man eine Shell in einem isolierten Container und alle Abhängigkeiten finden sich in `$HOME/.guix-profile`, was die Nutzung von Zwischenspeichern wie `config.cache` (siehe Abschnitt "Cache Files" in *Autoconf*) erleichtert und wodurch absolute Dateinamen, die in erzeugten Dateien wie `Makefile`s und Ähnlichem festgehalten werden, gültig bleiben. Die Tatsache, dass die Shell in einem Container läuft, erlaubt Ihnen ein Gefühl der Sicherheit: Nichts außer dem aktuellen Verzeichnis und Guiles Abhängigkeiten ist in dieser isolierten Umgebung sichtbar; nichts vom System kann Ihre Arbeit störend beeinflussen.

7.2 Stufe 1: Erstellen mit Guix

Jetzt, wo wir eine Paketdefinition vorliegen haben (siehe Abschnitt 7.1 [Einstieg], Seite 70), können wir sie dann auch gleich benutzen, um Guile mit Guix zu erstellen? Uns fehlt der Eintrag im `source`-Feld, das wir leer gelassen haben, denn für `guix shell` spielen nur die *inputs* unseres Pakets eine Rolle – um die Entwicklungsumgebung zu unserem Paket aufzusetzen –, aber das Paket selber wurde bisher nicht gebraucht.

Wenn wir das Paket mit Guix erstellt bekommen möchten, müssen wir etwas ins `source`-Feld eintragen, etwa so:

```
(use-modules (guix)
             (guix git-download)  ;für ,git-predicate‘
             ...)

(define vcs-file?
  ;; Wahr zurückliefern, wenn die angegebene Datei unter
  ;; Versionskontrolle steht.
  (or (git-predicate (current-source-directory))
      (const #t)))                              ;es ist kein Git-Checkout
```

```
(package
  (name "guile")
  (version "3.0.99-git")                          ;komische Versionsnummer
  (source (local-file "." "guile-checkout"
                      #:recursive? #t
                      #:select? vcs-file?))
  ...)
```

Dies sind unsere Änderungen verglichen mit dem vorherigen Abschnitt:

1. Wir haben `(guix git-download)` zu unseren importierten Modulen hinzugefügt, damit wir dessen Prozedur `git-predicate` benutzen können.
2. Wir haben `vcs-file?` als Prozedur definiert, die wahr zurückgibt, wenn ihr eine Datei übergeben wird, die unter Versionskontrolle steht. Es gehört sich, den Fall abzufangen, wenn wir uns in keinem Git-Checkout befinden: Dann wird immer wahr geliefert.
3. Wir setzen `source` auf `local-file` (`https://guix.gnu.org/manual/devel/de/html_node/G_002dAusdrucke.html#index-local_002dfile`) – einer rekursiven Kopie des aktuellen Verzeichnisses (`"."`), eingeschränkt auf solche Dateien, die unter Versionskontrolle gestellt sind (mit `#:select?`).

Von da an erfüllt `guix.scm` einen zweiten Zweck: Wir können die Software mit Guix erstellen. Der Vorteil der Erstellung mit Guix ist, dass eine „saubere“ Erstellung durchgeführt wird – Sie können sich sicher sein, das Ergebnis der Erstellung beruht nicht auf Dateien in Ihrem Quellbaum oder anderen Dingen in Ihrem System – und auch, dass Sie viele Möglichkeiten testen können. Als Erstes wäre da eine einfache native Erstellung:

```
guix build -f guix.scm
```

Aber auch für ein anderes System können Sie die Erstellung durchführen (unter Umständen müssen Sie erst die Auslagerungsfunktion, siehe Abschnitt “Auslagern des Daemons einrichten” in *Referenzhandbuch zu GNU Guix*, oder transparente Emulation, siehe Abschnitt “Virtualisierungsdienste” in *Referenzhandbuch zu GNU Guix*, einrichten):

```
guix build -f guix.scm -s aarch64-linux -s riscv64-linux
```

... oder Sie cross-kompilieren:

```
guix build -f guix.scm --target=x86_64-w64-mingw32
```

Möglich sind auch *Paketumwandlungsoptionen*, womit Sie Varianten Ihres Pakets testen können (siehe Abschnitt “Paketumwandlungsoptionen” in *Referenzhandbuch zu GNU Guix*):

```
# Wie sieht es aus, wenn wir mit Clang statt GCC erstellen?
guix build -f guix.scm \
  --with-c-toolchain=guile@3.0.99-git=clang-toolchain

# Was ist mit wenig getesteten configure-Befehlszeilenoptionen?
guix build -f guix.scm \
  --with-configure-flag=guile@3.0.99-git=--disable-networking
```

Praktisch!

7.3 Stufe 2: Das Repository als Kanal

Jetzt haben wir ein Git-Repository mit (unter anderem) einer Paketdefinition (siehe Abschnitt 7.2 [Erstellen mit Guix], Seite 72). Wäre es nicht besser, wenn wir daraus einen *Kanal* machen würden (siehe Abschnitt "Kanäle" in *Referenzhandbuch zu GNU Guix*)? Schließlich sind Kanäle entwickelt worden, um Nutzer mit Paketdefinitionen zu versorgen, und genau das tun wir mit `guix.scm`.

