What is Guix?

Josselin Poiret

Nantes Université ; Gallinette Team, Inria

November 8, 2023

Who has heard about Guix before?

Who has heard about Guix before?

Who could concisely describe what Guix can do?

Guix seems hard to pin down.

Guix seems hard to pin down. You may have heard any combination of the following:

Guix seems hard to pin down. You may have heard any combination of the following:

- Guix is a functional package manager;

Guix seems hard to pin down. You may have heard any combination of the following:

- Guix is a functional package manager;
- Guix is a Linux distribution;

Guix seems hard to pin down. You may have heard any combination of the following:

- Guix is a functional package manager;
- Guix is a Linux distribution;
- Guix is a computing environment tool;

Guix seems hard to pin down. You may have heard any combination of the following:

- Guix is a functional package manager;
- Guix is a Linux distribution;
- Guix is a computing environment tool;
- Guix is a time-machine;

Guix seems hard to pin down. You may have heard any combination of the following:

- Guix is a functional package manager;
- Guix is a Linux distribution;
- Guix is a computing environment tool;
- Guix is a time-machine;
- Guix is just French Nix.

Guix seems hard to pin down. You may have heard any combination of the following:

- Guix is a functional package manager;
- Guix is a Linux distribution;
- Guix is a computing environment tool;
- Guix is a time-machine;
- Guix is just French Nix.

Make up your mind!

We'll focus on how Guix helps us achieve reproducibility.

But also some other nice Guix features along the way!

Guix is a package manager:

```
$ guix install agda
[...]
$ agda MyCoolTheorem.agda

$ guix install ocaml
[...]
$ ocamlopt simulation.ml
```

Guix is a package manager:

```
$ guix install agda
[...]
$ agda MyCoolTheorem.agda

$ guix install ocaml
[...]
$ ocamlopt simulation.ml
```

It can run on top of any distribution and won't interfere with it.

More importantly, Guix is a *functional* package manager.

More importantly, Guix is a *functional* package manager.

In Guix, packages are pure functions

$$\text{dependencies} + \text{source} \mapsto \text{build output}$$

i.e. if you give it the same dependencies and source, it should give you *exactly* the same result, bit-for-bit.

Guix achieves this by storing each build output in `/gnu/store`, under a directory

$$\mathrm{hash}(\mathrm{dependencies}, \mathrm{recipe}, \mathrm{source})\texttt{-name-version}$$

This has several ramifications:

This has several ramifications:

- Multiple library versions can co-exist (even the same library version but built with different dependencies!);

This has several ramifications:

- Multiple library versions can co-exist (even the same library version but built with different dependencies!);
- We can easily identify references to dependencies in a build output;

This has several ramifications:

- Multiple library versions can co-exist (even the same library version but built with different dependencies!);
- We can easily identify references to dependencies in a build output;
- We never modify state! You don't need to do a rebuild dance when a library is updated;

This has several ramifications:

- Multiple library versions can co-exist (even the same library version but built with different dependencies!);
- We can easily identify references to dependencies in a build output;
- We never modify state! You don't need to do a rebuild dance when a library is updated;
- We know the output store path beforehand, so we can download it from a substitute server instead!

How do we make package builds free of side-effects? What prevents other package managers/build systems from building reproducibly?

The Docker example

Alice uses a Docker image hosted on nicehub.com for their research, and tells Bob that if they want to reproduce it, they can just do `docker pull cool-research` and start using it as well.

The Docker example

Alice uses a Docker image hosted on nicehub.com for their research, and tells Bob that if they want to reproduce it, they can just do `docker pull cool-research` and start using it as well.

Bob, whose mailbox was full at the time, only receives that mail 2 years later, but is excited to try it out! They run the command, only to realise that nicehub.com has decided to delete all built images for free users...

The Docker example

Alice uses a Docker image hosted on nicehub.com for their research, and tells Bob that if they want to reproduce it, they can just do `docker pull cool-research` and start using it as well.

Bob, whose mailbox was full at the time, only receives that mail 2 years later, but is excited to try it out! They run the command, only to realise that nicehub.com has decided to delete all built images for free users...

