Reproductibilité des environnements logiciels
une solution avec GNU Guix ?

Simon Tournier

Inserm US53 - UAR CNRS 2030
simon.tournier@u-paris.fr

Ateliers FCC, 21 juin 2022

https://hpc.guix.info

Université Paris Cité

Pourquoi j'en suis venu à GNU Guix

≈ 2010 Thésard

Développement d'1-2 outils utilisant un gestionnaire de paquets classique

(Simulation numérique `C` et `Fortran` avec Debian / Ubuntu / `apt`)

≈ 2014 Post-doc

Développement de 2-3 outils utilisant un gestionnaire de paquets sans droit administrateur

(Simulation numérique `Python` et `C++` avec `conda`)

2016 Ingénieur. de Recherche

- Administration d'un *cluster* (`modulefiles`)
- Utilisation de 10+ outils pour un même projet

(Analyse « bioinformatique »)

Question : **pourquoi cela fonctionne-t-il pour Alice et pas pour Bob ? Et vice-versa.**

Le monde est *open*, n'est-ce pas ?

- *open* journal
- *open* data
- *open* source
- *open* science
- *open* etc.

Quel est le problème de reproductibilité dans un contexte scientifique ?
(même si tout devient *open*)

« ordinateur » : traitement automatique de données

Le monde est *open*, n'est-ce pas ?

- *open* journal
- *open* data
- *open* source
- *open* science
- *open* etc.

Quel est le problème de reproductibilité dans un contexte scientifique ?
(même si tout devient *open*)

« ordinateur » : traitement automatique de données $\Longrightarrow$ environnement computationnel

Quel contrôle de l'environnement computationnel ?

Exemple d'un calcul

Listing 1 – Fonction J_0 de Bessel

```
#include <stdio.h>
#include <math.h>

int main(){
  printf("%E\n", j0f(0x1.33d152p+1f));
}
```

Alice	voit :	5.643440E-08
Carole	voit :	5.963430E-08

Pourquoi ? Le code est disponible pourtant.

Établir si le différence est significative ou non est laissé à l'expertise des scientifiques du domaine.

Quelques questions sur ce calcul

```c
#include <stdio.h>
#include <math.h>

int main(){
  printf("%E\n", j0f(0x1.33d152p+1f));
}
```

Alice	voit :	5.643440E-08
Carole	voit :	5.963430E-08

- Quel compilateur ?
- Quelles bibliothèques (`<math.h>`) ?
- Quelles versions ?
- Quelles options de compilation ?

En d'autres termes

- Quelles sont les sources des outils ?
- Quelles sont les outils requis pour la construction ?
- Quelles sont les outils requis pour l'exécution ?
- Comment chaque outil est-il produit ?

Répondre à ces questions signifie **contrôler la variabilité** de l'environnement computationnel

Comment capturer ces informations ?

(Réponse usuelle :Gestionnaire de paquets (Conda, APT, Brew, ...) ; *Modulefiles* ; Conteneur ; etc.)

Options de constructions (compilation)

```
#include <stdio.h>
#include <math.h>

int main(){
  printf("%E\n", j0f(0x1.33d152p+1f));
}
```

```
alice@laptop$   gcc bessel.c                     && ./a.out
                5.643440E-08
carole@desktop$ gcc bessel.c -lm -fno-builtin    && ./a.out
                5.963430E-08
```

(Ah, sacré *constant folding*)

Alice et Carole ont fait leur calcul dans deux environnements computationnels différents

Enjeu de la reproductibilité en recherche

analyse post-mortem du bétail

D'un point de vue la « méthode scientifique » :

Tout l'enjeu est le **contrôle de la variabilité**

D'un point de vue du « savoir scientifique » (caractère universel) :

- Un observateur indépendant doit être capable d'observer le même résultat.
- L'observation doit être pérenne (dans une certaine mesure).