Tatsächlich sollten wir es zu einem Kanal machen, aber aufgepasst: Die `.scm`-Datei(en) unseres Kanals gehören in ein gesondertes Verzeichnis, damit `guix pull` nicht versucht, die falschen `.scm`-Dateien zu laden, wenn jemand damit den Kanal herunterlädt – und in Guile gibt es viele `.scm`-Dateien! Wir fangen mit der Trennung an, behalten aber auf oberster Ebene eine symbolische Verknüpfung `guix.scm` zur Nutzung mit `guix shell`:

```
mkdir -p .guix/modules
mv guix.scm .guix/modules/guile-package.scm
ln -s .guix/modules/guile-package.scm guix.scm
```

Damit es als Kanal verwendet werden kann, erweitern wir unsere `guix.scm`-Datei zu einem *Paketmodul* (siehe Abschnitt "Paketmodule" in *Referenzhandbuch zu GNU Guix*). Das geschieht, indem wir die `use-modules`-Form am Anfang durch eine `define-module`-Form ersetzen. Auch müssen wir dann eine Paketvariable *exportieren*, mit `define-public`, und dennoch am Ende der Datei den Paketwert zurückliefern, damit `guix shell` und `guix build -f guix.scm` weiterhin funktionieren. Das Endergebnis sieht so aus (ohne zu wiederholen, was wir nicht verändern):

```
(define-module (guile-package)
  #:use-module (guix)
  #:use-module (guix git-download)   ;für ,git-predicate‘
  ...)

(define vcs-file?
  ;; Wahr zurückliefern, wenn die angegebene Datei unter
  ;; Versionskontrolle steht.
  (or (git-predicate (dirname (dirname (current-source-directory))))
      (const #t)))                              ;es ist kein Git-Checkout

(define-public guile
  (package
    (name "guile")
    (version "3.0.99-git")                        ;komische Versionsnummer
    (source (local-file "../.." "guile-checkout"
                        #:recursive? #t
                        #:select? vcs-file?))
    ...))

;; Das oben definierte Paketobjekt muss am Modulende zurückgeliefert werden.
guile
```

Zuletzt brauchen wir noch eine `.guix-channel`-Datei (`https://guix.gnu.org/manual/devel/de/html_node/Paketmodule-in-einem-Unterverzeichnis.html`), um Guix zu erklären, woher es die Paketmodule in unserem Repository bekommt:

```
;; Mit dieser Datei kann man dieses Repo als Guix-Kanal eintragen.

(channel
  (version 0)
  (directory ".guix/modules"))  ;Paketmodule finden sich unter .guix/modules/
```

Zusammengefasst liegen diese Dateien jetzt im Repository:

```
.
 .guix-channel
 guix.scm → .guix/modules/guile-package.scm
 .guix
     modules
        guile-package.scm
```

Und das war alles: Wir haben einen Kanal erschaffen! (Noch besser wäre es, auch *Kanalautorisierungen* (`https://guix.gnu.org/en/manual/devel/de/html_node/Kanalautorisierungen-angeben.html`) zu unterstützen, damit Benutzer wissen, dass der heruntergeladene Code echt ist. Wir ersparen Ihnen die Feinheiten hier, aber ziehen Sie es in Betracht!) Nutzer können von diesem Kanal pullen, indem sie ihn zu `~/.config/guix/channels.scm` hinzufügen (`https://guix.gnu.org/manual/devel/de/html_node/Weitere-Kanale-angeben.html`), etwa so:

```
(append (list (channel
                (name 'guile)
                (url "https://git.savannah.gnu.org/git/guile.git")
                (branch "main")))
        %default-channels)
```

Nach einem Aufruf von `guix pull` sind die neuen Pakete zu sehen:

```
$ guix describe
Generation 264  26. Mai 2023 16:00:35    (aktuell)
  guile 36fd2b4
    Repository-URL: https://git.savannah.gnu.org/git/guile.git
    Branch: main
    Commit: 36fd2b4920ae926c79b936c29e739e71a6dff2bc
  guix c5bc698
    Repository-URL: https://git.savannah.gnu.org/git/guix.git
    Commit: c5bc698e8922d78ed85989985cc2ceb034de2f23
$ guix package -A ^guile$
guile   3.0.99-git      out,debug       guile-package.scm:51:4
guile   3.0.9           out,debug       gnu/packages/guile.scm:317:2
guile   2.2.7           out,debug       gnu/packages/guile.scm:258:2
guile   2.2.4           out,debug       gnu/packages/guile.scm:304:2
guile   2.0.14          out,debug       gnu/packages/guile.scm:148:2
guile   1.8.8           out             gnu/packages/guile.scm:77:2
$ guix build guile@3.0.99-git
```

```
[...]
/gnu/store/axnzbl89yz7ld78bmx72vpqp802dwsar-guile-3.0.99-git-debug
/gnu/store/r34gsij7f0glg2fbakcmmk0zn4v62s5w-guile-3.0.99-git
```

Auf diesem Weg liefern Sie, als Entwickler, Ihre Software direkt in die Hände der Nutzer! Es gibt keine Mittelsmänner, trotzdem bleiben Transparenz und Provenienzverfolgung erhalten.

Auf dieser Grundlage kann außerdem jeder leicht Docker-Abbilder, Pakete als Deb/RPM oder einen einfachen Tarball mit `guix pack` anfertigen (siehe Abschnitt "Aufruf von guix pack" in *Referenzhandbuch zu GNU Guix*):

```
# Sie wünschen ein Docker-Image mit unserem Guile-Snapshot?
guix pack -f docker -S /bin=bin guile@3.0.99-git

# Und ein verschiebliches RPM?
guix pack -f rpm -R -S /bin=bin guile@3.0.99-git
```

7.4 Bonus: Paketvarianten

Jetzt, wo wir es zu einem richtigen Kanal gebracht haben (siehe Abschnitt 7.3 [Das Repository als Kanal], Seite 74), drängt sich die Frage auf, warum er nur ein Paket enthält. Da geht mehr, denn wir können gleich mehrere *Paketvarianten* (siehe Abschnitt "Paketvarianten definieren" in *Referenzhandbuch zu GNU Guix*) in unserer Datei `guile-package.scm` festlegen, und zwar Varianten, die wir Entwickler von Guile gerne testen würden – wofür wir oben noch Paketumwandlungsoptionen gebraucht haben. Wir können Sie auf diese Weise eintragen:

```
;; Dies ist die Datei ,.guix/modules/guile-package.scm‘.

(define-module (guile-package)
  ...)

(define-public guile
  ...)

(define (package-with-configure-flags p flags)
  "Liefert P erweitert um die 'configure'-Optionen in FLAGS."
  (package/inherit p
    (arguments
     (substitute-keyword-arguments (package-arguments p)
       ((#:configure-flags original-flags #~(list))
        #~(append #$original-flags #$flags))))))

(define-public guile-without-threads
  (package
    (inherit (package-with-configure-flags guile
                                           #~(list "--without-threads")))
    (name "guile-without-threads")))
```

```
(define-public guile-without-networking
  (package
    (inherit (package-with-configure-flags guile
                                           #~(list "--disable-networking")))
    (name "guile-without-networking")))


;; Das oben definierte Paketobjekt muss am Modulende zurückgeliefert werden.
guile
```