"I'll just rebuild it myself, then!". But Bob opens the Dockerfile, only to discover the following first two lines:

The Docker example

Alice uses a Docker image hosted on nicehub.com for their research, and tells Bob that if they want to reproduce it, they can just do `docker pull cool-research` and start using it as well.

Bob, whose mailbox was full at the time, only receives that mail 2 years later, but is excited to try it out! They run the command, only to realise that nicehub.com has decided to delete all built images for free users...

"I'll just rebuild it myself, then!". But Bob opens the Dockerfile, only to discover the following first two lines:

```
FROM ubuntu@latest
RUN apt-get update && apt-get upgrade
```

We follow a strict protocol!

We follow a strict protocol!

- Isolate builds from the rest of the system (not unlike Docker!);

We follow a strict protocol!

- Isolate builds from the rest of the system (not unlike Docker!);
- But also from the network (unlike Docker!);

We follow a strict protocol!

- Isolate builds from the rest of the system (not unlike Docker!);
- But also from the network (unlike Docker!);
- Only sources whose hash we know beforehand can be fetched from the network;

We follow a strict protocol!

- Isolate builds from the rest of the system (not unlike Docker!);
- But also from the network (unlike Docker!);
- Only sources whose hash we know beforehand can be fetched from the network;
- Patch out unwanted behavior from underlying build tools: "Why does build system X embed the prime factors of the current date in the resulting binary???" and other fun things.

Making everything reproducible is a herculean effort. However, Guix has tools to help us along the way:

```
$ guix build --check --rounds=5 --no-substitutes
    --no-grafts libyaml
[...]
successfully built /gnu/store/6c6rs67wf0jwqnwqm821v
libyaml-0.2.5.drv
```

Making everything reproducible is a herculean effort. However, Guix has tools to help us along the way:

```
$ guix build --check --rounds=5 --no-substitutes
    --no-grafts libyaml
[...]
successfully built /gnu/store/6c6rs67wf0jwqnwqm821v
libyaml-0.2.5.drv
```

You can also challenge substitute servers to reproducibility duels with `guix challenge`.

Guix QA: Reproducible builds

The following table gives an overview of packages that can be built reproducibly, as well as known issues. All data is from th

System	Package reproducibility			
	Matching	Not matching	Unknown	
aarch64-linux	17490	1748	12332	
armhf-linux	7057	696	18931	
i586-gnu	5	0	20683	
i686-linux	22750	1459	5202	
powerpc64le-linux	7648	168	19013	
riscv64-linux	0	0	26614	
x86_64-linux	28295	4094	1536	

Guix, unlike language-specific package managers, takes care of the *full* stack of dependencies.

Example: Stack claims that its main design point is reproducible builds[1]. It also manages your Haskell toolchain. But where do those Haskell toolchain binaries come from, and how do they work?

[1] `https://docs.haskellstack.org/en/stable/GUIDE/`

Stack 2.13.1 uses `ghc-build`:

- `musl` to indicate `libc.musl-x86_64.so.1` is present and Stack should use the GHC binary distribution for Alpine Linux.
- `tinfo6` to indicate `libgmp.so.10` and `libtinfo.so.6` are present and `libc6` is compatible with `libc6` 2.32.
- `tinfo6-libc6-pre232` to indicate `libgmp.so.10` and `libtinfo.so.6` are present and `libc6` is not compatible with `libc6` 2.32.
- `ncurses6` to indicate `libgmp.so.10` and `libncursesw.so.6` are present
- `gmp4` to indicate `libgmp.so.3` is present

By default, Stack associates:

- the `tinfo6` build with the 'Fedora 33' binary distribution of GHC 9.4.1 to 9.4.4. Those binary distributions require versions of `libc6` that are compatible with `libc6` 2.32; and
- the `tinfo6-libc6-pre232` build with the 'Debian 10' binary distribution of GHC 9.4.1 to 9.4.4. Those binary distributions require versions of `libc6` that are compatible with `libc6` 2.28.

The same can be said for external dependencies: what if you need bindings to a C library? You want *all* the software to be managed by one tool.

