Code Staging in GNU Guix

Ludovic Courtès

GPCE
23 October 2017, Vancouver, Canada

Inria informatics mathematics

Functional software deployment.

```
$ guix package -i emacs ocaml idris ghc
...

$ guix package -r emacs -i vim
...
```

```
$ guix package -i emacs ocaml idris ghc
...

$ guix package -r emacs -i vim
...

$ guix package --roll-back
...
```

```
(operating-system
  (host-name "gastown")
  (timezone "Canada/Pacific")
  (locale "fr_CA.utf8")
  (bootloader (grub-configuration (device "/dev/sda")))
  (file-systems (cons (file-system
                        (device "my-root")
                        (title 'label)
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))
  (users (cons (user-account
                (name "alice")
                (group "users")
                (home-directory "/home/alice"))
               %base-user-accounts))
  (services (cons (service openssh-service-type)
                  %base-services)))
```

```
$ guix system vm config.scm
…

$ guix system reconfigure config.scm
…

$ guix system roll-back
…
```

Functional software deployment paradigm:

1. build process = **pure function**
2. built software = **persistent graph**

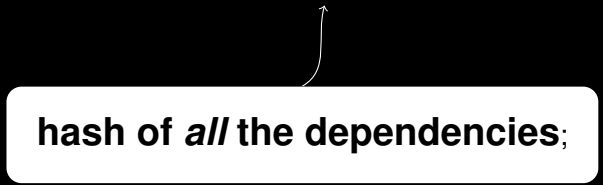
Imposing a Memory Management Discipline on Software Deployment, Dolstra et al., 2004
(Nix package manager)

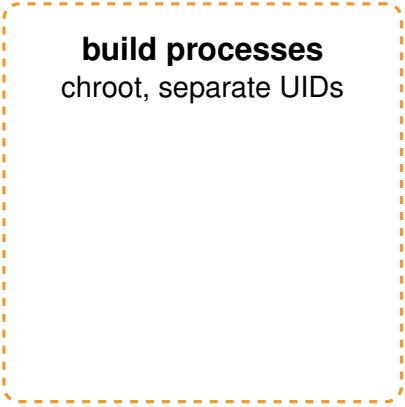
```
(define hello
  (package
    (name "hello")
    (version "2.10")
    (source (origin
              (method url-fetch)
              (uri (string-append "https://ftp.gnu.org/.../hello-"
                                  version ".tar.gz"))
              (sha256 (base32 "0wqd...dz6"))))
    (build-system gnu-build-system)
    (synopsis "An example GNU package")
    (description "Produce a friendly greeting.")
    (home-page "https://gnu.org/software/hello/")
    (license gpl3+)))
```

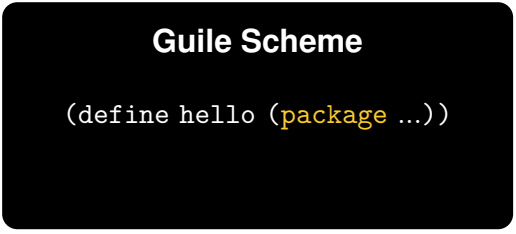
Functional Package Management with Guix, Courtès, 2013

```
$ guix build hello
```