Sobald wir den Kanal mit `guix pull` aktiviert haben, können wir diese Varianten als normale Pakete erstellen. Es geht aber auch, einen Befehl wie hier in einem Checkout von Guile auf oberster Ebene auszuführen:

```
guix build -L $PWD/.guix/modules guile-without-threads
```

7.5 Stufe 3: Kontinuierliche Integration einrichten

Mit dem oben definierten Kanal (siehe Abschnitt 7.3 [Das Repository als Kanal], Seite 74) können wir erst recht etwas anstellen, sobald wir *Kontinuierliche Integration* (`https://de.wikipedia.org/wiki/Kontinuierliche_Integration`) (Continuous Integration, CI) eingerichtet haben. Hier gibt es mehrere Wege zum Ziel.

Sie können ein dem Mainstream entsprechendes Werkzeug zur kontinuierlichen Integration nehmen wie GitLab-CI. Dazu müssen Sie Aufgaben („Jobs“) in einem Docker-Abbild oder einer virtuellen Maschine, auf der Guix installiert ist, ausführen lassen. Wenn wir das für Guile vorhätten, legten wir eine Aufgabe an, die einen Shell-Befehl wie diesen laufen lässt:

```
guix build -L $PWD/.guix/modules guile@3.0.99-git
```

So klappt es gut und der Vorteil ist, dass Sie den Befehl leicht Ihrer bevorzugten CI-Plattform beibringen können.

Aber trotzdem können Sie Guix am besten ausnutzen, wenn Sie Cuirass (`https://guix.gnu.org/en/cuirass`) verwenden, ein CI-Werkzeug speziell für Guix und eng damit integriert. Cuirass zu benutzen statt einem CI-Werkzeug, das jemand für Sie hostet, macht mehr Aufwand, aber die Einrichtung hält sich in Grenzen, wenn Sie Cuirass' Dienst auf Guix System benutzen (siehe Abschnitt "Kontinuierliche Integration" in *Referenzhandbuch zu GNU Guix*). Kehren wir zu unserem Beispiel zurück, statten wir Cuirass mit so einer Spezifikationsdatei aus:

```
;; Cuirass-Spezifikationsdatei, um alle Pakete des ‚guile‘-Kanals zu erstellen.
(list (specification
        (name "guile")
        (build '(channels guile))
        (channels
         (append (list (channel
                        (name 'guile)
                        (url "https://git.savannah.gnu.org/git/guile.git")
                        (branch "main")))
                 %default-channels))))
```

Es gibt zwei wichtige Unterschiede zu dem, was Sie mit anderen CI-Werkzeugen tun würden:

- Cuirass weiß über *zwei* Kanäle Bescheid, `guile` und `guix`. Tatsächlich zählen viele Pakete vom `guix`-Kanal zu den Abhängigkeiten des von uns entwickelten `guile`-Pakets – GCC, GNU libc, libffi und weitere. Es kann passieren, dass sich die Pakete vom `guix`-Kanal ändern und dadurch die Erstellung unseres `guile`-Pakets schiefgeht; so etwas wollen wir als die Entwickler von Guile so früh erfahren, wie es nur geht.
- Die Erstellungsergebnisse werden *nicht* entsorgt, sondern sie können als *Substitute*, d.h. vorerstellten Binärdateien, allen Nutzern unseres `guile`-Kanals transparent zur Verfügung gestellt werden! (Siehe Abschnitt "Substitute" in *Referenzhandbuch zu GNU Guix* für Hintergrundwissen zu Substituten.)

Aus Entwicklersicht sieht das Ergebnis am Ende so aus wie die auf Guiles Status-Seite (`https://ci.guix.gnu.org/jobset/guile`) aufgelisteten *Auswertungen*: Jede Auswertung besteht aus einer Kombination von Commits der `guix`- und `guile`-Kanäle, wovon es mehrere *Aufträge* (Jobs) gibt – je einen Job pro Paket, das in `guile-package.scm` definiert ist, mal der Anzahl der Zielarchitekturen.

Substitute gibt es kostenlos dazu! Zum Beispiel kann jeder, weil unser `guile`-Jobset auf ci.guix.gnu.org erstellt wird und dort `guix publish` (siehe Abschnitt "Aufruf von guix publish" in *Referenzhandbuch zu GNU Guix*) zusätzlich zu Cuirass läuft, automatisch die Substitute für `guile`-Erstellungen von ci.guix.gnu.org beziehen; es ist dazu kein Mehraufwand nötig.

7.6 Bonus: Erstellungs-Manifest

Mit der Cuirass-Spezifikation oben können wir uns gut anfreunden: Jedes Paket in unserem Kanal wird erstellt, einschließlich einiger Varianten (siehe Abschnitt 7.5 [Kontinuierliche Integration einrichten], Seite 77). Allerdings können wir in manchen Fällen noch nicht alles ausdrücken, was wir möchten, etwa wenn wir in manchen Cuirass-Jobs cross-kompilieren möchten, Transformationen festlegen oder Docker-Abbilder, RPM-/Deb-Pakete oder sogar Systemtests wollen.