What about when sources disappear? Are we out of luck?

What about when sources disappear? Are we out of luck?

- Guix can download sources from SWH if upstream's unavailable;
- Guix queues packages' sources for inclusion into SWH.

Development environments

Guix doesn't install your software in e. g. `/bin`, just makes it available by pointing to things in the store.

Development environments

Guix doesn't install your software in e. g. `/bin`, just makes it available by pointing to things in the store.

This also means we can do that temporarily! Suppose I want to build an OCaml program once, I can just do:

```
$ guix shell ocaml -- ocamlopt hello.ml -o hello
[...]
$ ./hello
```

For more serious projects, you can list the dependencies in a `manifest.scm` file, and run

```
$ guix shell -m manifest.scm
```

For more serious projects, you can list the dependencies in a `manifest.scm` file, and run

```
$ guix shell -m manifest.scm
```

You can even distribute that file with your project for others to use!

Going back in time

Guix is rolling-release: packages are updated as we go along. How can we recover environments used in the past?

Going back in time

Guix is rolling-release: packages are updated as we go along. How can we recover environments used in the past?

```
$ guix describe
Generation 75 Oct 31 2023 10:25:56 (current)
  guix c089537
    repository URL: https://git.savannah.gnu.org/git/guix.git
    branch: master
    commit: c0895371c5759c7d9edb330774e90f192cc4cf2c
```

Going back in time

Guix is rolling-release: packages are updated as we go along. How can we recover environments used in the past?

```
$ guix describe
Generation 75 Oct 31 2023 10:25:56 (current)
  guix c089537
    repository URL: https://git.savannah.gnu.org/git/guix.git
    branch: master
    commit: c0895371c5759c7d9edb330774e90f192cc4cf2c
```

Guix can describe itself, and you can also ask Guix to use an older version of itself to run some commands:

```
$ guix describe -f channels > channels.scm
[ some time later ]
$ guix time-machine -C channels.scm -- shell ocaml
```

Sharing the `channels.scm` and `manifest.scm` files lets anyone reproduce your environment with the magic

```
$ guix time-machine -C channels.scm
    -- shell -m manifest.scm
```

Exporting environments

What if Alice and Bob are working together, but Bob doesn't use Guix (yet)?

Exporting environments

What if Alice and Bob are working together, but Bob doesn't use Guix (yet)?

Alice can send a pack containing their whole computing environment with `guix pack`, for use on any other Linux system (even WSL!).

Exporting environments

What if Alice and Bob are working together, but Bob doesn't use Guix (yet)?

Alice can send a pack containing their whole computing environment with `guix pack`, for use on any other Linux system (even WSL!).

```
$ guix pack -RR -S /bin=bin -S /etc=etc
    emacs emacs-agda2-mode agda agda-cubical
```