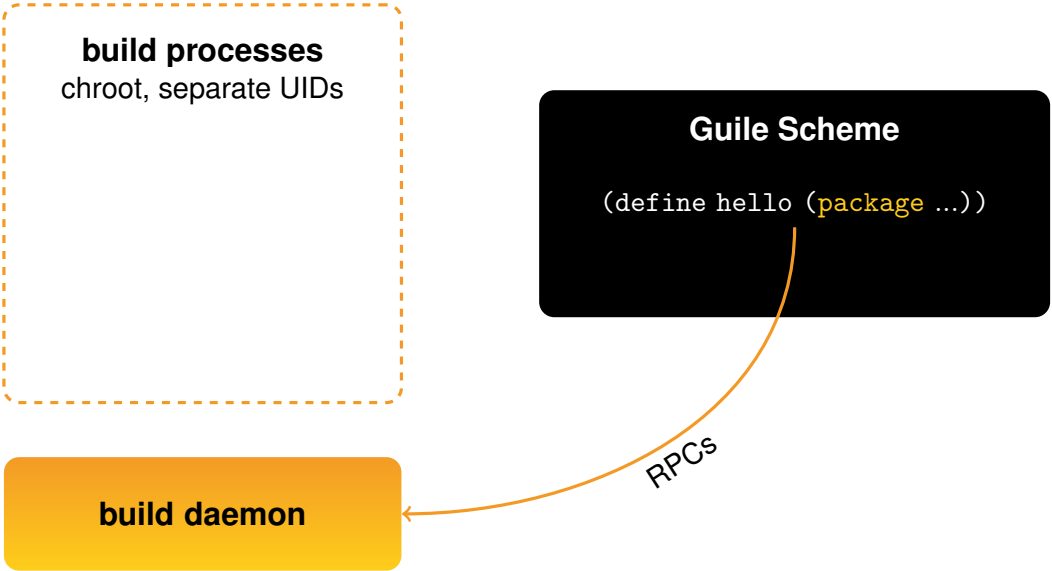
```
$ guix build hello
/gnu/store/ sp4wxa09zxfcs4vchm6as014msa5yi22 -hello-2.10
```

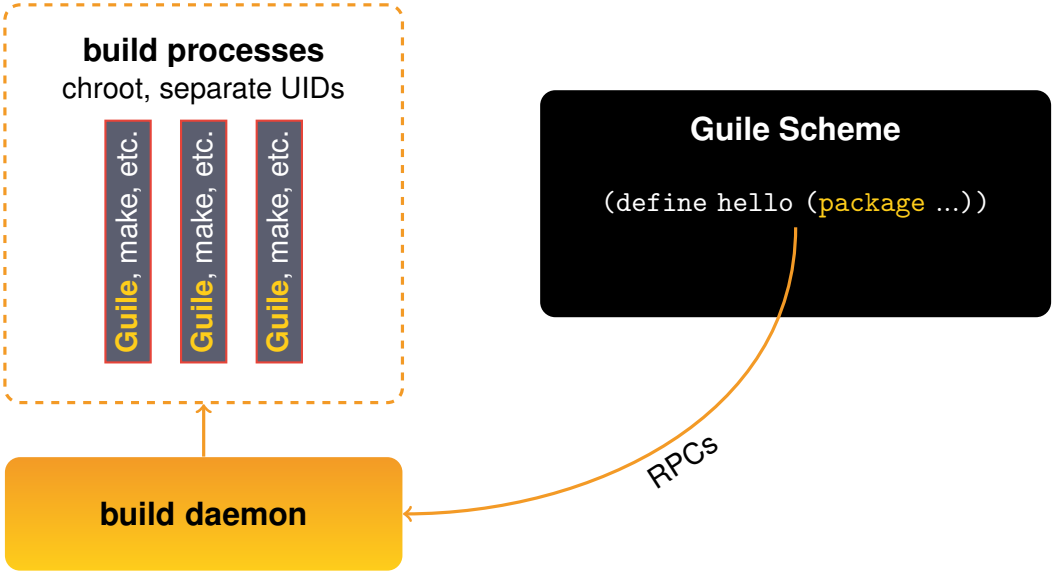



build processes
chroot, separate UIDs


Guile Scheme
(define hello (package ...))


build daemon


build processes
chroot, separate UIDs
Guile Scheme
(define hello (package ...))
RPCs
build daemon


build processes
chroot, separate UIDs
Guile, make, etc.
Guile, make, etc.
Guile, make, etc.
Guile Scheme
(define hello (package ...))
RPCs
build daemon

Code staging.