Dans un monde où (presque) tout est donnée numérique (*data*),

comment refaire plus tard et là-bas ce qui a été fait aujourd'hui et ici ?

(sous entendu avec un « ordinateur »)

Ce que nous allons aborder

- Comment refaire demain là-bas ce que l'on a fait hier ici ?
- Quelle granularité sur la transparence ?

1. Introduction
   - Pourquoi seriez-vous intéressé par Guix ?
2. Gestion de paquets
3. Création d'images
4. Une histoire de versions
   - Version ? Graphe ?
   - Définir un paquet
   - Changer d'état
   - Préservation
5. Au final. . .

Pour fixer les idées

Logiciel	code source ou programme *binaire* associé
Paquet	recette pour configurer, construire, installer un logiciel
Dépendance	autre paquet nécessaire
Gestionnaire de paquets	automatisation du processus traitant la recette du paquet (et ses dépendances)
Environnement computationnel	pile de tous les logiciels nécessaires pour la configuration, construction et installation d'une collection de logiciels

Comment Alice et ses collaborateurs peuvent-ils obtenir le même environnement pour *calculer* avec Python et Numpy ?

Prés. avec un biais issu d'un environnement plus « scientifique » et moins « ASR »
mais **Guix s'adapte à tous les cas d'usage** (ou presque)

Scenarii

- Alice utilise `python@3.9` et `numpy@1.20.3`

```
$ sudo apt install python python-numpy
```

- Carole **collabore** avec Alice... mais utilise `python3.8` et `numpy@1.16.5` pour un autre projet

```
$ apt-cache madison python-numpy
python-numpy | 1:1.16.5-2ubuntu7 | ...
```

- Charlie **mets à jour** son système et **tout est cassé**

```
$ sudo apt upgrade
The following packages have unmet dependencies:
E: Broken packages
```

- Bob utilise les **mêmes versions** qu'Alice mais n'a **pas le même résultat**
- Dan essaie de **rejouer plus tard** le scénario d'Alice mais rencontre l'enfer des dépendances

Repeatability in Computer Science (lien)

Solution(s)

1. gestionnaire de paquets : APT (Debian/Ubuntu), YUM (RedHat), etc.
2. gestionnaire d'environnements : Modulefiles, Conda, etc.
3. conteneur : Docker, Singularity

Guix = #1 + #2 + #3

APT, Yum Difficile de faire coexister plusieurs versions ou revenir en arrière ?

Modulefiles Comment sont-ils maintenus ? (qui les utilise sur son *laptop* ?)

Conda Quelle granularité sur la transparence ? (qui sait comment a été produit PyTorch dans `pip install torch` ? (lien))

Docker Dockerfile basé sur APT, YUM etc.

```
RUN apt-get update && apt-get install
```

Guix est gestionnaire d'environnements sous *stéroïde*

un **gestionnaire de paquets**	(comme APT, Yum, etc.)
transactionnel et déclaratif	(revenir en arrière, versions concurrentes)
qui produit des ***packs* distribuables**	(conteneur Docker ou Singularity)
qui génèrent des ***machines virtuelles* isolées**	(*à la* Ansible ou Packer)
sur lequel on construit une distribution Linux	(nous n'en parlerons pas)
...et aussi une bibliothèque Scheme...	(nous n'en parlerons pas, non plus)

Cette petite présentation est un rapide coup de projecteur sur :

la gestion de paquets *fonctionnelle* $\Rightarrow$ reproductibilité

Ce que nous présentons fonctionne sur n'importe quelle distribution Linux

(Facile à essayer...)

distro externe

Guix s'installe sur **n'importe quelle distribution** Linux récente.

Il faut les droits administrateur (`root`) pour l'installation.

```
$ cd /tmp
$ wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
$ chmod +x guix-install.sh
$ sudo ./guix-install.sh
```

(Quelques réglages supplémentaires, voir le manuel)

Pour commencer :

```
$ guix help
```

Comment faire la gestion de paquets ?