Das erreichen Sie, indem Sie ein *Manifest* mit den Angaben schreiben (siehe Abschnitt "Manifeste verfassen" in *Referenzhandbuch zu GNU Guix*). Das Manifest, was wir für Guile haben, hat Einträge für die oben definierten Paketvarianten sowie zusätzliche Varianten und Cross-Erstellungen:

```
;; Dies ist ,.guix/manifest.scm‘.

(use-modules (guix)
             (guix profiles)
             (guile-package))   ;unser eigenes Paketmodul importieren

(define* (package->manifest-entry* package system
                                   #:key target)
  "Gibt einen Manifest-Eintrag für das Paket PACKAGE für SYSTEM
zurück. Optional wird für das Zielsystem TARGET cross-kompiliert."
  (manifest-entry
    (inherit (package->manifest-entry package))
```

```
      (name (string-append (package-name package) "." system
                           (if target
                               (string-append "." target)
                               "")))
      (item (with-parameters ((%current-system system)
                              (%current-target-system target))
              package))))

(define native-builds
  (manifest
   (append (map (lambda (system)
                  (package->manifest-entry* guile system))

                '("x86_64-linux" "i686-linux"
                  "aarch64-linux" "armhf-linux"
                  "powerpc64le-linux"))
           (map (lambda (guile)
                  (package->manifest-entry* guile "x86_64-linux"))
                (cons (package
                        (inherit (package-with-c-toolchain
                                  guile
                                  `(("clang-toolchain"
                                     ,(specification->package
                                       "clang-toolchain")))))
                        (name "guile-clang"))
                      (list guile-without-threads
                            guile-without-networking
                            guile-debug
                            guile-strict-typing))))))

(define cross-builds
  (manifest
   (map (lambda (target)
          (package->manifest-entry* guile "x86_64-linux"
                                    #:target target))
        '("i586-pc-gnu"
          "aarch64-linux-gnu"
          "riscv64-linux-gnu"
          "i686-w64-mingw32"
          "x86_64-linux-gnu"))))

(concatenate-manifests (list native-builds cross-builds))
```

Wir gehen nicht auf die Details dieses Manifests ein; es genügt zu wissen, dass wir dadurch mehr Flexibilität haben. Wir müssen jetzt Cuirass die Information geben, dass damit dieses Manifest erstellt werden soll; dazu verwenden wir eine leicht abgeänderte Spezifikation verglichen mit vorher:

```
;; Cuirass-Spezifikationsdatei, um alle Pakete des ‚guile‘-Kanals zu erstellen.
(list (specification
        (name "guile")
        (build '(manifest ".guix/manifest.scm"))
        (channels
         (append (list (channel
                        (name 'guile)
                        (url "https://git.savannah.gnu.org/git/guile.git")
                        (branch "main")))
                 %default-channels))))
```

Wir haben im Teil `(build ...)` der Spezifikation jetzt `'(manifest ".guix/manifest.scm")` verwendet, damit unser Manifest genommen wird, und das war schon alles!

7.7 Zusammenfassung

In diesem Kapitel haben wir die Vorgehensweise auf Guile angewandt und das Ergebnis lässt sich hier bestaunen:

- `.guix-channel` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix-channel?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`),
- `.guix/modules/guile-package.scm` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix/modules/guile-package.scm?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`) auch symbolisch verknüpft als `guix.scm` auf der oberstem Verzeichnisebene,
- `.guix/manifest.scm` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix/manifest.scm?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`).

Heutzutage werden Repositorys gemeinhin mit Dotfiles für vielerlei Werkzeuge angereichert: `.envrc`, `.gitlab-ci.yml`, `.github/workflows`, `Dockerfile`, `.buildpacks`, `Aptfile`, `requirements.txt` und Weiteres. Es wirkt so, als würden wir Sie zu *noch mehr* Dateien überreden, jedoch können Sie mit diesen Dateien so viel ausdrücken, dass sie die meisten oder alle der oben genannten Dateien *ersetzen*.

Mit wenigen Dateien bekommen wir Unterstützung für:

- Entwicklungsumgebungen (`guix shell`),
- Test-Erstellungen in naturbelassener Umgebung, auch für Paketvarianten und cross-kompiliert (`guix build`),
- kontinuierliche Integration (ob mit Cuirass oder mit einem anderen Werkzeug),
- kontinuierliche Auslieferung an Nutzer (über den Kanal und mit vorerstellten Binärdateien),
- abgeleitete Erstellungsartefakte wie Docker-Abbildern oder Deb-/RPM-Pakete (`guix pack`).

Das ist (wie wir es sehen) ein Schweizer Taschenmesser für reproduzierbare Software-Auslieferung und diese Erklärung sollte aufgezeigt haben, welchen Nutzen Sie als Entwickler daraus gewinnen können!

8 Umgebungen verwalten

Guix liefert mehrere Werkzeuge mit, um die Umgebung zu verwalten. Dieses Kapitel zeigt solche Werkzeuge.

8.1 Guix-Umgebung mit direnv

Guix stellt ein ‘`direnv`’-Paket zur Verfügung, mit der die Shell nach einem Verzeichniswechsel erweitert werden kann. Dieses Werkzeug kann benutzt werden, um eine reine, „pure“ Guix-Umgebung vorzubereiten.

Das folgende Beispiel zeigt eine Shell-Funktion für die `~/.direnvrc`-Datei, die in einer Datei `~/src/guix/.envrc` in Guix' Git-Repository benutzt werden kann, um eine zur Beschreibung im Abschnitt “Erstellung aus dem Git” in *Referenzhandbuch zu GNU Guix* ähnliche Erstellungsumgebung herzustellen.

