

Functional Package Management with Guix

Ludovic Courtès

`ludo@gnu.org`

European Lisp Symposium

3 June 2013, Madrid

¡Hola!



(guile)

¡Hola!



¡Hola!



what's Guix?

<http://gnu.org/software/guix/>

- ▶ **functional package manager**
 - ▶ written in **Guile Scheme**
 - ▶ a new programming layer for **Nix**

what's Guix?

<http://gnu.org/software/guix/>

- ▶ **functional package manager**
 - ▶ written in **Guile Scheme**
 - ▶ a new programming layer for **Nix**
- ▶ **GNU's package manager**
 - ▶ foundation for the **GNU System**
 - ▶ GNU(/Linux) distro, est. 2012
 - ▶ focus on user freedom + consistent user interface

so what's Nix?

<http://nixos.org/nix/>

- ▶ another **functional package manager**
- ▶ basis of Guix
- ▶ foundation of NixOS GNU/Linux
 - ▶ GNU/Linux distro, est. 2006
 - ▶ i686, x86_64, armv5tel
 - ▶ \approx 8000 packages

so what's Nix?

<http://nixos.org/nix/>

- ▶ another **functional package manager**
- ▶ basis of Guix
- ▶ foundation of NixOS GNU/Linux
 - ▶ GNU/Linux distro, est. 2006
 - ▶ i686, x86_64, armv5tel
 - ▶ \approx 8000 packages
- ▶ more on Nix later...

Guix's main contributions

1. package description language **embedded in Scheme**
2. **build programs** written in Scheme

Guix's main contributions

1. package description language **embedded in Scheme**
 - ▶ benefit from Guile's **tooling** (compiler, i18n, etc.)
 - ▶ leverage Scheme macros for **domain-specific languages**
2. **build programs** written in Scheme

Guix's main contributions

1. package description language **embedded in Scheme**
 - ▶ benefit from Guile's **tooling** (compiler, i18n, etc.)
 - ▶ leverage Scheme macros for **domain-specific languages**
2. **build programs** written in Scheme
 - ▶ more **expressive** than Bash (!)
 - ▶ a single programming language → **two-tier** system

functional package management

features

foundations

Nix's approach

from Nix to Guix

rationale

programming interfaces

builder-side code

discussion

functional package management

features

foundations

Nix's approach

from Nix to Guix

rationale

programming interfaces

builder-side code

discussion

per-user, unprivileged package installation

```
alice@foo$ guix package --install=gcc
```

per-user, unprivileged package installation

```
alice@foo$ guix package --install=gcc
```

```
bob@foo$ guix package --install=gcc-4.7.3
```

per-user, unprivileged package installation

```
alice@foo$ guix package --install=gcc
alice@foo$ guix gc --references 'which gcc'
/nix/store/...-glibc-2.17
/nix/store/...-gcc-4.8.0
...

bob@foo$ guix package --install=gcc-4.7.3
```


per-user, unprivileged package installation

```
alice@foo$ guix package --install=gcc
alice@foo$ guix gc --references 'which gcc'
/nix/store/...-glibc-2.17
/nix/store/...-gcc-4.8.0
...
```

```
bob@foo$ guix package --install=gcc-4.7.3
bob@foo$ guix gc --references 'which gcc'
/nix/store/...-glibc-2.13
/nix/store/...-gcc-4.7.3
...
```

transparent binary/source deployment

```
alice@foo$ guix package --install=emacs
```

```
The following package will be installed:
```

```
  emacs-24.3 out /nix/store/...-emacs-24.3
```

```
The following files will be downloaded:
```

```
  /nix/store/...-emacs-24.3
```

```
  /nix/store/...-libxpm-3.5.10
```

```
  /nix/store/...-libxext-1.3.1
```

```
  /nix/store/...-libxaw-1.0.11
```

transparent binary/source deployment

```
alice@foo$ guix package --install=emacs
```

The following package will be installed:

```
emacs-24.3 out /nix/store/...-emacs-24.3
```

The following files will be **downloaded**:

```
/nix/store/...-libxext-1.3.1
```

```
/nix/store/...-libxaw-1.0.11
```

The following derivations will be **built**:

```
/nix/store/...-emacs-24.3.drv
```

```
/nix/store/...-libxpm-3.5.10.drv
```

transactional upgrades

```
$ guix package --upgrade
```

```
The following packages will be installed:
```

```
hop-2.4.0 out /nix/store/...-hop-2.4.0
```

```
gdb-7.6 out /nix/store/...-gdb-7.6
```

```
geiser-0.4 out /nix/store/...-geiser-0.4
```

```
glibc-2.17 out /nix/store/...-glibc-2.17
```

```
guile-2.0.9 out /nix/store/...-guile-2.0.9
```

```
...
```

transactional upgrades

```
$ guix package --upgrade
```

```
The following packages will be installed:
```

```
hop-2.4.0 out /nix/store/...-hop-2.4.0
```

```
gdb-7.6 out /nix/store/...-gdb-7.6
```

```
geiser-0.4 out /nix/store/...-geiser-0.4
```

```
glibc-2.17 out /nix/store/...-glibc-2.17
```

```
guile-2.0.9 out /nix/store/...-guile-2.0.9
```

```
...
```

```
$ hop --version ; guile --version
```

```
Hop-2.4.0
```

```
guile (GNU Guile) 2.0.9
```



transactional upgrades

```
$ guix package --upgrade
```

```
The following packages will be installed:
```

```
hop-2.4.0 out /nix/store/...-hop-2.4.0
```

```
gdb-7.6 out /nix/store/...-gdb-7.6
```

```
geiser-0.4 out /nix/store/...-geiser-0.4
```

```
glibc-2.17 out /nix/store/...-glibc-2.17
```

```
guile-2.0.9 out /nix/store/...-guile-2.0.9
```

```
...
```



transactional upgrades

```
$ guix package --upgrade
```

```
The following packages will be installed:
```

```
hop-2.4.0 out /nix/store/...-hop-2.4.0
```

```
gdb-7.6 out /nix/store/...-gdb-7.6
```

```
geiser-0.4 out /nix/store/...-geiser-0.4
```

```
glibc-2.17 out /nix/store/...-glibc-2.17
```

```
guile-2.0.9 out /nix/store/...-guile-2.0.9
```

```
...
```