(Exemple : Alice sans privilège particulier)

- Démos avec des outils du « *monde scientifique* »
- Guix représente 20k+ paquets (constante progression)
- Cela serait identique pour d'autres outils comme Nginx, LDAP, SSH, etc.

Commandes basiques

Exemple : outils scientifiques classiques en Python

(`demo/getting-started`)

(voir vidéo JRES)

Commandes basiques : Résumé

```
guix search dynamically-typed programming language # 1.
guix show    python                                # 2.
guix install python                                # 3.
guix install python-ipython python-numpy           # 4.
guix remove  python-ipython                        # 5.
guix install python-matplotlib python-scipy        # 6.
```

alias de `guix package`, p. ex. `guix package --install`

Transactionnel

```
guix package --install python                                    # 3.
guix package --install python-ipython python-numpy               # 4.
guix package -r python-ipython -i python-matplotlib python-scipy # 5. & 6.
```

Guix, un gestionnaire de paquets comme les autres ?

- Interface *ligne de commande* comme les autres gestionnaires de paquets
- Installation/suppression sans privilège particulier
- Transactionnel (= pas d'état « *cassé* »)
- *Substituts* binaires (téléchargement d'éléments pré-construits)

Trois fonctionnalités puissantes :

- Les *profils* et leur composition
- Gestion déclarative
- Environnement isolé à la volée

Le *profil* par défaut

Les commandes d'installation précédentes finissent avec le conseil :

```
hint: Consider setting the necessary environment variables by running:

     GUIX_PROFILE="/home/alice/.guix-profile"
     . "$GUIX_PROFILE/etc/profile"

Alternately, see `guix package --search-paths -p "$HOME/.guix-profile"'.
```

`$HOME/.guix-profile` est le *profil* par défaut

Mais qu'est-ce un *profil* ?

Filesystem Hierarchy Standard (FHS) = norme de la hiérarchie des systèmes de fichiers

(définit l'arborescence et contenu des répertoires systèmes pour les systèmes Unix)

```
usr
|-- bin        Binaires exécutables
|-- etc        Fichiers de configuration
|-- include    Entêtes des bibliothèques partagées
|-- lib        Bibliothèques partagées
|-- share      Documentation entre autres
```

`ls $HOME/.guix-profile`

Un *profil* est un répertoire contenant les paquets installés

Exemples de fonctionnalités des *profils*

- Historique des paquets installés / supprimés (`--list-generations`)
- Retour en arrière (`--roll-back` or `--switch-generations`)

(`demo/generations`)

- Profils indépendants
- Contrôle fin des variables d'environnement (`--search-paths`)
- Composition

(`demo/multi-profiles`)

Profil, en résumé

Les paquets sont installés dans un *profil* qui est :

- un lien symbolique,
- pointant vers un élément du dépôt (*store*),
- et les éléments ont une structure hiérarchique type FHS (comme `/usr/`).

On peut créer autant de profils que l'on souhaite

Toutes les options de `guix package` s'appliquent à n'importe quel profil

```
guix install python python-numpy --profile=outils-python
```

Profil, en résumé

Les paquets sont installés dans un *profil* qui est :

- un lien symbolique,
- pointant vers un élément du dépôt (*store*),

Dépôt (*store*)

- Monté à `/gnu/store`
- Dédupliqué

- et les éléments ont une structure hiérarchique type FHS (comme `/usr/`).

