Introduction to G-expression

(how to beat around the bush)

Simon Tournier

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

April, 30th, 2024

https://hpc.guix.info

CaféGuix

Université Paris Cité

Too long; don't read

Guix is implemented in Scheme/Guile language.

Guix provides a Domain-Specific Language (DSL),

This DSL helps in defining `packages`.

"G-expression" is another embedded DSL and adapted to build *thing*.

This DSL helps in describing a sequence of actions;
to be performed to produce item in the store.

Too short; more read

- Code Staging in GNU Guix, *in GPCE'17, 16th ACM SIGPLAN International Conference*
  https://doi.org/10.48550/arXiv.1709.00833

- Guix manual: read twice; 1, 2, 3 and 4 ("anti-order") then 4, 3, 2 and 1 ("natural" order)
  1. section G-Expressions
     https://guix.gnu.org/manual/devel/en/guix.html#G_002dExpressions
  2. section The Store Monad
     https://guix.gnu.org/manual/devel/en/guix.html#The-Store-Monad
  3. section Derivations
     https://guix.gnu.org/manual/devel/en/guix.html#Derivations
  4. section The Store
     https://guix.gnu.org/manual/devel/en/guix.html#The-Store

- source code
  - other packages
  - the build system you know the most — e.g., `(guix build-system julia)`
  - the hard way: module `(guix gexp)`

From my experience

the main difficulties when speaking about *G-expression* are from:

1. Missing knowledge about what is Guile-specific language (`lambda`, `let`, conventions, etc.)
2. Missing knowledge about Scheme concepts ("*quotation*")

here the aim is thus to introduce some

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.

Defining Packages: key points

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

all sounds clear...

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 `#$`

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 `r-torch` then `,q`
2. Type `(use-modules (gnu packages cran))` (or `,use(gnu packages maths)`) and again `r-torch`
3. Try `(package-name r-torch)` then `,use(guix packages)` (or `,use(guix)`) and repeat

Two names: the Scheme variable and the string.

Package from `guix repl` II

1. How to display the version?
2. Try `(package-inputs r-torch)`
3. About the arguments?

```
scheme@(guix-user)> ,pp (package-arguments r-torch)
$3 = (#:phases
 #<gexp (modify-phases %standard-phases (add-after (quote install) (quote link-
(lambda* (#:key inputs #:allow-other-keys) (let ((deps (string-append #<gexp-ou
(string-append deps "/liblantern.so")))))) gnu/packages/cran.scm:30475:8 769ee1

scheme@(guix-user)> ,pp (package-arguments gsl)
$4 = (#:configure-flags
 (list "--disable-static")
 #:phases
 (modify-phases %standard-phases))
```

First things first

'S' is before 'G'

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

S-exp: opening-parenthesis something ... closing-parentheis

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

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

First things first — 'S' is before 'G'

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

S-exp: opening-parenthesis something ... closing-parentheis

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' is before 'G'

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

S-exp: opening-parenthesis something ... closing-parentheis

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`

warning: keyword starting with `#:` is not a convention

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))
```

read-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 r-torch) "python-pytor
```

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)))))))
```

We want G-expression!

Enough of S-expression.

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
```

#: introduces keyword.
What is #~ or #$@?

G-expression

Remember quasiquote and unquote?

`#~`	is similar as	`` ` ``	with context	(unevaluate except escaped)
`#$`	is similar as	`,`	with context	(evaluate that escaped)
`#$@`	is similar as	`,@`	with context	(evaluate and insert)

where context means system of host machine, store state, etc.

Intuition

```
scheme@(guix-user)> ,use(gnu packages maths)
scheme@(guix-user)> (define gsl-name (package-name gsl))
scheme@(guix-user)> `(begin (string-append ,gsl-name "/yet/another"))
```

```
scheme@(guix-user)> `(begin (string-append ,gsl "/yet/another"))
```

Intuition II

```
scheme@(guix-user)> `(begin (string-append ,gsl-name "/yet/another"))
$1 = (begin (string-append "gsl" "/yet/another"))
```

```
scheme@(guix-user)> `(begin (string-append ,gsl "/yet/another"))
$2 = (begin (string-append #<package gsl@2.7.1 gnu/packages/maths.scm:679 77da2
```

