How to make advanced packages

Simon Tournier

Inserm US53 - UAR CNRS 2030
simon.tournier@inserm.fr

November, 10th, 2023

https://hpc.guix.info

Workshop on
Reproducible Software Environments
Montpellier

Université Paris Cité

Plan for today

A pedestrian journey toward G-expressions and Schemey-way

- The aim is to provide some “helpers”,
- For easing the reading of Guix manual and source code.

Preliminary I — how to connect using ssh on VM

1. Local file: ~/.ssh/config

```
Host bastion
     Hostname repro-guix.u-ga.fr
     User user_1   # or user_2 or user_3 or user_4
     StrictHostKeyChecking no
     IdentityFile ~/.ssh/stournier

Host repro-guix*
     User user_1  # same as previously
     ProxyCommand ssh bastion -W %h:22
```

Preliminary I bis — how to connect using ssh on VM

1. Create a SSH key for the session (here `~/.ssh/stournier`, pick your name)
   `ssh-keygen -t rsa -b 4096 -f ~/.ssh/stournier`
2. Copy the public SSH key
   `cat ~/.ssh/stournier.pub | ssh bastion 'cat >> .ssh/authorized_keys'`
3. Start your editor (VSCode or Emacs via Tramp or else)

Or just connect to one VM, then start Emacs in text mode:

```
guix shell emacs-minimal -- emacs -f shell
```

Preliminary II

1. Clone Git repository: `https://gitlab.com/zimoun/advanced-packages-2023`
2. Start a terminal

```
$ guix show -L examples/packages hi

$ guix build -L examples/packages hi
$ guix build -L examples/packages hi --no-grafts --check -K

$ guix edit -L $(pwd)/examples/packages hi
```

workaround VSCode: `EDITOR=./vscode-wrapper guix edit`

Plain text: `EDITOR=less guix edit`

Defining Packages: key points — file: examples/packages/first.scm

- **define-module** Create a Guile module
  - **#:use-module** List the modules required for Guile *compiling* the recipe
- **define-public** Define and export
  - **package** Object representing a package (Scheme record)
    - **name** The string we prefer
    - **version** A string that makes sense
    - **source** Define where to fetch the source code
    - **build-system** Define how to build
    - **arguments** The arguments for the build system
    - **inputs** List the other package dependencies

```
guix repl -L examples/packages
```

Package from guix repl

Recommendation for the file ~/.guile

```
(use-modules (ice-9 readline)       ;; package guile-readline, guile?
             (ice-9 format)
             (ice-9 pretty-print))
(activate-readline)
```

1. Type `hi` then `,q`
2. Type `(use-modules (first))` (or `,use(first)`) and again `hi`
3. Try `(package-name hi)` then `,use(guix packages)` (or `,use(guix)`) and repeat

Two names: the Scheme variable and the string.

1. How to display the version?
2. Try `(package-inputs hi)`

Examples of packages

```
$ guix edit gsl
$ guix edit r-torch
```

What does it mean?

keyword	`define-public`, `let`, `lambda`
record	`package`
convention	`%something`, `something?`, `something*`
symbol	quote (`'`), backtick (`` ` ``), comma (`,`), comma at (`,@`), underscore (`_`) G-expressions: `#~` or `#$`

First things first

S-expression: atom or expression of the form `(x y ...)`

atom: `+`, `*`, `list`, etc.

expression: `(list 'one 2 "three")`

First things first

S-expression: atom or expression of the form `(x y ...)`

atom: `+`, `*`, `list`, etc.

expression: `(list 'one 2 "three")`

Call (evaluation) with parenthesis

e.g., apply the atom `list` to the rest

`(list 'one 2 "three")` returns the list composed by the elements `(one 2 "three")`

First things first

S-expression: atom or expression of the form `(x y ...)`

atom: `+`, `*`, `list`, etc.

expression: `(list 'one 2 "three")`

Call (evaluation) with parenthesis

e.g., apply the atom `list` to the rest

`(list 'one 2 "three")` returns the list composed by the elements `(one 2 "three")`

Quote protects from the call (do not evaluate)

e.g., `'one` returns plain one

e.g., `'(list one 2 "three")` returns `(list 'one 2 "three")`

`'(list 'one 2 "three")` returns `(list (quote one) 2 "three")`

Second thing second

variable `(define some-variable 42)`
procedure `(lambda (argument) (something argument))`