On peut créer autant de profils que l'on souhaite

Toutes les options de `guix package` s'appliquent à n'importe quel profil

```
guix install python python-numpy --profile=outils-python
```

Gestion déclarative

déclaratif = fichier de configuration

Un fichier `some-python.scm` peut contenir cette déclaration :

```
(specifications->manifest
 (list
  "python"
  "python-matplotlib"
  "python-numpy"
  "python-scipy"))
```

```
guix package --manifest=some-python.scm
```

équivalent à

```
guix install python python-matplotlib python-numpy python-scipy
```

Gestion *déclarative* : remarques

Version ? Nous le verrons dans la suite

Langage ? *Domain-Specific Language* (DSL) basé sur Scheme (« langage fonctionnel Lisp »)

- `(Oui (quand (= Lisp parenthèses) (baroque)))`
- Mais **continuum** :
  1. configuration (`manifest`)
  2. définition des paquets (ou services)
  3. extension
  4. le cœur est écrit aussi en Scheme

Guix est **adaptable** à ses besoins

Déclaratif vs Impératif (et non pas Donnée inerte vs Programme)

Programmation déclarative = programmation fonctionnelle ou descriptive (LaTeX) ou logique (Prolog)

Gestion déclarative : exemple de transformation (machine de Goldberg :-) (lien))

```
(define python "python")

(specifications->manifest
 (append
  (list python)
  (map (lambda (pkg)
         (string-append python "-" pkg))
       (list
        "matplotlib"
        "numpy"
        "scipy"))))
```

Guix DSL, *variables*, Scheme et chaîne de caractères.

Transformations de paquet : survol

Comment utiliser GCC@7 pour compiler le paquet `python` ?

Un paquet = recette pour configurer, construire, installer un logiciel

(`./configure && make && make install`)

La recette définit :

- un code source et potentiellement des modifications *ad-hoc* (`patch`)
- des outils de construction (compilateurs, moteur de production etc., p. ex. `gcc`, `cmake`)
- des dépendances

Une transformation permet de les réécrire

Transformations : ligne de commande

```
guix package --help-transformations
```

```
--with-source       use SOURCE when building the corresponding package
--with-branch       build PACKAGE from the latest commit of BRANCH
--with-commit       build PACKAGE from COMMIT
--with-git--url     build PACKAGE from the repository at URL
--with-patch        add FILE to the list of patches of PACKAGE
--with-latest       use the latest upstream release of PACKAGE
--with-c-toolchain  build PACKAGE and its dependents with TOOLCHAIN
--with-debug-info   build PACKAGE and preserve its debug info
--without-tests     build PACKAGE without running its tests
--with-input        replace dependency PACKAGE by REPLACEMENT
--with-graft        graft REPLACEMENT on packages that refer to PACKAGE
```

Transformations via fichier manifeste

```
(use-modules (guix transformations))

(define transform
  (options->transformation
   '((with-c-toolchain . "python=gcc-toolchain@7"))))

(packages->manifest
 (map (compose transform specification->package)
      (list
       "python"
       "python-matplotlib"
       "python-numpy"
       "python-scipy")))
```

Profils temporaires

Exemple : ajouter temporairement `IPython`

(`demo/shell-ipython`)

Étendre temporairement un *profil*

```
guix shell -m manifest.scm
guix shell -m manifest.scm python-ipython -- ipython3
```

- `--pure` : réinitialise des variables d'environnement existantes
- `--container` : lance un conteneur isolé
- `--development` : inclus les dépendances du paquet

```
guix shell -m some-python.scm python-ipython             # 1.
guix shell -m some-python.scm python-ipython --pure      # 2.
guix shell -m some-python.scm python-ipython --container # 3.
```

Bonus : `guix shell emacs git git:send-email --development guix`

À ce stade, il semble naturel de vouloir partager ou construire des conteneurs isolés.

Comment créer des images ?

(Exemple : Alice sans privilège particulier)

Comment capturer un environnement ? Conteneur

Conteneur = smoothie :-)

- Comment est construit le conteneur ? Dockerfile ?
- Comment sont construits les binaires inclus dans le conteneur ?

```
FROM amd64/debian:stretch
RUN apt-get update && apt-get install git make curl gcc g++ ...
RUN curl -L -O https://... && ... && make -j 4 && ...
RUN git clone https://... && ... && make ... /usr/local/lib/libopenblas.a ...
```

source (lien) : Gmsh A three-dimensional finite element mesh generator (lien)

Avec un Dockerfile au temps t, comment regénérer l'image au temps t' ?