Erstellen Sie eine `~/.direnvrc` mit einem Bash-Code darin:

```
# Dank an <https://github.com/direnv/direnv/issues/73#issuecomment-152284914>
export_function()
{
  local name=$1
  local alias_dir=$PWD/.direnv/aliases
  mkdir -p "$alias_dir"
  PATH_add "$alias_dir"
  local target="$alias_dir/$name"
  if declare -f "$name" >/dev/null; then
    echo "#!$SHELL" > "$target"
    declare -f "$name" >> "$target" 2>/dev/null
    # Beachten Sie, wir fügen Shell-Variable in den Funktionsauslöser ein.
    echo "$name \$*" >> "$target"
    chmod +x "$target"
  fi
}

use_guix()
{
    # GitHub-Token festlegen.
    export GUIX_GITHUB_TOKEN="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

    # Deaktivieren von 'GUIX_PACKAGE_PATH'.
    export GUIX_PACKAGE_PATH=""

    # Müllsammlerwurzel neu erzeugen.
    gcroots="$HOME/.config/guix/gcroots"
    mkdir -p "$gcroots"
    gcroot="$gcroots/guix"
    if [ -L "$gcroot" ]
    then
```

```
    rm -v "$gcroot"
fi

# Verschiedene Pakete.
PACKAGES_MAINTENANCE=(
    direnv
    git
    git:send-email
    git-cal
    gnupg
    guile-colorized
    guile-readline
    less
    ncurses
    openssh
    xdot
)

# In die Umgebung aufzunehmende Pakete.
PACKAGES=(help2man guile-sqlite3 guile-gcrypt)

# Thanks <https://lists.gnu.org/archive/html/guix-devel/2016-09/msg00859.html>
eval "$(guix shell --search-paths --root="$gcroot" --pure guix \
 ${PACKAGES[@]} ${PACKAGES_MAINTENANCE[@]} "$@")"

# Predefine configure flags.
configure()
{
    ./configure
}
export_function configure

# make ausführen und optional etwas erstellen.
build()
{
    make -j 2
    if [ $# -gt 0 ]
    then
        ./pre-inst-env guix build "$@"
    fi
}
export_function build

# Git-Befehl zum Pushen vordefinieren.
push()
{
    git push --set-upstream origin
```

```
    }
    export_function push

    clear                         # Den Bildschirm löschen.
    git-cal --author='Ihr Name'  # Kalender bisheriger Beiträge zeigen.

    # Befehlsübersicht anzeigen.
    echo "
build          ein Paket oder, ohne Argumente, ein Projekt erstellen
configure      ./configure mit vordefinierten Parametern
push           ins Upstream-Git-Repository pushen
"
}
```

Jedes Projekt, das eine `.envrc` mit einer Zeichenkette `use guix` enthält, wird vordefinierte Umgebungsvariable und Prozeduren verwenden.

Führen Sie `direnv allow` aus, um die Umgebung bei der ersten Nutzung einzurichten.

9 Auf einem Rechencluster installieren

Guix eignet sich sehr für Forscher und jene, die sich mit Hochleistungsrechnen (HPC (High-Performance Computing)) befassen: Komplexe Software-Stacks lassen sich leicht aufspielen und deren Zusammenstellung ist auch noch reproduzierbar – Sie können genau die gleiche Software für andere Maschinen und zu einem späteren Zeitpunkt wieder installieren.

In diesem Kapitel wird erörtert, wie der Systemadministrator eines Clusters auf diesem systemweit Guix zur Verfügung stellen kann, so dass es auf allen Knoten im Cluster benutzt werden kann. Außerdem gehen wir auf mögliche Probleme ein[1].

> **Anmerkung:** Wir gehen hier davon aus, dass auf dem Cluster eine andere GNU/Linux-Distribution als Guix System läuft und wir Guix auf dieser installieren werden.

9.1 Den Zentralrechner konfigurieren

Unsere Empfehlung ist, eine Maschine zum Zentralrechner zu ernennen (als „Head Node“) und auf dieser `guix-daemon` auszuführen, wobei `/gnu/store` über NFS mit den Arbeitsrechnern („Compute Nodes“) geteilt wird.

Zur Erinnerung: `guix-daemon` ist das Hintergrundprogramm, mit dem für Clients Erstellungsprozesse angelegt und Dateien heruntergeladen werden können (siehe Abschnitt “Aufruf des guix-daemon” in *Referenzhandbuch zu GNU Guix*), und das allgemein auf `/gnu/store` zugreift. Dort liegen dann alle Paket-Binärdateien, die irgendeiner der Nutzer erstellen lassen hat (siehe Abschnitt “Der Store” in *Referenzhandbuch zu GNU Guix*). Mit „Clients“ meinen wir die Guix-Befehle, die Benutzer aufgerufen haben, wie etwa `guix install`. Auf einem Cluster können diese Befehle auf den Arbeitsrechnern aufgerufen werden und dennoch werden wir sie mit der zentralen `guix-daemon`-Instanz sprechen lassen.

Legen wir los. Für den Anfang folgen wir auf dem Zentralrechner der Anleitung zur Installation aus einer Binärdatei (siehe Abschnitt “Aus Binärdatei installieren” in *Referenzhandbuch zu GNU Guix*). Zum Glück gibt es das Installationsskript, so dass das schnell gehen dürfte. Wenn die Installation dann abgeschlossen ist, müssen wir daran Anpassungen vornehmen.

Weil wir möchten, dass `guix-daemon` *nicht* nur auf dem Zentralrechner zugänglich ist, sondern von jedem der Arbeitsrechner erreicht wird, richten wir es so ein, dass er auf Verbindungen über TCP/IP lauscht. Dazu bearbeiten wir die systemd-Datei zum Start von `guix-daemon`, `/etc/systemd/system/guix-daemon.service`, und fügen ein Argument `--listen` zur `ExecStart`-Zeile hinzu. Diese sieht jetzt ungefähr so aus:

```
ExecStart=/var/guix/profiles/per-user/root/current-guix/bin/guix-daemon \
    --build-users-group=guixbuild \
    --listen=/var/guix/daemon-socket/socket --listen=0.0.0.0
```

Die Änderungen wirken sich erst aus, wenn der Dienst neu gestartet wurde:

```
systemctl daemon-reload
systemctl restart guix-daemon
```

[1] Das Kapitel ist eine aufgearbeitete Fassung eines Blog-Eintrags auf dem Guix-HPC-Webauftritt von 2017 (`https://hpc.guix.info/blog/2017/11/installing-guix-on-a-cluster/`).