Staging: take #1

```
(define build-exp
  ;; Build-side code.
  '(symlink "/gnu/store/123...-coreutils-8.25"
            "/gnu/store/abc...-result"))
```

Staging: take #1

```
(define build-exp
  ;; Build-side code.
  '(symlink (assoc-ref %build-inputs "coreutils")
            %output))

;; ... with unhygienic global variable:
;; (define %build-inputs
;;   '(("coreutils" . "/gnu/store/...-coreutils-8.25")))

(define inputs
  ;; What goes into the chroot.
  `(("coreutils" ,coreutils)))

(build-expression->derivation store "symlink-to-coreutils"
                              build-exp
                              #:inputs inputs)
```

Staging: take #1

```
(define build-exp
  ;; Build-side code.
  '(symlink (assoc-ref %build-inputs "coreutils")
            %output))

;; ... with unhygienic global variable:
;; (define %build-inputs
;;   '())
```

```
(build-expression->derivation store "symlink-to-coreutils"
                              build-exp
                                        )
```

- Nix: generates Bash code through string interpolation
- *Writing Hygienic Macros in Scheme with Syntax-Case*, (Dybvig, 1992)
- *A Multi-Tier Semantics for Hop* (Serrano & Queinnec, 2010)

Take #2: G-expressions

```
(define build-exp
  ;; First-class object that carries info
  ;; about its dependencies.
  (gexp (symlink (ungexp coreutils)
                 (ungexp output))))

;; Leads to a build script like:
;; (symlink "/gnu/store/123...-coreutils-8.25"
;;          (getenv "out"))

(gexp->derivation "symlink-to-coreutils" build-exp)
```

Take #2: G-expressions

```
(define build-exp
  ;; First-class object that carries info
  ;; about its dependencies.
  #~(symlink #$coreutils #$output))

;; Leads to a build script like:
;; (symlink "/gnu/store/123...-coreutils-8.25"
;;          (getenv "out"))

(gexp->derivation "symlink-to-coreutils" build-exp)
```

Take #2: G-expressions

```
(define build-exp
  ;; First-class object that carries info
  ;; about its dependencies.
  #~(symlink #$coreutils #$output))

;; Leads to a build script like:
;; (symlink "/gnu/store/h8a...-coreutils-8.25"
;;          (getenv "out"))

(gexp->derivation "symlink-to-coreutils" build-exp
                  #:system "i686-linux")
```

Modules

```
(define build-exp


    #~(begin
        (use-modules (guix build utils))
        (mkdir-p (string-append #$output "/bin")))

(gexp->derivation "empty-bin-dir" build-exp)
;; ERROR: (guix build utils) not found!
```

Modules

```
(define build-exp
  ;; Compile (guix build utils) and add it
  ;; to the chroot.
  (with-imported-modules '((guix build utils))
    #~(begin
        (use-modules (guix build utils))
        (mkdir-p (string-append #$output "/bin")))

(gexp->derivation "empty-bin-dir" build-exp)
```

Initial RAM Disk

```
(expression->initrd
 (with-imported-modules (source-module-closure
                         '((gnu build linux-boot)
                           (guix build utils)))
   #~(begin
       (use-modules (gnu build linux-boot)
                    (guix build utils))

       (boot-system #:mounts '#$file-systems
                    #:linux-modules '#$linux-modules
                    #:linux-module-directory '#$kodir)))
```

Towards More Use Cases...

```
(eval-remotely #~(system* #$(file-append ffmpeg "/bin/ffmpeg") ...)
               (open-ssh-session "example.org"))

(eval-in-vm #~(uname) vm)

...
```

Scope Preservation

```
(let ((gen-body (lambda (x)
                  #~(let ((x 40))
                      (+ x #$x)))))
  #~(let ((x 2))
      #$(gen-body #~x))
```

Scope Preservation

```
(let ((gen-body (lambda (x)
                  #~(let ((x 40))
                      (+ x #$x)))))
  #~(let ((x 2))
      #$(gen-body #~x))

⇝ (let ((x-1bd8-0 2))
    (let ((x-4f05-0 40))
      (+ x-4f05-0 x-1bd8-0)))
```

Dybvig 1992, Kiselyov 2008, Rhiger 2012

Limitations

- hygiene vs. **macro-introduced bindings**?
- **modules** in scope?
- **serialization** of non-primitive data types?
- cross-stage **debugging info** à la Hop?

Summary.

- G-exps **tie staging to software deployment**
- **used at scale in Guix**
- entirely **implemented as a macro**

ludo@gnu.org

https://gnu.org/software/guix/

Ghost track!