Qu'est-ce qu'un *pack* ?

pack = collection de paquets dans un format d'archive

Quel est le but d'un *pack* ?

- Alice distribue « tout » à Carole,
- Carole n'a pas installé Guix mais aura l'exact même environnement.

Qu'est-ce qu'un format d'archive ?

- `tar` (*tarballs*)
- Docker
- Singularity
- paquet binaire Debian `.deb`

Qu'est-ce que « tout » ?

Carole a besoin de la *clôture transitive* (= toutes les dépendances)

```
$ guix size python-numpy --sort=closure
store item                total     self
python-numpy-1.20.3       301.5     23.6   7.8%
...
python-3.9.9              155.3     63.7  21.1%
openblas-0.3.18           152.8     40.0  13.3%
...
total: 301.5 MiB
```

`guix pack` permet de créer cette archive contenant « tout »

Création d'un *pack* pour le distribuer

- Alice construit un *pack* au format Docker

```
guix pack --format=docker -m manifest.scm
```

puis distribue ce conteneur Docker (via un *registry* ou autre).

- Carole n'utilise pas (encore ?) Guix

```
$ docker run -ti projet-alice python3
Python 3.9.9 (main, Jan  1 1970, 00:00:01)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

et utilise l'exact même environnement computationnel qu'Alice.

Au final, `guix pack`, c'est...

Agnostique sur le format du « conteneur »

- `tar` (*tarballs*)
- Docker
- Singularity
- paquet binaire Debian `.deb`

- archives repositionnables
- sans `Dockerfile`
- via `squashfs`
- sans `debian/rule` (expérimental)

Adaptable aux cas d'usage

Création d'une image avec `guix system` : un monde de services

`guix system` permet une <u>**configuration déclarative** d'un *système*</u>

- `guix system search` pour trouver les services disponibles
- `guix system image` pour construire une image de type :
  - qcow2
  - docker
  - iso9660, uncompressed-iso9660, efi-raw, raw-with-offset
  - rock64-raw, pinebook-pro-raw, pine64-raw, novena-raw
  - hurd-raw, hurd-qcow2
- `guix system vm` pour construire une machine virtuelle (VM)

  (la VM partage son dépôt avec le système hôte)

Machines virtuelles avec Guix

- Gestion déclarative
- Réutilisation via des patrons (*template*)

Très bien tout cela, mais en quoi est-ce reproductible ?

Parlons de versions !

(Exemple : Carole collabore avec Alice)

Quelle est ma version de Guix ?

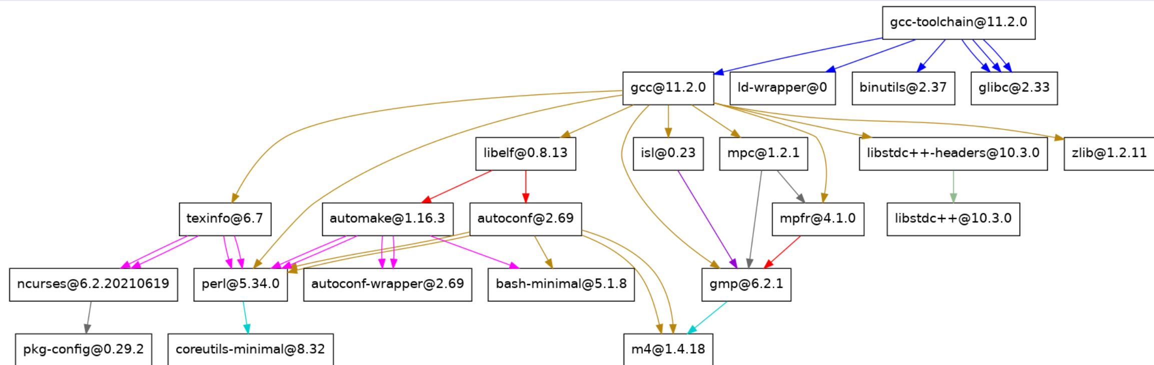
```
$ guix describe
Generation 76 Apr 25 2022 12:44:37 (current)
  guix eb34ff1
    repository URL: https://git.savannah.gnu.org/git/guix.git
    branch: master
    commit: eb34ff16cc9038880e87e1a58a93331fca37ad92