(interrupted right in the middle)

```
$ hop --version ; guile --version
```

```
Hop-1.3.1
```

```
guile (GNU Guile) 1.8.8
```

transactional upgrades

```
$ guix package --upgrade
```

```
The following packages will be installed:
```

```
hop-2.4.0 out /nix/store/...-hop-2.4.0
```

```
gdb-7.6 out /nix/store/...-gdb-7.6
```

```
geiser-0.4 out /nix/store/...-geiser-0.4
```

```
glibc-2.17 out /nix/store/...-glibc-2.17
```

```
guile-2.0.9 out /nix/store/...-guile-2.0.9
```

```
...
```

(interrupted right in the middle)

```
$ hop --version ; guile --version
```

```
Hop-1.3.1
```

```
guile (GNU Guile) 1.8.8
```



per-user rollback

```
$ emacs --version  
GNU Emacs 24.2
```



per-user rollback



```
$ emacs --version  
GNU Emacs 24.2
```

```
$ guix package --upgrade=emacs
```

```
The following packages will be installed:
```

```
  emacs-24.3.1 out /nix/store/...-emacs-24.3.1
```

```
...
```

per-user rollback



```
$ emacs --version  
GNU Emacs 24.2
```

```
$ guix package --upgrade=emacs  
The following packages will be installed:  
  emacs-24.3.1 out /nix/store/...-emacs-24.3.1  
...
```

```
$ emacs --version  
Segmentation Fault
```

per-user rollback



```
$ emacs --version  
GNU Emacs 24.2
```

```
$ guix package --upgrade=emacs  
The following packages will be installed:  
  emacs-24.3.1 out /nix/store/...-emacs-24.3.1  
...
```

```
$ emacs --version  
Segmentation Fault
```

```
$ guix package --roll-back  
switching from generation 43 to 42
```

per-user rollback



```
$ emacs --version  
GNU Emacs 24.2
```

```
$ guix package --upgrade=emacs  
The following packages will be installed:  
  emacs-24.3.1 out /nix/store/...-emacs-24.3.1  
...
```

```
$ emacs --version  
Segmentation Fault
```

```
$ guix package --roll-back  
switching from generation 43 to 42
```

```
$ emacs --version  
GNU Emacs 24.2
```

functional package management

- features

- foundations**

- Nix's approach

from Nix to Guix

- rationale

- programming interfaces

- builder-side code

discussion

functional package management

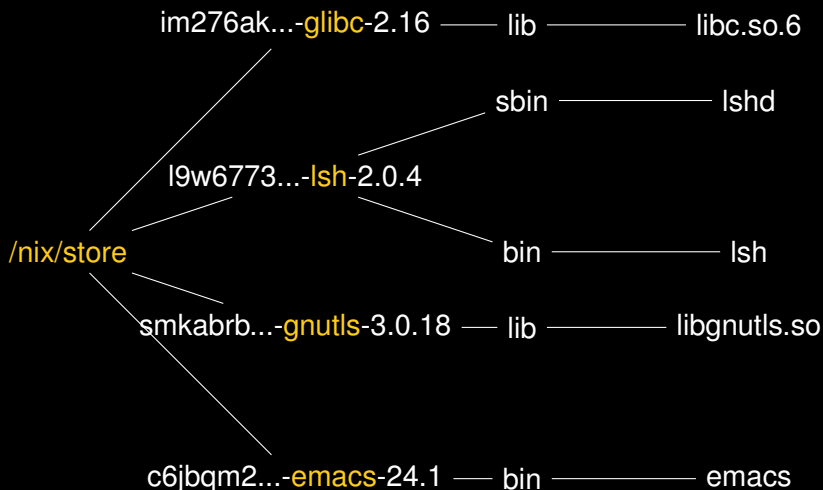
*regarding the build & installation process
of a package as a **pure function***

controlling the build environment

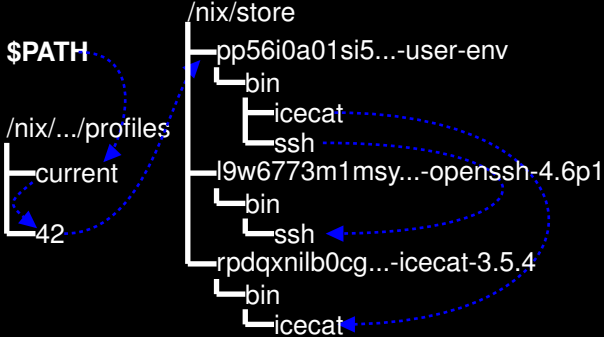
... as pioneered by Nix

1. one directory per installed package
2. immutable installation directories
3. undeclared dependencies invisible to the build process
4. build performed in chroot, with separate UID, etc.