Define a procedure

```
(define my-name-procedure
  (lambda (argument1 argument2)
    (something-with argument1)))
```

```
(define (my-name-procedure argument1 argument2)
  (something-with argument1))
```

Call `(my-name-procedure 1 "two")`

`define-public` is sugar to `define` and export (see « Creating Guile Modules (link)) »

Local variables = let

Independent local variables

```
(define (add-plus-2 x y)
  (let ((two 2)
        (x+y (+ x y)))
    (+ x+y two)))
```

Inter-dependant local variables

```
(define (add-plus-2-bis x y)
  (let* ((two 2)
         (x+two (+ x two))
         (result (+ y x+two)))
    result))
```

Local variables: example — seen in package julia-biogenerics

```
(define-public julia-biogenerics
  (let ((commit "a75abaf459250e2b5e22b4d9adf25fd36d2acab6")
        (revision "1"))
    (package
      (name "julia-biogenerics")
      (version (git-version "0.0.0" revision commit))
    ...
```

Conventions

predicate **ends with question mark (`?`)**, return boolean (`#t` or `#f`

note: `#true` or `#false` works too)

e.g., `(string-prefix?  "hello" "hello-world")`

Conventions

predicate **ends with question mark (?)**, return boolean (#t or #f

note: #true or #false works too)

e.g., `(string-prefix? "hello" "hello-world")`

variant **ends with star mark (*)** e.g., `let*`

`lambda*` more argument

```
(lambda* (#:key inputs #:allow-other-keys)
            (setenv "CONFIG_SHELL"
                    (search-input-file inputs "/bin/sh")))
;; seen in package frama-c
```

Conventions

predicate **ends with question mark (?)**, return boolean (#t or #f

note: #true or #false works too)

e.g., `(string-prefix? "hello" "hello-world")`

variant **ends with star mark (*)** e.g., `let*`

`lambda*` more argument

```
(lambda* (#:key inputs #:allow-other-keys)
         (setenv "CONFIG_SHELL"
                 (search-input-file inputs "/bin/sh")))
;; seen in package frama-c
```

keyword **starts with sharp colon (#:)**

e.g., `#:key`, `#:configure-flags`, `#:phases`

Conventions

predicate **ends with question mark (?)**, return boolean (#t or #f
note: #true or #false works too)
e.g., `(string-prefix? "hello" "hello-world")`

variant **ends with star mark (*)** e.g., `let*`
`lambda*` more argument

```
(lambda* (#:key inputs #:allow-other-keys)
         (setenv "CONFIG_SHELL"
                 (search-input-file inputs "/bin/sh")))
;; seen in package frama-c
```

keyword **starts with sharp colon (#:)**
e.g., `#:key`, `#:configure-flags`, `#:phases`

"global" **starts with percent (%)**
e.g., `%standard-phases`

Quote, quasiquote, unquote

quote	do not evaluate (keep as it is)	quote '
quasiquote	unevaluate except escaped	backtick `
unquote	evaluate that escaped	coma ,

```
guix repl
```

Quote, quasiquote, unquote

quote	do not evaluate (keep as it is)	quote ’
quasiquote	unevaluate except escaped	backtick `
unquote	evaluate that escaped	coma ,

guix repl

1. Type

```
scheme@(guix-user)> (define ho "path/to/ho")
scheme@(guix-user)> (string-append ho "/bin/bye")
scheme@(guix-user)> `(string-append ho "/bin/bye")
scheme@(guix-user)> `(string-append ,ho "/bin/bye")
```

Quote, quasiquote, unquote

quote	do not evaluate (keep as it is)	quote ’
quasiquote	unevaluate except escaped	backtick `
unquote	evaluate that escaped	coma ,

guix repl

1. Type

```
scheme@(guix-user)> (define ho "path/to/ho")
scheme@(guix-user)> (string-append ho "/bin/bye")
scheme@(guix-user)> `(string-append ho "/bin/bye")
scheme@(guix-user)> `(string-append ,ho "/bin/bye")
```

