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.

Alternatively you can also run `guix shell guile -- guile` if you'd rather not have Guile installed in your user profile.

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 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
  ```

- 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)
  ```

- Mit dem *quote*-Zeichen wird das Auswerten eines geklammerten Ausdrucks abgeschaltet: 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 "Hello " "Guix" "\n"))
  ⇒ (display (string-append "Hello " "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 Ergenis eine Liste verschiedenartiger Elemente ist: Zahlen, Symbole (in diesem Fall `a`) und als letztes Element selbst wieder eine Liste.

- 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`.

Going further: Scheme is a language that has been widely used to teach programming and you'll find plenty of material using it as a vehicle. Here's a selection of documents to learn more about Scheme:

- *A Scheme Primer* (`https://spritely.institute/static/papers/scheme-primer.html`), by Christine Lemmer-Webber and the Spritely Institute.
- *Scheme at a Glance* (`http://www.troubleshooters.com/codecorn/scheme_guile/hello.htm`), by Steve Litt.
- *Structure and Interpretation of Computer Programs* (`https://mitpress.mit.edu/sites/default/files/sicp/index.html`), by Harold Abelson and Gerald Jay Sussman, with Julie Sussman. Colloquially known as "SICP", this book is a reference.

  You can also install it and read it from your computer:

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

An unofficial ebook (`https://sarabander.github.io/sicp/`) is also available.

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 '`GUIX_PACKAGE_PATH`'

Anmerkung: Seit Guix 0.16 sind die vielseitigeren Kanäle von Guix die bevorzugte Wahl und sie lösen den '`GUIX_PACKAGE_PATH`' ab. Siehe den nächsten Abschnitt.

Es kann mühsam sein, die Datei auf der Befehlszeile anzugeben, statt einfach `guix package --install my-hello` aufzurufen, wie man es bei den offiziellen Paketen tun würde.

Guix ermöglicht es, den Prozess zu optimieren, indem man so viele „Paketdeklarationsverzeichnisse“, wie man will, hinzufügt.

Erzeugen Sie ein Verzeichnis, beispielsweise `~/guix-packages`, und fügen Sie es zur Umgebungsvariablen `GUIX_PACKAGE_PATH` hinzu:

```
$ mkdir ~/guix-packages
$ export GUIX_PACKAGE_PATH=~/guix-packages
```

Um mehrere Verzeichnisse hinzuzufügen, trennen Sie diese ab durch einen Doppelpunkt (`:`).

Unser '`my-hello`' von vorher braucht zudem ein paar 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.

'`my-hello`' sollte nun Teil der Paketsammlung sein, genau wie all die anderen, offiziellen Pakete. Sie können das so ausprobieren:

```
$ guix package --show=my-hello
```

2.1.2.3 Guix-Kanäle

Guix 0.16 hat Kanäle eingeführt, die sehr ähnlich zu '`GUIX_PACKAGE_PATH`' sind, sich aber besser integrieren und Provenienzverfolgung ermöglichen. Kanäle befinden sich nicht unbedingt auf einem lokalen Rechner, sie können zum Beispiel auch anderen als öffentliches Git-Repository angeboten werden. Natürlich können zur selben Zeit mehrere Kanäle benutzt werden.

Siehe Abschnitt “Kanäle” in *Referenzhandbuch zu GNU Guix* für Details zu deren Einrichtung.

2.1.2.4 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 ssh)
  #:use-module (gnu packages web)
  #:use-module (gnu packages pkg-config)
  #:use-module (gnu packages python)
  #:use-module (gnu packages compression)
  #:use-module (gnu packages tls))

(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)))
                ;; Remove bundled software.
                (snippet '(delete-file-recursively "deps"))))
      (build-system cmake-build-system)
      (outputs '("out" "debug"))
```

```
      (arguments
       `(#:tests? #true                          ; Run the test suite (this is the defa
         #:configure-flags '("-DUSE_SHA1DC=ON") ; SHA-1 collision detection
         #: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")))))
           ;; Run checks more verbosely.
           (replace 'check
             (lambda _ (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
       ;; These two libraries are in 'Requires.private' in libgit2.pc.
       (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.")
      ;; 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" "risc
                     #: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, desweiteren 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:

   - pinebook-pro-raw
   - pine64-raw
   - novena-raw
   - hurd-raw
   - hurd-qcow2
   - qcow2
   - uncompressed-iso9660
   - efi-raw
   - arm64-raw
   - arm32-raw
   - iso9660
```

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 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.

3.4.1 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.

3.4.2 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/`).

3.5 Fensterverwalter (Window Manager) anpassen

3.5.1 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.s
(xft:cache-fonts)
(set-font (make-instance 'xft:font :family "DejaVu Sans Mono" :subfamily "Book" :size
```

3.5.2 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.