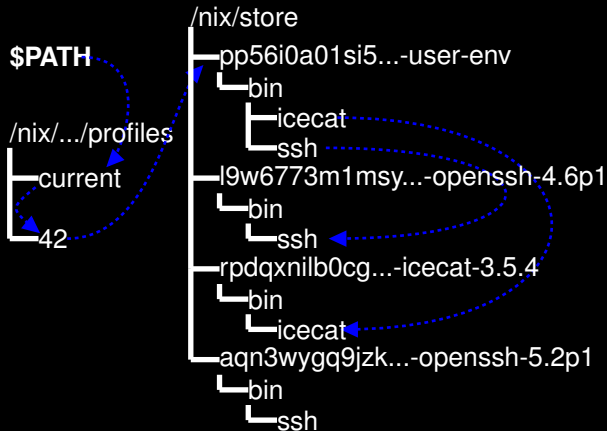
the store



user environments

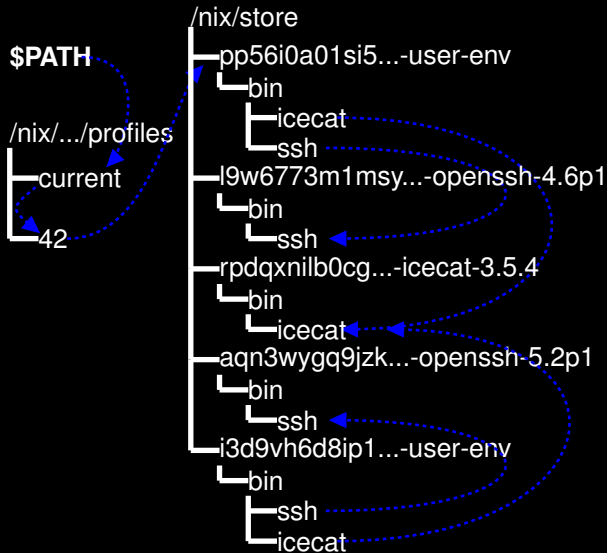


user environments



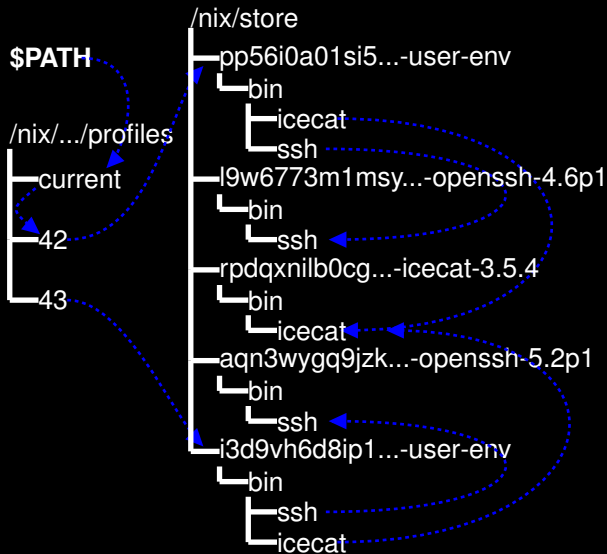
`guix package --upgrade=openssh`

user environments



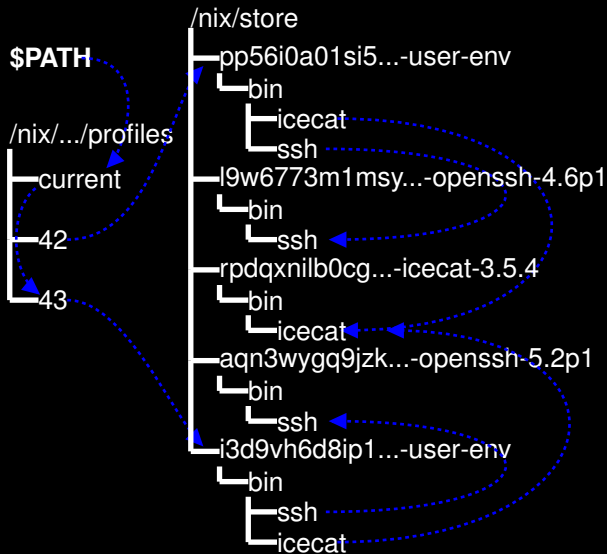
`guix package --upgrade=openssh`

user environments



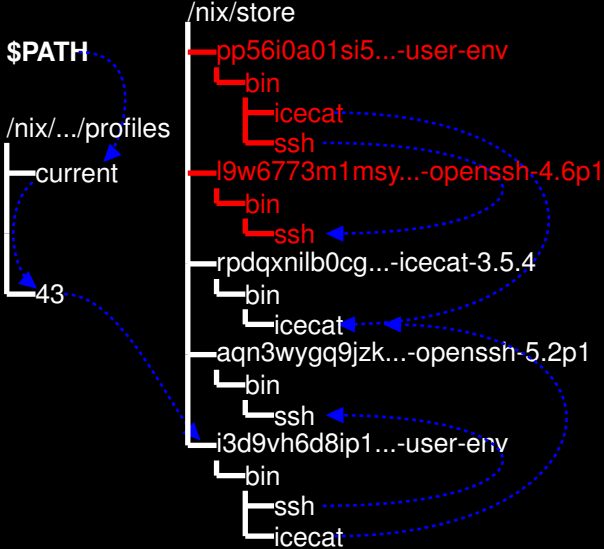
`guix package --upgrade=openssh`

user environments

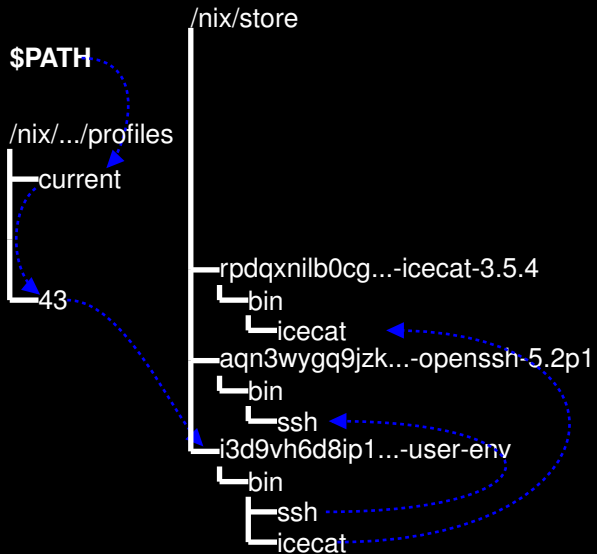


`guix package --upgrade=openssh`

user environments



user environments




guix gc

store file names

```
$ guix build guile
```

store file names

```
$ guix build guile  
/nix/store/ h2g4sc09h4... -guile-2.0.9
```



hash of *all* the dependencies

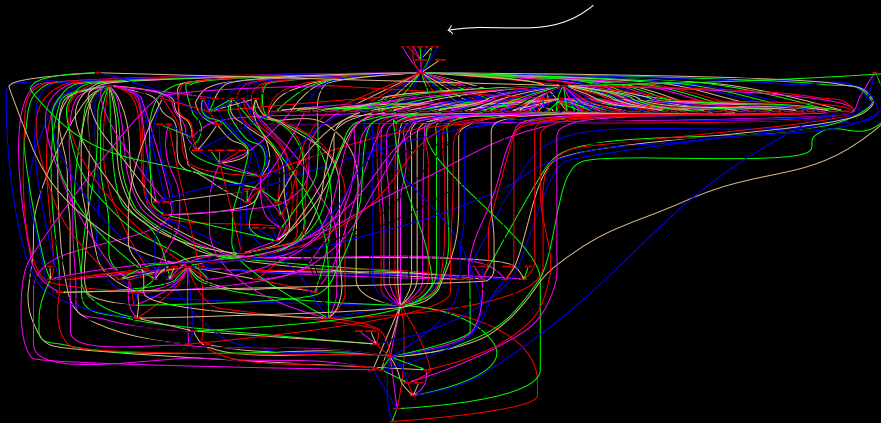
store file names

```
$ guix build guile  
/nix/store/h2g4sc09h4...-guile-2.0.9
```

```
$ guix gc --references /nix/store/...-guile-2.0.9  
/nix/store/4jl83jgzaac...-glibc-2.17  
/nix/store/iplay43cg58...-libunistring-0.9.3  
/nix/store/47p47v92cj9...-libffi-3.0.9  
/nix/store/drkwck2j965...-gmp-5.0.5  
...
```