$ guix --version
guix (GNU Guix) eb34ff16cc9038880e87e1a58a93331fca37ad92
```

Un état fixe toute la collection des paquets et de Guix lui-même

(Un état peut contenir plusieurs canaux (*channel* = dépôt Git),
avec des URL, branches ou commits divers et variés)

Alice dit « GCC à la version 11.2.0 »



Est-ce la même version de GCC si `mpfr` est à la version 4.0 ?

Graphe complet : 43 ou 104 ou 125 ou 218 nœuds
(suivant ce que l'on considère comme graine binaire du *bootstrap*)

État = Graphe Acyclique Dirigé (*DAG*)

```
guix graph --max-depth=6 python | dot -Tpng > graph-python.png
```


Graphe complet : Python = 137 nœuds, Numpy = 189, Matplotlib = 915, Scipy = 1439 nœuds

Révision = un graphe spécifique

une version = un graphe

```
$ guix describe
Generation 76 Apr 25 2022 12:44:37 (current)
  guix eb34ff1
    repository URL: https://git.savannah.gnu.org/git/guix.git
    branch: master
    commit: eb34ff16cc9038880e87e1a58a93331fca37ad92
```

La révision `eb34ff1` capture **tout** le graphe

- Alice dit « j'ai utilisé Guix à la révision `eb34ff1` »
- Carole connaît toutes les informations pour reproduire le même environnement

Définition d'un paquet (nœud du graphe)

```
(define python
  (package
    (name "python")
    (version "3.9.9")
    (source ... )
    (build-system gnu-build-system)
    (arguments ... )
    (inputs (list bzip2 expat gdbm libffi sqlite
                  openssl readline zlib tcl tk))))
```

- Chaque inputs est une définition similaire (récursion → graphe)
- Il n'y a pas de cycle (bzip2 ou ses inputs ne peuvent pas utiliser python)

(Quel commencement du graphe ? Problème du *bootstrap* dont nous ne parlerons pas ici)

Gestionnaire de paquets = gestionnaire de graphe

Comment capturer les informations du transparent 4 ?

- Quelles sont les sources des outils ? `source`
- Quelles sont les outils requis pour la construction ?
- Quelles sont les outils requis pour l'exécution ?

  } `inputs`, `propagated-`, `native-inputs`
- Comment chaque outil est-il produit ? `build-system`, `arguments`

```
(package                        ; definition du noeud python
  (name "python")
  (version "3.9.9")
  (source ... )                           ; -> Gitlab, etc.
  (build-system gnu-build-system) ; ./configure; make; install
  (arguments ... )                        ; options de production
  (inputs (list ...)))          ; liste d'autres noeuds -> graphe (DAG)
```

Concrètement, en pratique ?

une version = un graphe

Mise à jour de Guix

`guix pull`

(création d'une nouvelle *génération* interne)

- Il est possible de spécifier un *état* :

```
  alice@laptop$  guix describe --format=channels > alice-laptop.scm
carole@desktop$  guix pull --channels=alice-laptop.scm
```

- Il est possible de se placer **temporairement** dans un état spécifique pour exécuter une commande (comme créer un profil)

```
guix time-machine --commit=c61df1792c -- install python -p python/autres
```

Collaboration

Alice décrit son environnement :

- la révision (Guix lui-même et aussi potentiellement les autres canaux) :