> **Anmerkung:** Mit `--listen=0.0.0.0` ist gemeint, dass `guix-daemon` *alle* auf Port 44146 eingehenden TCP-Verbindungen annimmt (siehe Abschnitt "Aufruf des guix-daemon" in *Referenzhandbuch zu GNU Guix*). Das ist normalerweise auf Rechen-Clustern in Ordnung, weil der Zentralrechner für gewöhnlich ausschließlich vom lokalen Netzwerk des Clusters aus erreichbar ist – unter keinen Umständen darf er für das gesamte Internet zugänglich sein!

Als Nächstes müssen wir unsere NFS-Freigaben in `/etc/exports` (`https://linux.die.net/man/5/exports`) definieren. Dazu fügen wir etwas wie hier hinzu:

```
/gnu/store    *(ro)
/var/guix     *(rw, async)
/var/log/guix *(ro)
```

Es genügt, das Verzeichnis `/gnu/store` nur lesbar zu exportieren, weil es nur durch `guix-daemon` auf dem Hauptknoten jemals verändert wird. In `/var/guix` befinden sich *Benutzerprofile*, die man mit `guix package` anlegen kann. Damit Benutzer also Pakete mit `guix package` installieren können, muss Lese- und Schreibzugriff ermöglicht werden.

Nutzer können so viele Profile anlegen, wie sie möchten, zusätzlich zum Standardprofil, `~/.guix-profile`. Zum Beispiel bekommt jemand, die `guix package -p ~/dev/python-dev -i python` ausführt, Python in einem Profil installiert, das über die symbolische Verknüpfung `~/dev/python-dev` erreichbar ist. Damit dieses Profil *nicht* irgendwann vom Müllsammler weggeräumt wird – d.h. damit Python nicht aus `/gnu/store` entfernt wird, solange dieses Profil existiert –, *müssen die Persönlichen Verzeichnisse in* `/home` *auch auf dem Zentralrechner eingebunden sein*. Wir wollen, dass `guix-daemon` über solche Nicht-Standard-Profile weiß, dass die darin referenzierte Software noch gebraucht wird.

Es kann allerdings sinnvoll sein, Software die *nicht* gebraucht wird, regelmäßig aus `/gnu/store` zu löschen. Verwenden Sie dazu `guix gc` (siehe Abschnitt "Aufruf von guix gc" in *Referenzhandbuch zu GNU Guix*). Dazu können Sie so einen crontab-Eintrag auf dem Zentralknoten vornehmen:

```
root@master# crontab -e
```

... und schreiben Sie etwa:

```
# Jeden Tag um 5 Uhr morgens den Müllsammler schicken,
# damit mindestens 10 GB auf /gnu/store verfügbar sind.
0 5 * * 1  /usr/local/bin/guix gc -F10G
```

So viel zum Zentralrechner! Jetzt geht's an die Arbeitsknoten.

9.2 Die Arbeitsrechner konfigurieren

Zunächst einmal müssen die Arbeitsknoten auch die NFS-Verzeichnisse einbinden, die vom Zentralrechner angeboten werden. Dafür müssen die folgenden Zeilen in `/etc/fstab` (`https://linux.die.net/man/5/fstab`) hinzugefügt werden:

```
zentralrechner:/gnu/store    /gnu/store    nfs  defaults,_netdev,vers=3 0 0
zentralrechner:/var/guix     /var/guix     nfs  defaults,_netdev,vers=3 0 0
zentralrechner:/var/log/guix /var/log/guix nfs  defaults,_netdev,vers=3 0 0
```

... wobei Sie anstelle von *zentralrechner* den Namen oder die IP-Adresse Ihres Zentralrechners hinterlegen. Von da an sollten Sie diese Verzeichnisse, wenn die Einhängepunkte auch existieren, jeweils auf den Arbeitsknoten einbinden können.

Zudem müssen wir unseren Benutzern einen `guix`-Befehl vorinstallieren, den sie verwenden können, wenn sie sich zum ersten Mal mit dem Cluster verbinden (damit werden sie `guix pull` aufrufen, wodurch ihnen ihr „eigener“ `guix`-Befehl zur Verfügung gestellt wird). Analog zum Installationsskript der Guix-Binärdatei auf dem Zentralrechner werden wir unser `guix` ebenfalls in `/usr/local/bin` unterbringen:

```
mkdir -p /usr/local/bin
ln -s /var/guix/profiles/per-user/root/current-guix/bin/guix \
      /usr/local/bin/guix
```

Wir müssen dafür sorgen, dass sich `guix` mit dem Daemon auf dem Zentralrechner verbindet, indem wir diese Zeilen zu `/etc/profile` hinzufügen:

```
GUIX_DAEMON_SOCKET="guix://zentralrechner"
export GUIX_DAEMON_SOCKET
```

Damit unseren Benutzern keine Warnungen angezeigt werden und damit `guix` die richtigen Locale-Spracheinstellungen benutzen kann, müssen wir angeben, wo die von Guix bereitgestellten Locale-Daten zu finden sind (siehe Abschnitt “Anwendungen einrichten” in *Referenzhandbuch zu GNU Guix*):

```
GUIX_LOCPATH=/var/guix/profiles/per-user/root/guix-profile/lib/locale
export GUIX_LOCPATH

# Der Name, den wir hier für die Locale angeben, muss gültig sein.
# Geben Sie "ls $GUIX_LOCPATH/*" ein, um die möglichen Namen zu sehen.
LC_ALL=fr_FR.utf8
export LC_ALL
```