3.5.2.1 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:

```
(screen-locker-service 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.

3.6 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 „Summary“.) 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
                     certs
                     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* nss-certs            ;für HTTPS-Zugriff
                 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.

3.7 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 quelllaufwerk ;; "quelllaufwerk" kann man nennen, wie man will
   (file-system
    (device (uuid "hier kommt die UUID hin"))
    (mount-point "/hier-der-pfad-zur-magnetfestplatte")
    (type "ext4"))) ;; Legen Sie den dazu passenden Typ fest.
```

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

```
(define (%quellverzeichnis) "/hier-der-pfad-zur-magnetfestplatte/tmp") ;; Dem "quellve
```

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

```
(file-systems (cons*

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

               quelllaufwerk ;; Muss dem Namen entsprechen, den Sie vorher ans Quelll

               (file-system
                (device (%quellverzeichnis)) ;; Geben Sie Acht, dass das "quellverzei
                (mount-point "/tmp")
                (type "none") ;; Wir binden ein Verzeichnis und keine Partition ein,
                (flags '(bind-mount))
                (dependencies (list quelllaufwerk)) ;; Geben Sie Acht, dass "quelllau
                )

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

               ))
```

3.8 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)
                      ;; Onion-Dienst von ci.guix.gnu.org
                      (substitute-urls
                       "https://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclc
                      (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
```

3.9 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")))))))
```

4 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.

4.1 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.

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. Desweiteren 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 4.1.5 [Reproduzierbare Profile], Seite 46.
- 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!

4.1.1 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 # wenn es noch nicht existiert
guix package --manifest=/pfad/zu/guix-my-project-manifest.scm --profile="$GUIX_EXTRA_P
```

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 /pfad/zu/guix-my-project-manifest.scm -p "$GUIX_EXTRA_PROFILES"/my-pro
```

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-ma
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
```

4.1.2 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.

4.1.3 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.

4.1.4 Der Vorteil von Manifesten

Manifeste 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.

Schauen wir uns ein Beispiel an:

1. Wir haben eine Umgebung, in der wir an einem Projekt hacken können, für das es noch kein Guix-Paket gibt. Wir richten die Umgebung mit einem Manifest ein und führen dann `guix environment -m manifest.scm` aus. So weit so gut.
2. Nach vielen Wochen haben wir in der Zwischenzeit schon ein paarmal `guix pull` laufen lassen. Vielleicht wurde eine Abhängigkeit aus unserem Manifest aktualisiert oder wir könnten `guix gc` ausgeführt haben, so dass manche Pakete, die von unserem Manifest gebraucht würden, vom Müllsammler geholt worden sind.
3. Eventually, we set to work on that project again, so we run `guix shell -m manifest.scm`. But now we have to wait for Guix to build and install stuff!

Ideal wäre es, wenn wir uns die Zeit für die Neuerstellung sparen könnten. Und das können wir auch: Alles, was wir brauchen, ist, das Manifest in ein Profil zu installieren

und `GUIX_PROFILE=/das/profil; . "$GUIX_PROFILE"/etc/profile` aufzurufen, 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.

4.1.5 Reproduzierbare Profile

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

- ein Manifest und
- eine Kanalspezifikation für 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. 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 "$GUIX_EXTRA"/my-project
guix pull --channels=channel-specs.scm --profile "$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-man
```

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.

5 Umgebungen verwalten

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

5.1 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)

# Dank an <https://lists.gnu.org/archive/html/guix-devel/2016-09/msg00859.html>
eval "$(guix environment --search-paths --root="$gcroot" --pure guix --ad-hoc ${PA

# configure-Optionen vordefinieren.
configure()
{
    ./configure --localstatedir=/var --prefix=
}
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.

6 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` 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