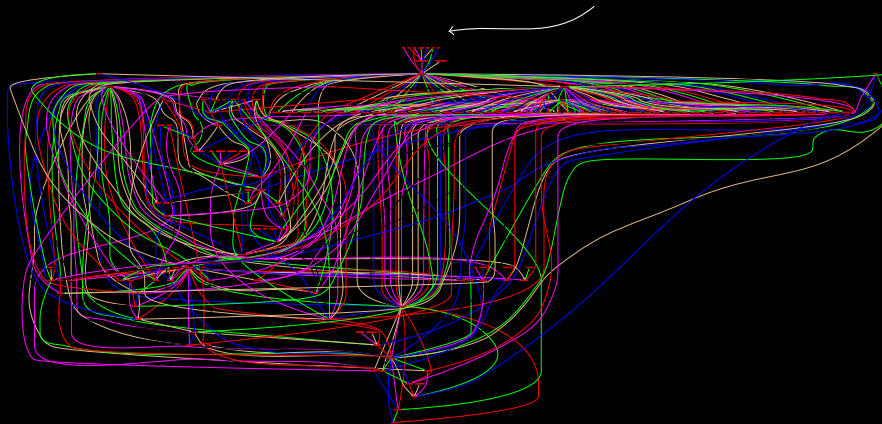
complete dependency specification

build-time dependencies of GNU Hello



complete dependency specification

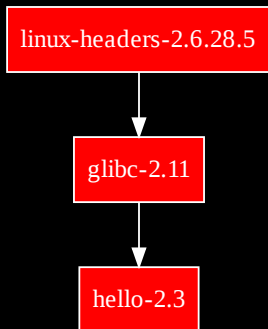
build-time dependencies of GNU Hello



... down to the compiler's compiler!

complete dependency specification

run-time dependencies of GNU Hello



run-time dependencies inferred by **conservative scanning**

functional packaging summarized

- ▶ **immutable** software installations
- ▶ builds/installs have **no side effects**
- ▶ build & deployment \equiv **calling a build function**
- ▶ the store \equiv **memoization**
- ▶ garbage collection...

functional package management

- features

- foundations

- Nix's approach**

- from Nix to Guix

 - rationale

 - programming interfaces

 - builder-side code

- discussion

Nix is twofold

functional package deployment

- ▶ the store
- ▶ file name hashes
- ▶ user environments
- ▶ transactional upgrades, etc.
- ▶ ...

Nix is twofold

functional package deployment

- ▶ the store
- ▶ file name hashes
- ▶ user environments
- ▶ transactional upgrades, etc.
- ▶ ...

Nix packaging language

- ▶ to describe package composition
- ▶ external DSL
- ▶ dynamically-typed, lazy
- ▶ easy integration of Bash snippets
- ▶ ...

Nix multi-user setup

build processes
chroot, separate UIDs

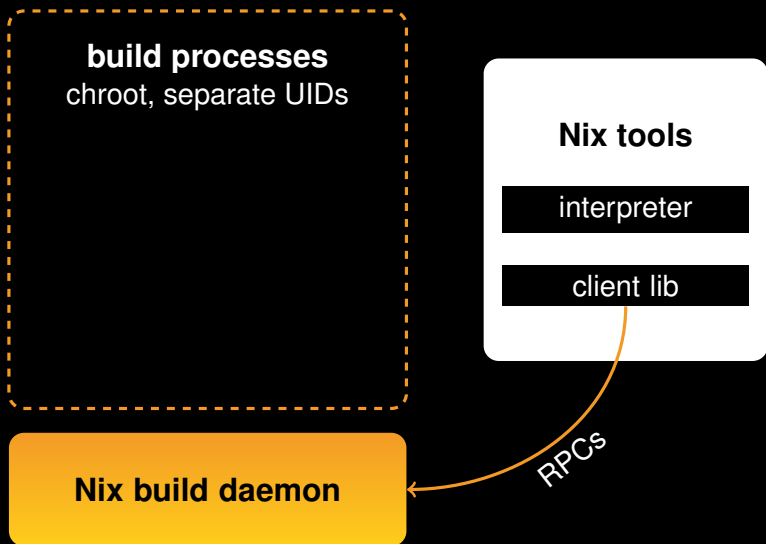
Nix build daemon

Nix tools

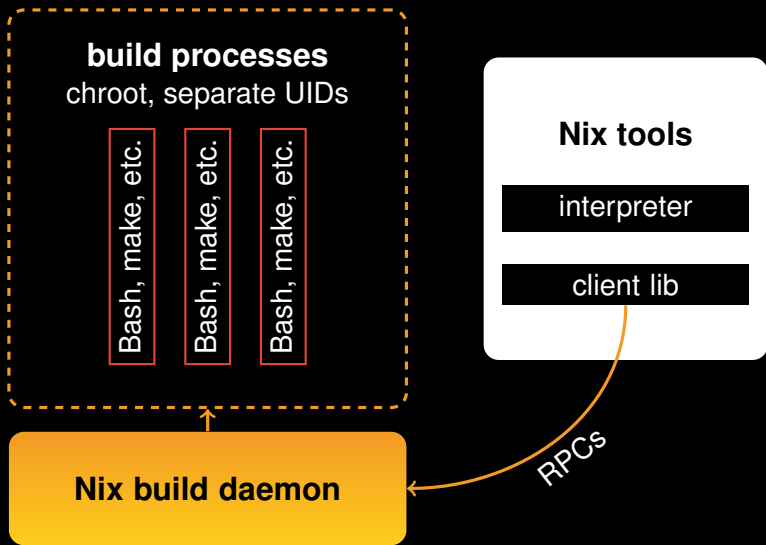
interpreter

client lib

Nix multi-user setup



Nix multi-user setup



Nix language build primitive

```
derivation {  
  name = "foo";  
  system = "x86_64-linux";  
  builder = "${./static-bash}";  
  args = [ "-c" "echo hello > " $out " ];  
}
```

Nix language build primitive

function call

```
derivation {  
  name = "foo";  
  system = "x86_64-linux";  
  builder = "${./static-bash}";  
  args = [ "-c" "echo hello > " $out " ];  
}
```

/nix/store/...-foo

named arguments

Nix language build primitive

```
let dep = derivation {
  name = "foo";
  system = "x86_64-linux";
  builder = "${./static-bash}";
  args = [ "-c" "echo hello > " $out " " ];
}; in derivation {
  name = "bar";
  system = "x86_64-linux";
  builder = "${./static-bash}";
  args = [ "-c"
    '' mkdir -p "$out"
      ln -s " ${dep} /some-result" "$out/my-result"
    '' ];
  PATH = "${coreutils}/bin";
}
```