2. Type

```
scheme@(guix-user)> (eval $4 (interaction-environment))
```

Quote, quasiquote, unquote

quote	do not evaluate (keep as it is)	quote ’
quasiquote	unevaluate except escaped	backtick `
unquote	evaluate that escaped	coma ,

guix repl

1. Type

```
scheme@(guix-user)> (define ho "path/to/ho")
scheme@(guix-user)> (string-append ho "/bin/bye")
scheme@(guix-user)> `(string-append ho "/bin/bye")
scheme@(guix-user)> `(string-append ,ho "/bin/bye")
```

2. Type

```
scheme@(guix-user)> (eval $4 (interaction-environment))
```

construction-time vs **eval-time**

Quote, quasiquote, unquote II

splicing

unquote-splicing as unquote and insert the elements

comma-at `,@`

the expression must evaluate to a list

1. Type

```
scheme@(guix-user)> (define of (list #:vegetable 'tomatoes
                                     #:dessert (list "cake" "pie")))
scheme@(guix-user)> `(more ,@of that)
scheme@(guix-user)> `(more ,of that)
```

Quote, quasiquote, unquote II bis — splicing

```
(arguments
  `(,@(package-arguments gsl)
     #:configure-flags (list "--disable-shared")
     #:make-flags (list "CFLAGS=-fPIC")))
;; seen in package gsl-static
```

1. Type

```
scheme@(guix-user)> ,use(gnu packages maths)
scheme@(guix-user)> ,pp (package-arguments gsl)
scheme@(guix-user)> ,pp `(,@(package-arguments gsl)
                           #:configure-flags (list "--disable-shared")
                           #:make-flags (list "CFLAGS=-fPIC"))
```

Quote, quasiquote, unquote III

digression

`substitute-keyword-arguments` substitutes keyword arguments

```
(arguments
 (substitute-keyword-arguments (package-arguments hdf4)
   ((#:configure-flags flags) `(cons* "--disable-netcdf" ,flags))))
;; seen in package hdf4-alt
```

Quote, quasiquote, unquote III

digression

substitute-keyword-arguments substitutes keyword arguments

```
(arguments
 (substitute-keyword-arguments (package-arguments hdf4)
   ((#:configure-flags flags) `(cons* "--disable-netcdf" ,flags))))
;; seen in package hdf4-alt
```

1.
```
scheme@(guix-user)> ,use(srfi srfi-1)
scheme@(guix-user)> ,pp (lset-difference equal?
          (substitute-keyword-arguments (package-arguments hdf4)
            ((#:configure-flags flags) `(cons* "--disable-netcdf" ,flags)))
          (package-arguments hdf4))
$1 = ((cons* "--disable-netcdf" (list "--enable-shared" "FCFLAGS=-fallow-ar
                                      "FFLAGS=-fallow-argument-mismatch"
                                      "--enable-hdf4-xdr")))
```

Association list

(*alist*) association list = list of pairs `(this . that)`

think: `(list (key1 . value1) (key2 . value2) ...)`

1. Type

```
scheme@(guix-user)> (define alst (list '(a . 1) '(2 . 3) '("foo" . v)))
scheme@(guix-user)> (assoc-ref alst "foo")
scheme@(guix-user)> (assoc-ref alst 'a)
```

2. Type

```
scheme@(guix-user)> (assoc-ref (package-inputs hi) "gawk")
```

Ready? seen in package feedgnuplot

```
1  (add-after 'install 'wrap
2    (lambda* (#:key inputs outputs #:allow-other-keys)
3      (let* ((out (assoc-ref outputs "out"))
4             (gnuplot (search-input-file inputs "/bin/gnuplot"))
5             (modules '("perl-list-moreutils" "perl-exporter-tiny"))
6             (PERL5LIB (string-join
7                        (map (lambda (input)
8                               (string-append (assoc-ref inputs input)
9                                              "/lib/perl5/site_perl"))
10                            modules)
11                       ":")))
12       (wrap-program (string-append out "/bin/feedgnuplot")
13         `("PERL5LIB" ":" suffix (,PERL5LIB))
14         `("PATH" ":" suffix (,(dirname gnuplot)))))))
```

Let build something!