Um die Nutzung zu vereinfachen, legt `guix package` selbstständig die Datei `~/.guix-profile/etc/profile` an, worin alle Umgebungsvariablen definiert sind, um die Pakete zu benutzen – `PATH`, `C_INCLUDE_PATH`, `PYTHONPATH`, etc. Das Gleiche gilt für `guix pull`, was eine Datei innerhalb von `~/.config/guix/current` anlegt. Diese Definitionen sollten Sie daher in `/etc/profile` mit `source` laden lassen:

```
for GUIX_PROFILE in "$HOME/.config/guix/current" "$HOME/.guix-profile"
do
  if [ -f "$GUIX_PROFILE/etc/profile" ]; then
    . "$GUIX_PROFILE/etc/profile"
  fi
done
```

Zu guter Letzt können Guix-Befehle durch Bash und zsh ergänzt werden. In `/etc/bashrc` fügen Sie dazu diese Zeile hinzu:

```
. /var/guix/profiles/per-user/root/current-guix/etc/bash_completion.d/guix
```

Erledigt!

Überprüfen Sie, dass alles klappt, indem Sie sich auf einem der Arbeitsrechner anmelden und dies ausführen:

```
guix install hello
```

Daraufhin sollte der Daemon auf dem Zentralrechner für Sie die vorerstellten Binärdateien herunterladen und nach `/gnu/store` entpacken, wonach `guix install` dann `~/.guix-profile` anlegt und darin wird der Befehl `~/.guix-profile/bin/hello` zu finden sein.

9.3 Netzwerkzugriff

Guix setzt voraus, dass Netzwerkzugriff besteht, um Quellcode und vorerstellte Binärdateien herunterzuladen. Die gute Nachricht ist, dass nur der Zentralrechner Netzwerkzugriff braucht, weil die Arbeitsrechner an diesen delegieren.

Üblicherweise haben Knoten im Cluster höchstens Zugriff auf eine ausgesuchte „Positivliste“ erlaubter Kommunikationspartner. Für den Zentralrechner muss zumindest `ci.guix.gnu.org` auf dieser Positivliste stehen, denn von dort stammen in der Vorgabeeinstellung die vorerstellten Binärdateien für alle Pakete, die das eigentliche Guix kennt.

Nebenbei ist `ci.guix.gnu.org` außerdem ein inhaltsadressierbarer Spiegelserver für die Quelldateien all dieser Pakete. Daher ist es ausreichend, wenn Sie *nur* `ci.guix.gnu.org` auf die Positivliste setzen.

Software-Pakete, die in einem getrennten Repository angeboten werden wie beispielsweise den HPC-Kanälen (`https://hpc.guix.info/channels`), können natürlich nicht von `ci.guix.gnu.org` bezogen werden. Für diese Pakete möchten Sie unter Umständen weitere Server auf die Positivliste setzen, von denen für die fremden Pakete Quell- und Binärdateien heruntergeladen werden können (vorausgesetzt jemand stellt vorerstellte Binärdateien für sie zur Verfügung). Wenn nicht, können Benutzer als Ausweg auch den Quellcode auf ihre Workstations herunterladen und dann in `/gnu/store` auf dem Cluster hochladen. Das geht so:

```
GUIX_DAEMON_SOCKET=ssh://compute-node.example.org \
  guix download http://starpu.gforge.inria.fr/files/starpu-1.2.3/starpu-1.2.3.tar.gz
```

Der obige Befehl lädt `starpu-1.2.3.tar.gz` herunter *und* lässt es über SSH durch die `guix-daemon`-Instanz auf dem Cluster speichern.

Wenn Ihr Cluster gänzlich von der Außenwelt abgeschnitten ist („air-gapped“), ist es aufwendiger. In diesem Fall würden wir dazu raten, dass sie sämtlichen nötigen Quellcode auf eine Workstation herunterladen, auf der Guix läuft. Dafür gibt es zum Beispiel die Befehlszeilenoption `--sources` für `guix build` (siehe Abschnitt “Aufruf von guix build” in *Referenzhandbuch zu GNU Guix*); folgendes Beispiel zeigt, wie Sie sämtlichen Quellcode, von dem das `openmpi`-Paket abhängt, herunterladen:

```
$ guix build --sources=transitive openmpi

...

/gnu/store/xc17sm60fb8nxadc4qy0c7rqph499z8s-openmpi-1.10.7.tar.bz2
/gnu/store/s67jx92lpipy2nfj5cz818xv430n4b7w-gcc-5.4.0.tar.xz
/gnu/store/npw9qh8a46lrxiwh9xwk0wpi3jlzmjnh-gmp-6.0.0a.tar.xz
/gnu/store/hcz0f4wkdbsvsdky3c0vdvcawhdkyldb-mpfr-3.1.5.tar.xz
/gnu/store/y9akh452n3p4w2v631nj0injx7y0d68x-mpc-1.0.3.tar.gz
/gnu/store/6g5c35q8avfnzs3v14dzl54cmrvddjm2-glibc-2.25.tar.xz
/gnu/store/p9k48dk3dvvk7gads7fk30xc2pxsd66z-hwloc-1.11.8.tar.bz2
/gnu/store/cry9lqidwfrfmgl0x389cs3syr15p13q-gcc-5.4.0.tar.xz
/gnu/store/7ak0v3rzpqm2c5q1mp3v7cj0rxz0qakf-libfabric-1.4.1.tar.bz2
/gnu/store/vh8syjrsilnbfcf582qhmvpg1v3rampf-rdma-core-14.tar.gz
...
```

(Wenn Sie es genauer wissen wollen, haben wir hier mehr als 320 MiB *komprimierten* Quellcodes.)