Nix language build primitive

```
let dep = derivation {
  name = "foo";
  system = "x86_64-linux";
  builder = "${./static-bash}";
  args = [ "-c" "echo hello > " $out " " ];
}; in derivation {
  name = "bar";
  system = "x86_64-linux";
  builder = "${./static-bash}";
  args = [ "-c"
    '' mkdir -p "$out"
      ln -s " ${dep} /some-result" "$out/my-result"
    '' ];
  PATH = "${coreutils}/bin";
}
```



expands to /nix/store/...-foo

Nix language high-level packaging

```
{ fetchurl, stdenv } :  
stdenv . mkDerivation {  
  name = "hello-2.3";  
  src = fetchurl {  
    url = mirror://gnu/hello/hello-2.3.tar.bz2;  
    sha256 = "0c7vijq8y68...";  
  };  
  
  meta = {  
    description = "Produces a friendly greeting";  
    homepage = http://www.gnu.org/software/hello/;  
    license = "GPLv3+";  
  };  
}
```

function definition

formal parameters

function call

Nix language high-level packaging

```
gcc, make, etc.
{ fetchurl, stdenv , gettext } :
stdenv .mkDerivation {
  name = "hello-2.3";
  src = fetchurl {
    url = mirror://gnu/hello/hello-2.3.tar.bz2;
    sha256 = "0c7vijq8y68...";
  };
  buildInputs = [ gettext ];
  meta = {
    description = "Produces a friendly greeting";
    homepage = http://www.gnu.org/software/hello/;
    license = "GPLv3+";
  };
}
```

← dependency

Nix language high-level packaging

```
{ fetchurl, stdenv , gettext } :
```

```
stdenv .mkDerivation {
```

```
  name = "hello-2.3";
```

```
  src = fetchurl {
```

```
    url = mirror://gnu/hello/hello-2.3.tar.bz2;
```

```
    sha256 = "0c7vijq8y68...";
```

```
  };
```

```
  buildInputs = [ gettext ];
```

```
  preCheck = "echo 'Test suite coming up!'";
```

```
  meta = {
```

```
    description = "Produces a friendly greeting";
```

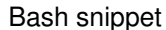
```
    homepage = http://www.gnu.org/software/hello/;
```

```
    license = "GPLv3+";
```

```
  };
```

```
}
```

Bash snippet



and now for parentheses...

functional package management

features

foundations

Nix's approach

from Nix to Guix

rationale

programming interfaces

builder-side code

discussion

functional package management

features

foundations

Nix's approach

from Nix to Guix

rationale

programming interfaces

builder-side code

discussion

The truth is that Lisp is not the right language for any particular problem. Rather, Lisp encourages one to attack a new problem by implementing new languages tailored to that problem.

– Albelson & Sussman, 1987

from Nix...

functional package deployment

- ▶ the store
- ▶ file name hashes
- ▶ user environments
- ▶ transactional upgrades, etc.
- ▶ ...

Nix packaging language

- ▶ to describe package composition
- ▶ external DSL
- ▶ dynamically-typed, lazy
- ▶ easy integration of Bash snippets
- ▶ ...

from Nix to Guix

functional package deployment

- ▶ the store
- ▶ file name hashes
- ▶ user environments
- ▶ transactional upgrades, etc.
- ▶ ...

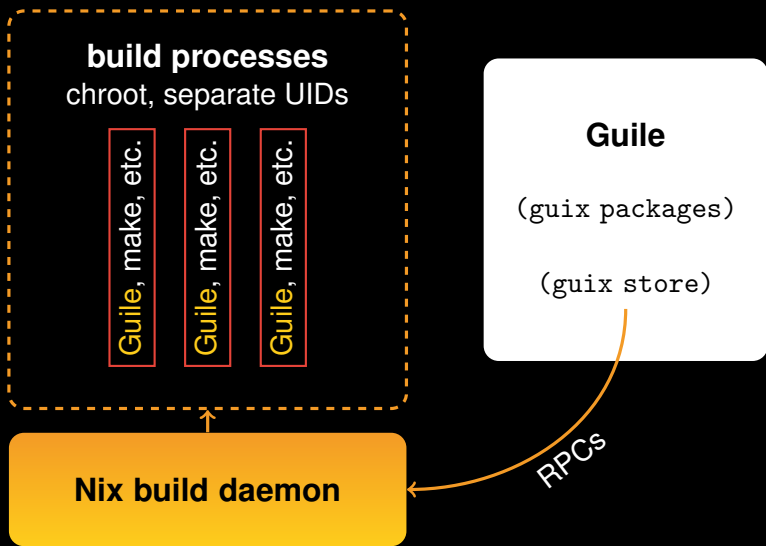
reuse this

Scheme!
Nix packaging language

- ▶ to describe package composition
- ▶ external DSL
- ▶ dynamically-typed, lazy
- ▶ easy integration of Bash snippets
- ▶ ...

Scheme!

Guix architecture



thesis

1. Scheme + EDSL at least as **expressive** as the Nix language
2. Scheme better suited than the shell for **build programs**
3. Guix provides a **unified & extensible** programming environment

functional package management

features

foundations

Nix's approach

from Nix to Guix

rationale

programming interfaces

builder-side code

discussion

programming interface layers

1. **declarative** packaging layer
2. Scheme **build expressions**
3. `derivation` primitive (from Nix)

declarative packaging layer

```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (origin
              (method url-fetch)
              (uri (string-append
                    "http://ftp.gnu.org/.../hello-" version
                    ".tar.gz"))
              (sha256 (base32 "0wqd...dz6")))))
  (build-system gnu-build-system)
  (synopsis "GNU Hello")
  (description "Produce a friendly greeting.")
  (home-page "http://www.gnu.org/software/hello/")
  (license gpl3+)))
```

declarative packaging layer

```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (origin
              (method url-fetch)
              (uri (string-append
                    "http://ftp.gnu.org/.../hello-" version
                    ".tar.gz"))
              (sha256 (base32 "0wqd...dz6"))))
    (build-system gnu-build-system)
    (synopsis "GNU Hello")
    (description "Produce a friendly greeting.")
    (home-page "http://www.gnu.org/software/hello/")
    (license gpl3+)))
```

how do we reach this level of abstraction?