Before, we need to fetch something, isn't it?

origin field seen in package feedgnuplot

```
(source (origin
    (method git-fetch)
    (uri (git-reference
          (url home-page)
          (commit (string-append "v" version))))
    (file-name (git-file-name name version))
    (sha256
     (base32
      "0403hwlian2s431m36qdzcczhvfjvh7128m64hmmwbbrgh0n7md7"))))
```

Defining origin

origin Object representing a source code (data) (Scheme record)
- method How to fetch
- uri Where to fetch
- sha256 Checksum, see `guix hash`

1. Remember the package `hi`

```
scheme@(guix-user)> (package-source hi)
```

2. Compare `feedgnuplot` (from module `(gnu packages maths)`)

```
scheme@(guix-user)> (package-source feedgnuplot)
```

3. Another terminal, compare

```
$ guix build -L examples/packages hi --source
$ guix build -L examples/packages feedgnuplot --source
```

Defining `origin` II

more `method`

fixed-output derivation = content known in advance

- `url-fetch` fetches data from URL (= a string or a list of strings)

  module `(guix download)`

see `origin` Reference (link) in Guix manual

Defining `origin` II

more `method`

fixed-output derivation = content known in advance

- `url-fetch` fetches data from URL (= a string or a list of strings)
  module `(guix download)`
- `git-fetch` fetches a `git-reference` object
  module `(guix git-download)`

see `origin` Reference (link) in Guix manual

Defining `origin` II

more `method`

fixed-output derivation = content known in advance

- `url-fetch` fetches data from URL (= a string or a list of strings) module `(guix download)`
- `git-fetch` fetches a `git-reference` object module `(guix git-download)`
- `hg-fetch` fetches a `hg-reference` object module `(guix hg-download)`
- `svn-fetch` fetches a `svn-reference` object module `(guix svn-download)`
- etc.

see `origin` Reference (link) in Guix manual

Modifying `origin`

1. Fetch the source code of the package gecode

```
$ guix build gecode --source
```

Modifying origin

1. Fetch the source code of the package gecode

   ```
   $ guix build gecode --source
   ```

2. Show what git-fetch does behind the scene

   ```
   $ ./examples/scripts/show-me-fetch.scm gecode
   $ eval $(./examples/scripts/show-me-fetch.scm gecode)
   ```

Modifying origin

1. Fetch the source code of the package gecode

   ```
   $ guix build gecode --source
   ```

2. Show what git-fetch does behind the scene

   ```
   $ ./examples/scripts/show-me-fetch.scm gecode
   $ eval $(./examples/scripts/show-me-fetch.scm gecode)
   ```

3. Compare both

   ```
   $ diff -rq /tmp/gecode $(guix build gecode -S)
   ```

Modifying origin

1. Fetch the source code of the package gecode

   ```
   $ guix build gecode --source
   ```

2. Show what git-fetch does behind the scene

   ```
   $ ./examples/scripts/show-me-fetch.scm gecode
   $ eval $(./examples/scripts/show-me-fetch.scm gecode)
   ```

3. Compare both

   ```
   $ diff -rq /tmp/gecode $(guix build gecode -S)
   ```

how to remove these files?

Modifying origin II

1. Compare both

```
$ diff -rq /tmp/gecode $(guix build gecode -S)
```

2. Another comparison

```
$ guix hash -S nar -f base32-nix -H sha256 -x $(guix build -S gecode)
$ guix hash -S nar -f base32-nix -H sha256 -x /tmp/gecode

$ rm /tmp/gecode/gecode/kernel/var-{imp,type}.hpp
$ guix hash -S nar -f nix-base32 -H sha256 -x /tmp/gecode
```

how to remove these files?

Modifying origin III

snippet

A S-expression (or G-expression) that will be run in the source directory

```
(origin
...
  (snippet
    '(begin
       ;; delete generated sources
       (for-each delete-file
                 '("gecode/kernel/var-imp.hpp"
                   "gecode/kernel/var-type.hpp"))))
```

patches vs snippet: it depends on

Look at the module `(guix builds utils)` for helpers as `delete-file-recursively`, etc.