Intuition III

```
scheme@(guix-user)> `(begin (string-append ,gsl-name "/yet/another"))
$1 = (begin (string-append "gsl" "/yet/another"))

scheme@(guix-user)> (eval $1 (interaction-environment))
$3 = "gsl/yet/another"
```

```
scheme@(guix-user)> `(begin (string-append ,gsl "/yet/another"))
$2 = (begin (string-append #<package gsl@2.7.1 gnu/packages/maths.scm:679 77da2
```

What is the result of

```
scheme@(guix-user)> (eval $2 (interaction-environment))
```

?

Intuition IV

```
scheme@(guix-user)> `(begin (string-append ,gsl "/yet/another"))
$2 = (begin (string-append #<package gsl@2.7.1 gnu/packages/maths.scm:679 77da2
```

```
scheme@(guix-user)> (eval $2 (interaction-environment))

ice-9/boot-9.scm:1685:16: In procedure raise-exception:
In procedure string-append: Wrong type (expecting string): #<package gsl@2.7.1

Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guix-user) [1]>
```

Intuition IV

```
scheme@(guix-user)> `(begin (string-append ,gsl "/yet/another"))
$2 = (begin (string-append #<package gsl@2.7.1 gnu/packages/maths.scm:679 77da2
```

```
scheme@(guix-user)> (eval $2 (interaction-environment))

ice-9/boot-9.scm:1685:16: In procedure raise-exception:
In procedure string-append: Wrong type (expecting string): #<package gsl@2.7.1

Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guix-user) [1]>
```

it is an error!

Because `gsl` is not a string.

Intuition V

```
scheme@(guix-user)> `(begin (string-append ,gsl "/yet/another"))
```

`	replaced by	#~
,	replaced by	#$

```
scheme@(guix-user)> #~(begin (string-append #$gsl "/yet/another"))
$4 = #<gexp (begin (string-append #<gexp-input #<package gsl@2.7.1 gnu/packages
```

Intuition VI

```
scheme@(guix-user)> #~(begin (string-append #$gsl "/yet/another"))
$4 = #<gexp (begin (string-append #<gexp-input #<package gsl@2.7.1 gnu/packages
```

```
scheme@(guix-user)> (define gexp->sexp (@@ (guix gexp) gexp->sexp))
scheme@(guix-user)> (gexp->sexp $4 "x86_64-linux" #f)
$5 = #<procedure 77da24f88e40 at guix/gexp.scm:1408:2 (state)>
```

```
scheme@(guix-user)> ,run-in-store (gexp->sexp $4 "x86_64-linux" #f)
$6 = (begin
  (string-append
    "/gnu/store/dzx94b3xv4h1ik1bfrbxaw7n84y9r8zz-gsl-2.7.1"
    "/yet/another"))
```

Example

```
#:phases
#~(modify-phases %standard-phases
[...]
    (replace 'install
      (lambda _
        (mkdir-p (string-append #$output "/bin"))
        (chmod "BQN" #o755)
        (rename-file "BQN" "bqn")
        (install-file "bqn" (string-append #$output "/bin"))))
```

package `cbqn-bootstrap`

Example II

```
#:phases
#~(modify-phases %standard-phases
    (delete 'configure)
    (add-before 'build 'generate-bytecode
      (lambda _
        (system (string-append #+dbqn
                               "/bin/dbqn ./genRuntime "
                               #+bqn-sources))))
[...]
(native-inputs (list dbqn bqn-sources))
(inputs (list icedtea-8 libffi))
```

#+ plays the same role as #$, but is a reference to a native package build (cross-compilation context)

Ready?

```
scheme@(guix-user)> `(begin (string-append ,gsl-name "/yet/another"))

scheme@(guix-user)> `(begin ,(string-append gsl-name "/yet/another"))

scheme@(guix-user)> #~(begin #$(string-append gsl "/yet/another"))
```

What happens?

see `file-append`

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 « Derivative, dual numbers and Guile »

Post « Guix: an intuition about G-expression »

Post « Guix: Quasiquote and G-expression »
illustration of reference tracking by G-exp using Fibonacci numbers

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**