Nix's derivation primitive in Scheme

```
(let* ((store (open-connection) )
      (bash ( add-to-store store "static-bash"
                          #t "sha256"
                          "./static-bash"))))
  ( derivation store "example-1.0"
    "x86_64-linux"
    bash
    '("-c" "echo hello > $out")
    '(("HOME" . "/homeless"))
    '()))
```

```
=> "/nix/store/nsswy...-example-1.0 .drv "
```

```
=> #<derivation "example-1.0" ...>
```

Nix's derivation primitive in Scheme

connect to the build daemon

```
(let* ((store (open-connection) )
      (bash ( add-to-store store "static-bash"
                        #t "sha256"
                        "./static-bash"))))
(derivation store "example-1.0"
  "x86_64-linux"
  bash
  '("-c" "echo hello > $out")
  '(("HOME" . "/homeless"))
  '()))
```

```
=> "/nix/store/nsswy...-example-1.0 .drv "
```

```
=> #<derivation "example-1.0" ...>
```

Nix's derivation primitive in Scheme

```
(let* ((store (open-connection))
      (bash (add-to-store store "static-bash"
                          #t "sha256"
                          "./static-bash"))))
  (derivation store "example-1.0"
              "x86_64-linux"
              bash
              `("-c" "echo hello > $out")
              `(("HOME" . "/homeless"))
              `()))
```

"intern" the file

/nix/store/...-static-bash

```
=> "/nix/store/nsswy...-example-1.0 .drv "
```

```
=> #<derivation "example-1.0" ...>
```

Nix's derivation primitive in Scheme

```
(let* ((store (open-connection) )
      (bash ( add-to-store store "static-bash"
                          #t "sha256"
                          "./static-bash"))))
  (derivation store "example-1.0"
             "x86_64-linux"
             bash
             ("echo hello > $out"
              "HOME" . "/homeless")))
  ()))
```

compute "derivation"—
i.e., build promise

```
=> "/nix/store/nsswy...-example-1.0.drv "
```

```
=> #<derivation "example-1.0" ...>
```

build expressions

```
(let* ((store (open-connection) )
      (builder '( begin
                  (mkdir %output)
                  (call-with-output-file
                     (string-append %output "/test")
                     (lambda (p)
                       (display '(hello guix) p))))))
      (drv ( build-expression->derivation
             store "foo" "x86_64-linux"
             builder
             '(("HOME" . "/nowhere"))))
      ( build-derivations store (list drv)))
```

build expressions

build script, to be eval'd in chroot

```
(let* ((store (open-connection))
      (builder '(begin
                 (mkdir %output)
                 (call-with-output-file
                  (string-append %output "/test")
                  (lambda (p)
                    (display '(hello guix) p))))))
      (drv (build-expression->derivation
            store "foo" "x86_64-linux"
            builder
            '(("HOME" . "/nowhere"))))
      (build-derivations store (list drv)))
```

build expressions

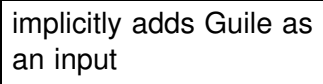
```
(let* ((store (open-connection) )
      (builder '( begin
                  (mkdir %output)
                  call-with-output-file
                  (string-append %output "/test")
                  (lambda (p)
                    (display '(hello guix) p))))))
      (drv ( build-expression->derivation
            store "foo" "x86_64-linux"
            builder
            '(("HOME" . "/nowhere"))))
      ( build-derivations store (list drv)))
```

compute derivation for this builder, system, and env. vars

build expressions

```
(let* ((store (open-connection))
      (builder '(begin
                 (mkdir %output)
                 (call-with-output-file
                  (string-append %output "/test")
                  (lambda (p)
                    (display '(hello guix) p))))))
      (drv (build-expression->derivation
            store "foo" "x86_64-linux"
            builder
            '(("HOME" . "/nowhere"))))
      (build-derivations store (list drv)))
```

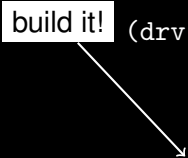
implicitly adds Guile as
an input



build expressions

```
(let* ((store (open-connection) )
      (builder '( begin
                  (mkdir %output)
                  (call-with-output-file
                     (string-append %output "/test")
                     (lambda (p)
                       (display '(hello guix) p))))))
      (drv ( build-expression->derivation
             store "foo" "x86_64-linux"
             builder
             '(("HOME" . "/nowhere"))))
      ( build-derivations store (list drv)))
```

build it!



declarative packaging layer

```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (origin
              (method url-fetch)
              (uri (string-append
                    "http://ftp.gnu.org/.../hello-" version
                    ".tar.gz"))
              (sha256 (base32 "0wqd...dz6")))))
  (build-system gnu-build-system )


  (inputs '(("gawk" , gawk )))

  (synopsis "GNU Hello")
  (description "Produce a friendly greeting.")
  (home-page "http://www.gnu.org/software/hello/")
  (license gpl3+)))
```

declarative packaging layer

```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (origin
              (method url-fetch)
              (uri (string-append
                    "http://ftp.gnu.org/.../hello-" version
                    ".tar.gz"))
              (sha256 (base32 "0wqd...dz6")))))
  (build-system gnu-build-system )

  (inputs '(("gawk" , gawk )))
  (synopsis "GNU Hello")
  (description "Produce a friendly greeting")
  (home-page "http://www.gnu.org/software/hello/")
  (license gpl3+)))
```



dependencies

declarative packaging layer

```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (origin
              (method url-fetch)
              (uri (string-append
                    "http://ftp.gnu.org/.../hello-" version
                    ".tar.gz"))
              (sha256 (base32 "0wqd...d=6"))))
    (build-system gnu-build-system)
    (inputs '(("gawk" , gawk)))
    (synopsis "GNU Hello")
    (description "Produce a friendly greeting")
    (home-page "http://www.gnu.org/software/hello/")
    (license gpl3+)))
```

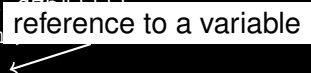
reference to a variable

dependencies

declarative packaging layer

```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (origin
              (method url-fetch)
              (uri (string-append
                    "http://ftp.gnu.org/.../hello-" version
                    ".tar.gz"))
              (sha256 (base32 "0wqda..."))))
    (build-system gnu-build-system)
    (inputs '("gawk" , my-other-awk)))
  (synopsis "GNU Hello")
  (description "Produce a friendly greeting.")
  (home-page "http://www.gnu.org/software/hello/")
  (license gpl3+)))
```

reference to a variable



declarative packaging layer

```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (method url-fetch
                  (uri (string-append
                        "http://ftp
                        ".tar.gz"))
                  (sha256 (base32 "0wqd...dz6")))))
    (build-system gnu-build-system )

    (inputs '(("gawk" , gawk )))

    (synopsis "GNU Hello")
    (description "Produce a friendly greeting.")
    (home-page "http://www.gnu.org/software/hello/")
    (license gpl3+)))
```

depends on gcc, make, bash, etc.

build-system protocol

```
(define gnu-build-system
  (build-system (name 'gnu)
    (description "./configure && make && make install")
    (build gnu-build)
    (cross-build gnu-cross-build)))
```

build-system protocol

```
(define gnu-build-system
  (build-system (name 'gnu)
    (description "./configure && make && make install")
    (build gnu-build)
    (cross-build gnu-cross-build)))
```