Cross-Compilation

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

;; Yields:
;; (begin
;;   (mkdir (getenv "out"))
;;   (system* (string-append "/gnu/store/123..." "/bin/ln")
;;            "-s"
;;            (string-append "/gnu/store/345..." ...)
;;            (string-append "/gnu/store/567..." ...)))
```

Cross-Compilation

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

;; Yields:
;; (begin
;;   (mkdir (getenv "out"))
;;   (system* (string-append "/gnu/store/123..." "/bin/ln")
;;            "-s"
;;            (string-append "/gnu/store/9ab..." ...)
;;            (string-append "/gnu/store/fc2..." ...)))
```

Defining “Compilers”

```
(define-gexp-compiler (package-compiler (package <package>)
                                        system target)
  ;; Return a derivation to build PACKAGE.
  (if target
      (package->cross-derivation package target system)
      (package->derivation package system)))
```

Defining “Compilers”

```
(define-gexp-compiler (package-compiler (package <package>)
                                        system target)
  ;; Return a derivation to build PACKAGE.
  (if target
      (package->cross-derivation package target system)
      (package->derivation package system)))


(define-record-type <plain-file>
  (plain-file name content)
  ...)

(define-gexp-compiler (plain-file-compiler (file <plain-file>)
                                           system target)
  ;; "Compile" FILE by adding it to the store.
  (match file
    (($ <plain-file> name content)
```

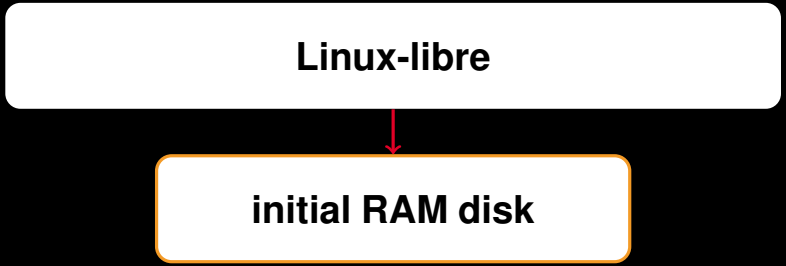
Compilers & “Expanders”

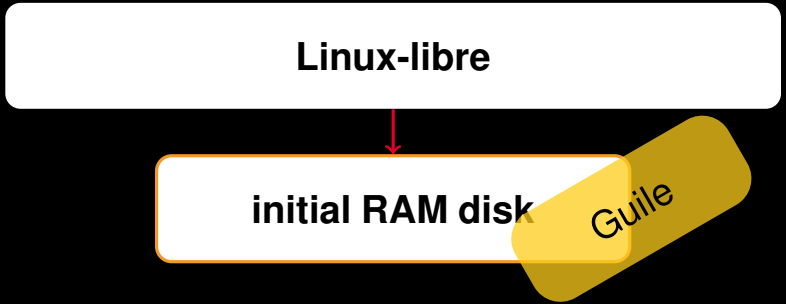
```
#~(string-append #$coreutils "/bin/ls")

;; Yields:
;; (string-append "/gnu/store/...-" "/bin/ls")
```

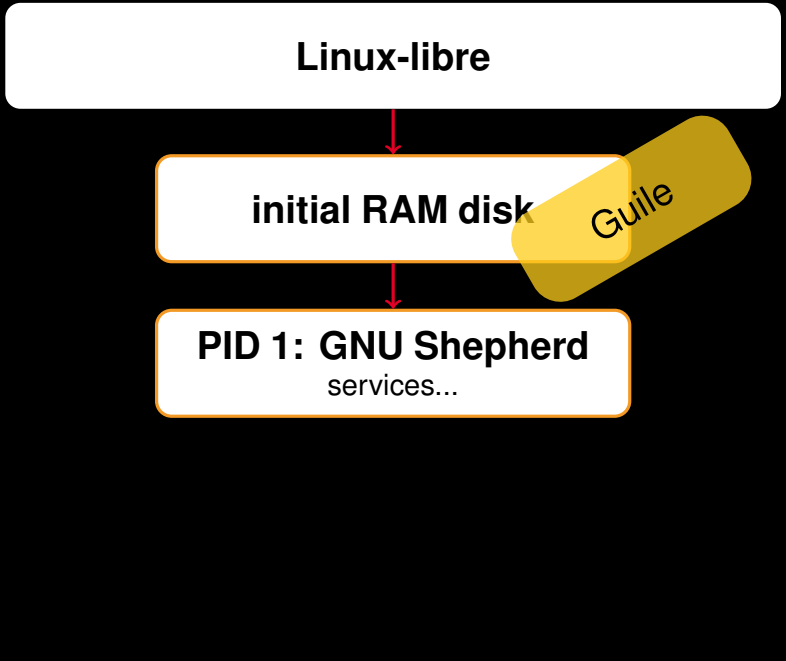
Compilers & “Expanders”