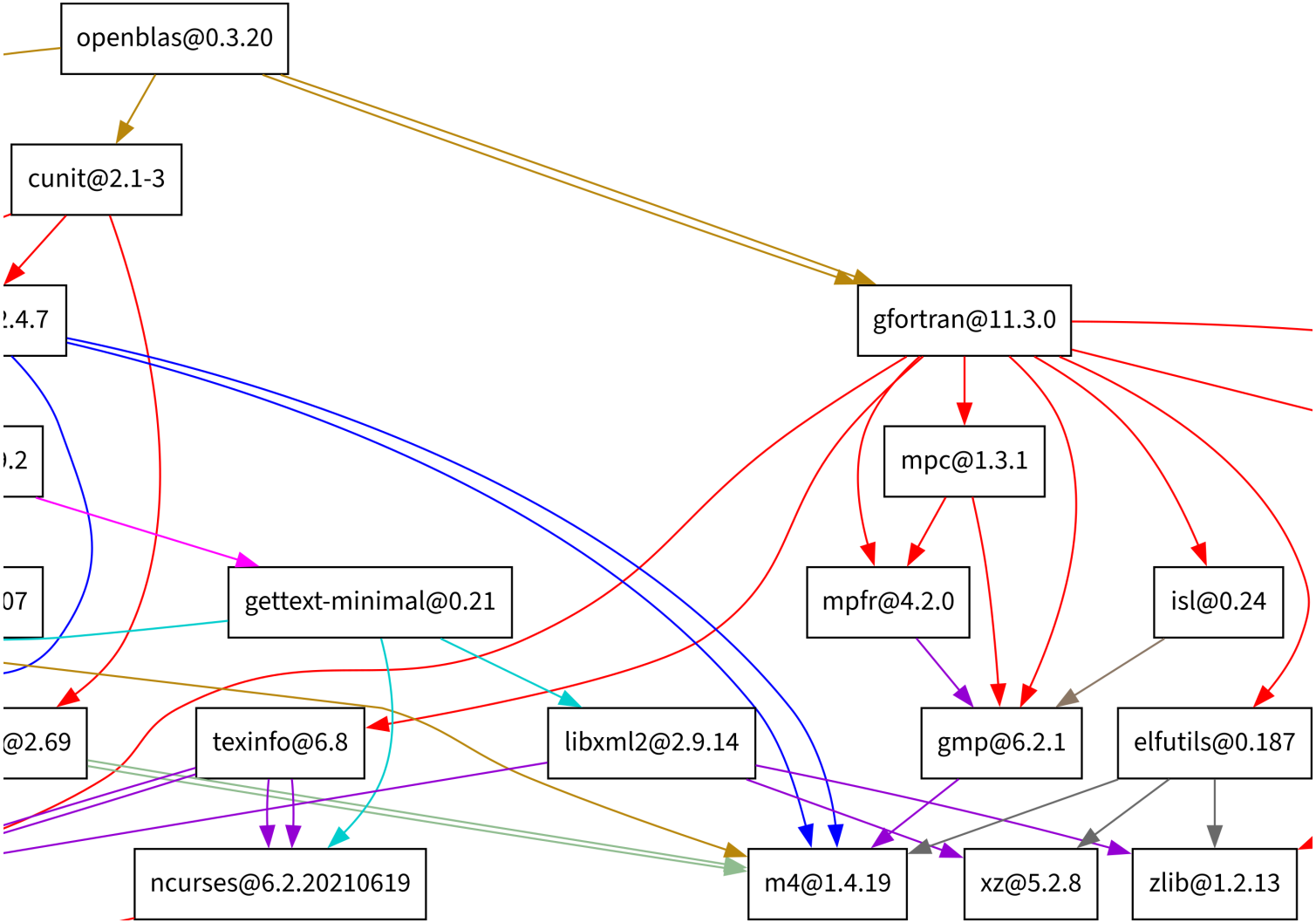
gives an archive with everything needed to Cubical Agda in Emacs, by extracting it anywhere!

One tool to rule them all: we can also output Docker images, RPMs, DEBs.

One tool to rule them all: we can also output Docker images, RPMs, DEBs.

This lets you run Guix-built software on e.g. clusters without Guix.

One can also inspect dependency graphs with `guix graph`.



Optimizations and Reproducibility

Adding architecture-specific optimizations might endanger reproducibility.

Optimizations and Reproducibility

Adding architecture-specific optimizations might endanger reproducibility.

In Guix, optimizations are part of the recipe, thus a change of optimizations builds a different version of the package, in a separate store path.

```
$ guix build --tune=skylake openblas
```

Optimizations and Reproducibility

Adding architecture-specific optimizations might endanger reproducibility.

In Guix, optimizations are part of the recipe, thus a change of optimizations builds a different version of the package, in a separate store path.

```
$ guix build --tune=skylake openblas
```

You can also replace some dependencies by optimized libraries with a simple package transformation:

```
$ guix build --with-input=gmp=my-faster-gmp openbla
```

Guix as a system

Guix has a complete set of Linux packages.

Next logical step: a Linux distribution!

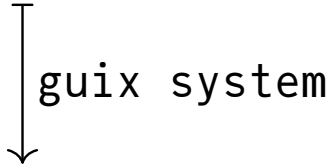
Guix as a system

Guix has a complete set of Linux packages.

Next logical step: a Linux distribution!