- ▶ python-build-system → python setup.py
- ▶ perl-build-system → perl Makefile.PL
- ▶ cmake-build-system → cmake .

building packages

```
(use-modules (guix packages) (guix store)
             (gnu packages base))
```

```
(define store
  (open-connection) )
```

connect to the Nix build daemon



```
(package? hello)
=> #t
```

building packages

```
(use-modules (guix packages) (guix store)
             (gnu packages base))
```

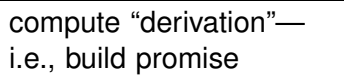
```
(define store
  (open-connection) )
```

```
(package? hello)
```

```
=> #t
```

```
(define drv (package-derivation store hello))
```

compute “derivation”—
i.e., build promise



building packages

```
(use-modules (guix packages) (guix store)
             (gnu packages base))
```

```
(define store
  (open-connection) )
```

```
(package? hello)
```

```
=> #t
```

```
(define drv (package-derivation store hello))
```

```
drv
```

```
=> "/nix/store/xyz...-hello-2.8.drv"
```

building packages

```
(use-modules (guix packages) (guix store)
             (gnu packages base))
```

```
(define store
  (open-connection) )
```

```
(package? hello)
```

```
=> #t
```

```
(define drv (package-derivation store hello))
```

```
drv
```

```
=> "/nix/store/xyz...-hello-2.8.drv"
```

```
(build-derivations (list drv))
```

... daemon builds/downloads package on our behalf...

building packages

```
(use-modules (guix packages) (guix store)
             (gnu packages base))
```

```
(define store
  (open-connection) )
```

```
(package? hello)
```

```
=> #t
```

```
(define drv (package-derivation store hello))
```

```
drv
```

```
=> "/nix/store/xyz...-hello-2.8.drv"
```

```
(build-derivations (list drv))
```

```
... daemon builds/downloads package on our behalf...
```

```
=> "/nix/store/pqr...-hello-2.8"
```

building packages

```
$ guix build hello
```

building packages

```
$ guix build hello
```

```
The following derivations will be built:
```

```
  /nix/store/4gy79...-gawk-4.0.0.drv
```

```
  /nix/store/7m2r9...-hello-2.8.drv
```

```
...
```

```
/nix/store/71aj1...-hello-2.8
```

building packages

```
$ guix build --target=armel-linux-gnueabi hello
```

```
The following derivations will be built:
```

```
  /nix/store/1gm99...-gcc-armel-linux-gnu-4.8.1.drv
```


```
  /nix/store/71ah1...-hello-2.8.drv
```

```
...
```

```
/nix/store/7m2r9...-hello-2.8
```


packages based on existing ones

copy fields from hello except
for version and source



```
(package (inherit hello)
  (version "2.7")
  (source
    (origin
      (method url-fetch)
      (uri "mirror://gnu/hello/hello-2.7.tar.gz")
      (sha256
        (base32 "7dqw3..."))))))
```

functional package adapters

```
(define (static-package p)
  ;; Return a statically-linked variant of P.
  (package (inherit p)
    (arguments
      '(:configure-flags '--disable-shared
        "LDFLAGS=-static")
      ,@(package-arguments p))))
```

system-dependent arguments

```
(define gawk
  (package
    (name "gawk")
    (version "4.0.2")
    (source (origin (method url-fetch)
                    (uri "http://ftp.gnu.org/...")
                    (sha256 (base32 "0sss...")))))
  (build-system gnu-build-system)
  (arguments
    (if (string-prefix? "i686" (%current-system))
        '(#:tests? #f) ; skip tests on 32-bit hosts
        '()))
  (inputs '("libsigsegv" ,libsigsegv))
  (home-page "http://www.gnu.org/software/gawk/")
  (synopsis "GNU Awk")))
```

system-dependent arguments

```
(define gawk
  (package
    (name "gawk")
    (version "4.0.2")
    (source (origin (method url-fetch)
                    (uri "http://ft
                        (sha256 (base32 "0sss...")))))
    (build-system gnu-build-system)
    (arguments
      (if (string-prefix? "i686" (%current-system))
          '(:tests? #f) ; skip tests on 32-bit hosts
          '()))
    (inputs '(("libsigsegv" ,libsigsegv)))
    (home "http://www.gnu.org/software/gawk/")
    (synopsis "GNU Awk")))
```

dynamically-scoped parameter (SRFI-39)

evaluated within the dynamic extent of package-derivation

under the hood: fancy records

```
(define-record-type* <package>
  package make-package package?

  (name package-name)
  (version package-version)
  (source package-source)
  (build-system package-build-system)
  (arguments package-arguments
              (default '()) (thunked))

  (inputs package-inputs
          (default '()) (thunked))

;; ...

(location package-location
  (default (current-source-location))))
```

under the hood: fancy records

```
(define-record-type* <package>
  package make-package package?
  (name package-name)
  (version package-version)
  (source package-source)
  (build-system package-build-system)
  (arguments package-arguments
              (default '()) (thunked))
  (inputs package-inputs
           (default '()) (thunked))
  ;; ...

  (location package-location
             (default (current-source-location))))
```

generated macro

enclose value in a thunk

functional package management

features

foundations

Nix's approach

from Nix to Guix

rationale

programming interfaces

builder-side code

discussion

builder side of gnu-build-system

```
(define %standard-phases
  '((configure . ,configure)
    (build . ,build)
    ;; ...
  ))

(define* (gnu-build #:key (phases %standard-phases)
                  #:allow-other-keys
                  #:rest args)
  ;; Run all the PHASES in order, passing them ARGS.
  (every (match-lambda
          ((name . proc)
           (format #t "starting phase '~a'~%" name)
           (let ((result (apply proc args)))
             (format #t "phase '~a' done~%" name)
             result))))
         phases))
```


inserting a build phase

```
(define howdy
  (package (inherit hello)
    (arguments
      '(:phases
        (alist-cons-after
          'configure 'change-hello
          (lambda* (:key system #:allow-other-keys)
            (substitute* "src/hello.c"
              (("Hello, world!")
                (string-append "Howdy! Running on "
                               system ".")))))
          %standard-phases )))))
```

inserting a build phase

```
(define howdy
  (package (inherit hello)
    (arguments
      '(:phases
        ( alist-cons-after
          'configure 'change-hello
          (lambda* (#:key system #:allow-other-keys)
            ( substitute* "src/hello.c"
              ("Hello, world!")
              (string-append "Howdy! Running on "
                             system ".")))))
          %standard-phases )))))
```