Daraus lässt sich ein gesammeltes Archiv für all den Quellcode basteln (siehe Abschnitt "Aufruf von guix archive" in *Referenzhandbuch zu GNU Guix*):

```
$ guix archive --export \
    `guix build --sources=transitive openmpi` \
    > openmpi-quellcode.nar
```

. . . und dieses Archiv können wir dann mittels Wechseldatenträgern auf den Cluster transportieren und dort entpacken:

```
$ guix archive --import < openmpi-quellcode.nar
```

Wiederholen Sie diesen Vorgang, wann immer Sie neuen Quellcode auf den Cluster schaffen müssen.

Wir schreiben das, aber zum jetzigen Zeitpunkt haben die Forschungsinstitute, die mit Guix-HPC arbeiten, gar keinen Cluster mit Air Gap. Wenn Sie Erfahrung damit haben, freuen wir uns über Rückmeldungen und Vorschläge.

9.4 Speicherplatz

Systemadministratoren sind oft besorgt, ob Guix nicht Unmengen an Plattenplatz verschlingen wird. Aber wenn überhaupt, werden wissenschaftliche Datensätze und nicht kompilierte Software die Speicherplatzfresser sein – unserer Erfahrung nach, da wir schon fast zehn Jahre lang Guix auf HPC-Clustern betreiben. Trotzdem lohnt sich ein Blick, wie Guix zum Speicherplatzverbrauch beiträgt.

Zunächst einmal kostet es *nicht* unbedingt viel, wenn mehrere Versionen oder Varianten eines bestimmten Pakets in `/gnu/store` liegen, weil `guix-daemon` gleiche Dateien dedupliziert und weil Paketvarianten in der Regel viele Dateien gemeinsam haben.

Dann ist es, wie oben beschrieben, unsere Empfehlung, mit einem „Cron-Job“ regelmäßig `guix gc` aufzurufen, was *nicht benutzte* Software aus `/gnu/store` löscht. Dennoch kann es sein, dass Benutzer eine ganze Menge Software in ihren Profilen übrig lassen oder dass sie viele alte Generationen ihrer Profile horten, die dann „lebendig“ sind und aus Sicht von `guix gc` *nicht* gelöscht werden dürfen.

Die Lösung besteht darin, dass Nutzer die alten Generationen ihrer Profile regelmäßig entfernen. Zum Beispiel werden mit folgendem Befehl alle Generationen entfernt, die älter als zwei Monate sind:

```
guix package --delete-generations=2m
```

Ebenso ist es eine gute Idee, den Nutzern nahezulegen, dass sie ihre Profile regelmäßig auf den neuesten Stand aktualisieren, weil dann weniger Varianten einer Software in `/gnu/store` bleiben müssen:

```
guix pull
guix upgrade
```

Notfalls können Systemadministratoren die Profile teilweise anstelle der Nutzer bereinigen. Das widerspricht allerdings einem Vorteil von Guix, dass Benutzer viel Freiheit und Kontrolle über ihre Software-Umgebungen bekommen. Wir empfehlen deutlich, die Kontrolle bei den Benutzern zu lassen.

9.5 Sicherheitsüberlegungen

Auf einem Hochleistungs-Cluster wird mit Guix normalerweise wissenschaftliche Software verwaltet. Über sicherheitskritische Software wie den Betriebssystem-Kernel und Systemdienste wie `sshd` und „Batch“-Auftragsplaner haben weiterhin die Systemadministratoren die Kontrolle.

Das Guix-Projekt kümmert sich zügig um Sicherheitsaktualisierungen (siehe Abschnitt “Sicherheitsaktualisierungen” in *Referenzhandbuch zu GNU Guix*). Um die Sicherheitsaktualisierungen zu bekommen, müssen Nutzer `guix pull && guix upgrade` ausführen.

Weil in Guix Software-Varianten eine eindeutige Bezeichnung haben, lässt es sich einfach überprüfen, ob jemand eine von Schwachstellen betroffene Software verwendet. Zum Beispiel kann man für die Variante von glibc 2.25, die den Patch zur Behebung der Sicherheitslücke namens „Stack Clash (`https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt`)“ noch nicht hat, prüfen, ob sie überhaupt in einem Benutzerprofil verwendet wird:

```
guix gc --referrers /gnu/store/...-glibc-2.25
```

Dadurch werden Profile gemeldet, die diese bestimmte glibc-Variante verwenden.

10 Danksagungen

Guix baut auf dem Nix-Paketverwaltungsprogramm (`https://nixos.org/nix/`) auf, das von Eelco Dolstra entworfen und entwickelt wurde, mit Beiträgen von anderen Leuten (siehe die Datei `nix/AUTHORS` in Guix). Nix hat für die funktionale Paketverwaltung die Pionierarbeit geleistet und noch nie dagewesene Funktionalitäten vorangetrieben wie transaktionsbasierte Paketaktualisierungen und die Rücksetzbarkeit selbiger, eigene Paketprofile für jeden Nutzer und referenziell transparente Erstellungsprozesse. Ohne diese Arbeit gäbe es Guix nicht.<

Die Nix-basierten Software-Distributionen Nixpkgs und NixOS waren auch eine Inspiration für Guix.

GNU Guix ist selbst das Produkt kollektiver Arbeit mit Beiträgen durch eine Vielzahl von Leuten. Siehe die Datei `AUTHORS` in Guix für mehr Informationen, wer diese wunderbaren Menschen sind. In der Datei `THANKS` finden Sie eine Liste der Leute, die uns geholfen haben, indem Sie Fehler gemeldet, sich um unsere Infrastruktur gekümmert, künstlerische Arbeit und schön gestaltete Themen beigesteuert, Vorschläge gemacht und noch vieles mehr getan haben – vielen Dank!

Dieses Dokument enthält angepasste Abschnitte aus Einträgen, die zuvor auf dem Blog von Guix unter `https://guix.gnu.org/blog` und dem Guix-HPC-Blog unter `https://hpc.guix.info/blog` veröffentlicht wurden.

Anhang A GNU-Lizenz für freie Dokumentation

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 `https://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.

Konzeptverzeichnis