Pass arguments to the build system

```
(arguments
 (list #:configure-flags
       #~(list "--enable-dynamic-build"
               #$@(if (target-x86?)
                      #~("--enable-vector-intrinsics=sse")
                      #~())
               "--enable-ldim-alignment")
       #:make-flags #~(list "FC=gfortran -fPIC")
       #:phases
       #~(modify-phases %standard-phases
```

`#:configure-flags` is keyword.
What is `#~` or `#$@`?

G-expression

Remember quasiquote and unquote?

`#~`	is similar as	`` ` ``	with context (host machine, store state, etc.)
`#$`	is similar as	`,`	with context
`#$@`	is similar as	`,@`	with context

```
#~(string-append #$hello "/some/string")
                    "means"
"/gnu/store/8bzzc70vgzdvj6qdzhdpd709m4y2kw7z-hello-2.12.1/some/string"
```

https://simon.tournier.info/posts/2023-11-02-gexp-intuition.html

G-expression II

```
(replace 'install
  (lambda* (#:key outputs #:allow-other-keys)
    (mkdir-p (string-append #$output "/bin"))
    (chmod "BQN" #o755)
    (rename-file "BQN" "bqn")
    (install-file "bqn" (string-append #$output "/bin"))))
```

Phases %standard-phases

%standard-phases is defined build system by build system

```
$ ./examples/scripts/compare-bs-phases.scm gnu python
```

Phases %standard-phases

%standard-phases is defined build system by build system

```
$ ./examples/scripts/compare-bs-phases.scm gnu python
```

Phases

1. 'configure
2. 'build
3. 'install
4. 'check
5. etc.

Phases %standard-phases

%standard-phases is defined build system by build system

```
$ ./examples/scripts/compare-bs-phases.scm gnu python
```

Phases

1. 'configure
2. 'build
3. 'install
4. 'check
5. etc.

Actions

1. replace
2. delete
3. add-before
4. add-after

Phases %standard-phases II

```
(arguments
 (list
  #:phases
  #~(modify-phases %standard-phases
      (delete 'configure)
      (add-before 'build 'set-prefix-in-makefile
        (lambda* (#:key inputs #:allow-other-keys)
          (substitute* "Makefile"
            (("PREFIX =.*")
             (string-append "PREFIX = " #$output "\n"))
            (("XMLLINT =.*")
             (string-append "XMLLINT = "
                            (search-input-file inputs "/bin/xmllint")
                            "\n")))))))))
```

Phases %standard-phases III

```
(arguments
  (if (not (target-x86-64?))
      ;; This test is only broken when using openblas, not openblas-ilp64.
      (list
        #:phases
        #~(modify-phases %standard-phases
            (add-after 'unpack 'adjust-tests
              (lambda _
                (substitute* "test/test_layoutarray.jl"
                  (("test all\\(B") "test_broken all(B"))))))
      '()))
;; see in julia-arraylayouts
```

Resources (links)

Talk « A tour of the Guix source tree » (video 40min)

Talk « Introduction to G-Expressions » (video 30min)

---

self-promotion

`https://simon.tournier.info/posts/`

Post « Automatic differentiation by dual numbers using Guile »

Post « From naive to rough intuition about G-expression »

Post « Quasiquote and G-expression: Fibonacci sequence using derivations »

Packaging = practise and practise again

If I might,

1. Dive into existing packages and deal with Guix manual and community.
2. Most of the "tricks" is about **a lot of** practise. Quoting `rekado`,

I wish I had anything to say about this other than:
"try again, give up, forget about it, remember it, ask for pointers, repeat"
`#guix-hpc` on 2023-10-13.

do not forget that **packaging is a craft**

Hope to discuss your patches soon :-)

zimoun on #guix or #guix-hpc libera.chat

mattermost.univ-nantes.fr

guix-science@gnu.org
guix-devel@gnu.org
guix-patches@gnu.org

CaféGuix

https://hpc.guix.info

```
$ ./etc/teams.scm list-members mentors
( <paren@disroot.org>
Christopher Baines <guix@cbaines.net>
Ludovic Courtès <ludo@gnu.org>
Mathieu Othacehe <othacehe@gnu.org>
Raghav Gururajan <rg@raghavgururajan.name>
Ricardo Wurmus <rekado@elephly.net>
Simon Tournier <zimon.toutoune@gmail.com>
Tobias Geerinckx-Rice <me@tobias.gr>
jgart <jgart@dismail.de>
```