Guix System takes the functional POV to a whole new level: *declarative system configuration*.

```
(operating-system
  (host-name "komputilo")
  (timezone "Europe/Berlin")
  (locale "en_US.utf8")
  (bootloader ...)
  (file-systems ...)
  (users ...)
  (packages (cons screen %base-packages))
  (services ...))
```



system instantiation

Guix System can be used to configure:

- servers;
- personal machines;
- VMs;
- ISO images.

Guix System can be used to configure:

- servers;
- personal machines;
- VMs;
- ISO images.

```
$ guix system image -f qcow2 config.scm
```

produces a QCOW2 image of some configuration!

Guix System can be used to configure:

- servers;
- personal machines;
- VMs;
- ISO images.

```
$ guix system image -f qcow2 config.scm
```

produces a QCOW2 image of some configuration!

While this might be interesting for cluster administrators, once you get enamored with Guix you'll want to try it out!

Extensibility

Guix itself is written in Scheme, a Lisp dialect.

Everything is accessible and extensible by your own scripts!

Extensibility

Guix itself is written in Scheme, a Lisp dialect.

Everything is accessible and extensible by your own scripts!

Tuning packages to your specific needs can go from a simple

```
$ guix install --with-branch=guile=main cuirass
```

Extensibility

Guix itself is written in Scheme, a Lisp dialect.

Everything is accessible and extensible by your own scripts!

Tuning packages to your specific needs can go from a simple

```
$ guix install --with-branch=guile=main cuirass
```

to writing a new package definition

```
(package/inherit flameshot
  (arguments
   (substitute-keyword-arguments (package-arguments flameshot)
     ((#:configure-flags flags #~('()))
      #~(cons* "-DUSE_WAYLAND_CLIPBOARD=ON"
               #$@flags))))
  (inputs
   (modify-inputs (package-inputs flameshot)
     (prepend qtwayland-5
              (@ (gnu packages kde-frameworks) kguiaddons)))))
```

Need a lot of packages that are not yet in Guix? Writing all of their definitions could take time...

Need a lot of packages that are not yet in Guix? Writing all of their definitions could take time...

We have automatic importers!

```
$ guix import pypi b4
```

```
(package
  (name "python-b4")
  (version "0.12.4")
  (source
   (origin
     (method url-fetch)
     (uri (pypi-uri "b4" version))
     (sha256
      (base32 "03gxjnch08kzi33kqarr9a43pmzqaykk69kb09pdsk3dv2v8nycz"))))
  (build-system pyproject-build-system)
  (propagated-inputs (list python-dkimpy python-dnspython
                           python-git-filter-repo python-patatt
                           python-requests))
  (home-page "https://git.kernel.org/pub/scm/utils/b4/b4.git/tree/README.rst")
  (synopsis "A tool to work with public-inbox and patch archives")
  (description
   "This package provides a tool to work with public-inbox and patch archives")
  (license #f))
```

The chicken-and-egg problem

Guix also strives to be bootstrappable: we should be able to build compilers from source!

Example: Rust

The chicken-and-egg problem

Guix also strives to be bootstrappable: we should be able to build compilers from source!

Example: Rust

Officially,

$$\cdots \to \text{Rust} \to \text{Rust} \to \text{Rust} \to \cdots$$

The chicken-and-egg problem

Guix also strives to be bootstrappable: we should be able to build compilers from source!

Example: Rust

Officially,

$$\cdots \to \text{Rust} \to \text{Rust} \to \text{Rust} \to \cdots$$

We need to break the cycle! We use `mrustc` for an old version of Rust.

Guix now features the full-source bootstrap for the x86_64 architecture, which is the same problem but for the base C compiler.

See Janneke's blog post *The Full-Source Bootstrap: Building from source all the way down*, as well as the Bootstrappable Builds project!

Guix is also an excellent manual, available both in HTML and Info formats, and cozy CLI!

Guix is also an excellent manual, available both in HTML and Info formats, and cozy CLI!

Last but not least, Guix is also an incredibly nice and welcoming community, which you can meet over on IRC, over mails or here (for those who could attend)!

Thank you for your attention. Any questions?

Guix