builder-side expression

inserting a build phase

```
(define howdy
  (package (inherit hello)
    (arguments
      '(:phases
        (alist-cons-after
          'configure 'change-hello
          (lambda* (#:key system #:allow-other-keys)
            (string-append "src/hello.c"
              " " "hello, world!")
              (string-append "Howdy! Running on "
                system ".")))))
        %standard-phases )))))
```

add a phase before configure

configure, build, check, install

inserting a build phase

```
(define howdy
  (package (inherit hello)
    (arguments
      '(:phases
        (alist-cons-after
          'configure 'change-hello
          (lambda* (#:key system #:allow-other-keys)
            ( substitute* "src/hello.c"
              ("Hello, world!")
              (string-append "Howdy! Running on "
                             system ".")))))
      %standard-phases )))))
```

patch things up à la sed

downloading sources

```
(origin
  (method url-fetch )
  (uri (string-append "mirror://gnu/gcc/gcc-"
                      version "/gcc-" version
                      ".tar.bz2"))
  (sha256 (base32 "1hx9...")))
```

downloading sources

use Guile HTTP(S)/FTP client



```
(origin  
  (method url-fetch )  
  (uri (string-append "mirror://gnu/gcc/gcc-"  
                      version "/gcc-" version  
                      ".tar.bz2"))  
  (sha256 (base32 "1hx9...")))
```

downloading sources

use Guile HTTP(S)/FTP client

```
(origin  
  (method url-fetch )  
  (uri (string-append "mirror://gnu/gcc/gcc-"  
                      version "/gcc-" version  
                      ".tar.bz2"))  
  (sha256 (base32 "1hx9...")))
```

how is the very first
tarball downloaded?

bootstrapping the distribution

0. statically-linked binaries of `mkdir`, `tar`, `xz`, `bash`, and `Guile`

bootstrapping the distribution

0. statically-linked binaries of `mkdir`, `tar`, `xz`, `bash`, and Guile
1. derivation runs Bash script to untar Guile

bootstrapping the distribution

0. statically-linked binaries of `mkdir`, `tar`, `xz`, `bash`, and Guile
1. derivation runs Bash script to untar Guile
2. use Guile to download statically-linked binaries of GCC, Binutils, `libc`, `Coreutils` et al., and Bash

bootstrapping the distribution

0. statically-linked binaries of `mkdir`, `tar`, `xz`, `bash`, and Guile
1. derivation runs Bash script to untar Guile
2. use Guile to download statically-linked binaries of GCC, Binutils, libc, Coreutils et al., and Bash
3. use that to build GNU Make
4. ...

functional package management

- features

- foundations

- Nix's approach

from Nix to Guix

- rationale

- programming interfaces

- builder-side code

discussion

status

- ▶ API/language support for builds & composition
- ▶ builder-side libs equiv. to `wget`, `find`, `grep`, `sed`, etc.
- ▶ expressive enough to build a variety of packages

benefits of DSL embedding

1. Guile tools readily available
2. simplified implementation of auxiliary tools

benefits of DSL embedding

1. Guile tools readily available
 - ▶ libraries, macros, compiler, etc.
 - ▶ i18n support (for package descriptions)
 - ▶ development environment: Emacs + Geiser
2. simplified implementation of auxiliary tools
 - ▶ off-line & on-line package auto-updater
 - ▶ description synchronization with external DB
 - ▶ searching packages by keyword

GNU/Linux distribution

- ▶ installable atop a running GNU/Linux system
- ▶ self-contained (pure!)
- ▶ transactional upgrade/roll-back, pre-built binaries, etc.
- ▶ \approx 400 packages
 - ▶ TeX Live, Xorg, GCC, ...
 - ▶ and 6 Scheme implementations! :-)

pushing the limits: booting to Guile

```
(expression->initrd
  '(begin
    (mkdir "/proc")
    (mount "none" "/proc" "proc")

    ;; Load Linux kernel modules.
    (let ((slurp (lambda (module)
                  (call-with-input-file
                     (string-append "/modules/" module)
                     get-bytevector-all))))
      (for-each (compose load-linux-module slurp)
                (list "md4.ko" "ecb.ko" "cifs.ko")))

    ;; Turn eth0 up.
    (let ((sock (socket AF_INET SOCK_STREAM 0)))
      (set-network-interface-flags sock "eth0" IFF_UP))

    ;; At last, the warm and friendly REPL.
    (start-repl)))
```

road map

- ▶ **short-term**

- ▶ tweak more packages for cross-compilation
- ▶ port to mips64el (N64), and armel (?)
- ▶ more packages: GTK+ stack, applications

road map

▶ short-term

- ▶ tweak more packages for cross-compilation
- ▶ port to mips64el (N64), and armel (?)
- ▶ more packages: GTK+ stack, applications

▶ medium-term

- ▶ stand-alone, bootable distribution!
- ▶ with NixOS-style whole-system configuration EDSL
- ▶ with the Guile-powered DMD init system

road map

▶ short-term

- ▶ tweak more packages for cross-compilation
- ▶ port to mips64el (N64), and armel (?)
- ▶ more packages: GTK+ stack, applications

▶ medium-term

- ▶ stand-alone, bootable distribution!
- ▶ with NixOS-style whole-system configuration EDSL
- ▶ with the Guile-powered DMD init system

Your help needed!

the first no-compromise GNU distribution

The First No-Compromise LISP Machine



LAMBDA

summary

- ▶ **features**

- ▶ transactional upgrades; rollback; per-user profiles
- ▶ full power of Guile to build & compose packages
- ▶ unified packaging development environment

- ▶ **foundations**

- ▶ purely functional package management
- ▶ packaging DSL embedded in Scheme
- ▶ second tier: flexible builds programs in Scheme

ludo@gnu.org



<http://gnu.org/software/guix/>

Copyright © 2010, 2012, 2013 Ludovic Courtès ludo@gnu.org.

Picture of user environments is:

Copyright © 2009 Eelco Dolstra e.dolstra@tudelft.nl.

Copyright of other images included in this document is held by their respective owners.

This work is licensed under the [Creative Commons Attribution-Share Alike 3.0](https://creativecommons.org/licenses/by-sa/3.0/) License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

At your option, you may instead copy, distribute and/or modify this document under the terms of the [GNU Free Documentation License, Version 1.3 or any later version](https://www.gnu.org/licenses/gfdl.html) 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 available at <http://www.gnu.org/licenses/gfdl.html>.

The source of this document is available from <http://git.sv.gnu.org/cgi/guix/maintenance.git>.