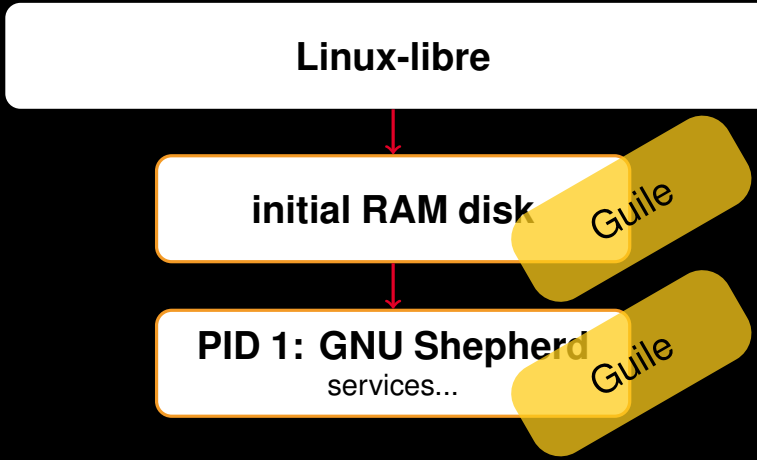
```
#~(string-append #$coreutils "/bin/ls")

;; Yields:
;; (string-append "/gnu/store/…" "/bin/ls")

(file-append coreutils "/bin/ls")

;; Yields:
;; "/gnu/store/…/bin/ls"
```

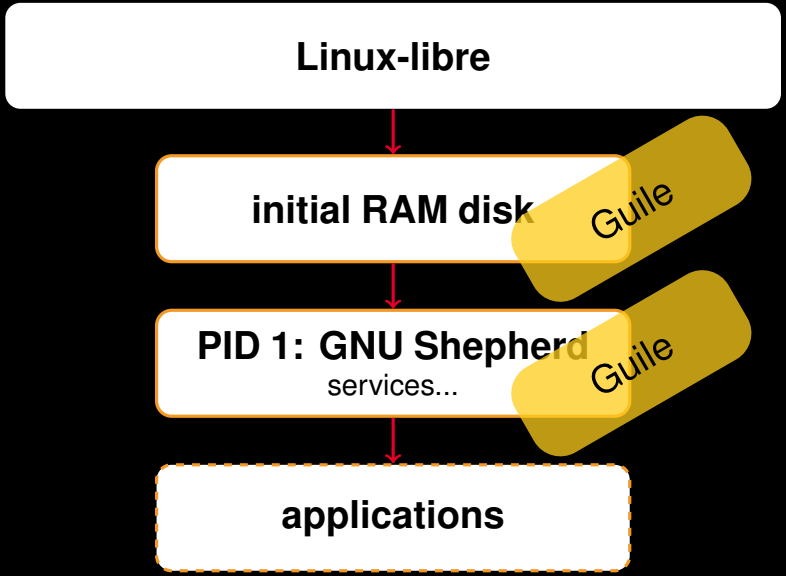
Linux-libre


Linux-libre
initial RAM disk


Linux-libre
initial RAM disk
Guile


Linux-libre
initial RAM disk
Guile
PID 1: GNU Shepherd
services...


Linux-libre
initial RAM disk
Guile
PID 1: GNU Shepherd
services...
Guile


Linux-libre
initial RAM disk
Guile
PID 1: GNU Shepherd
services...
Guile
applications