```
guix describe -f channels > alice.scm
```

- la liste des paquets avec le fichier `paquets.scm`

génère son environnement avec, e.g.,

```
guix shell -m paquets.scm
```

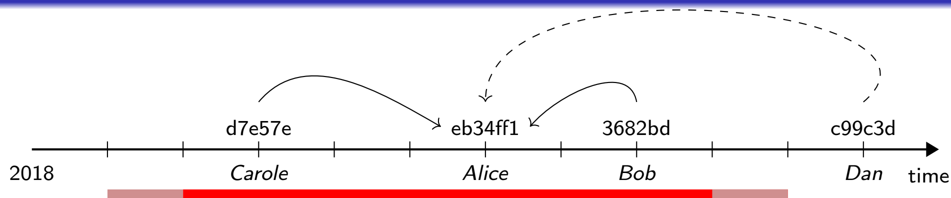
et **partage ses deux fichiers**.

Carole génère le même environnement **à partir des deux fichiers** d'Alice,

```
guix time-machine -C alice.scm -- shell -m paquets.scm
```

Dan peut donc aussi avoir le même environnement qu'Alice et Carole.

Reproductibilité en arrière, en avant : `guix time-machine`



Pour être reproductible dans le temps, il faut :

- Une préservation de **tous** les codes source (≈ 75% archivés (lien) dans Software Heritage (lien))
- Une *backward* compatibilité du noyau Linux
- Une compatibilité du *hardware* (p. ex. CPU, disque dur (NVMe), etc.)

Quelle est la taille de la fenêtre temporelle avec les 3 conditions satisfaites ?

(À ma connaissance, le projet Guix réalise une expérimentation grandeur nature et quasi-unique depuis sa v1.0 en 2019)

Pourquoi les préserver ?

Parce que les **services en ligne parfois s'arrêtent**

- Google Code (lien) début 2016
- Alioth (Debian) en 2018 remplacé par Salsa
- Gna ! en 2017 après 13 années d'activité
- Gitourious en 2015 (le second plus populaire service d'hébergement pour Git en 2011)
- etc.

Pourquoi les préserver ?

Parce que les **services en ligne parfois s'arrêtent**

- Google Code (lien) début 2016
- Alioth (Debian) en 2018 remplacé par Salsa
- Gna ! en 2017 après 13 années d'activité
- Gitourious en 2015 (le second plus populaire service d'hébergement pour Git en 2011)
- etc.
- `gforge.inria.fr` par exemple Guix issue #42162 (lien)

```
Believe it or not, gforge.inria.fr was finally phased out on
Sept. 30th.  And believe it or not, despite all the work and all the
chat :-), we lost the source tarball of Scotch 6.1.1 for a short period
of time (I found a copy and uploaded it to berlin a couple of hours
ago).
```

Comment les préserver ?

Forge $\neq$ Archive

L'objectif d'une forge est de permettre à plusieurs développeurs de **participer ensemble au développement** d'un ou plusieurs logiciels, le plus souvent à travers le réseau Internet.

https://fr.wikipedia.org/wiki/Forge_(informatique)

L'archivage est un ensemble d'actions qui a pour but de garantir l'**accessibilité sur le long terme** d'informations (dossiers, documents, données) que l'on doit ou souhaite conserver pour des raisons juridiques

https://fr.wikipedia.org/wiki/Archivage

Software Heritage « *are building the universal software archive* » (lien)

Les services en ligne parfois s'arrêtent...

Pourquoi serait-il différent pour Software Heritage ?

Aucune garantie mais...

Software Heritage is an open, non-profit initiative unveiled in 2016 by Inria. It is supported by a broad panel of institutional and industry partners, in collaboration with UNESCO.

The long term goal is to collect all publicly available software in source code form together with its development history, replicate it massively to ensure its preservation, and share it with everyone who needs it.

- Supporté par des institutions
- Avec la mission d'archiver

État des lieux de Guix et Software Heritage (SWH)

Guix est capable de

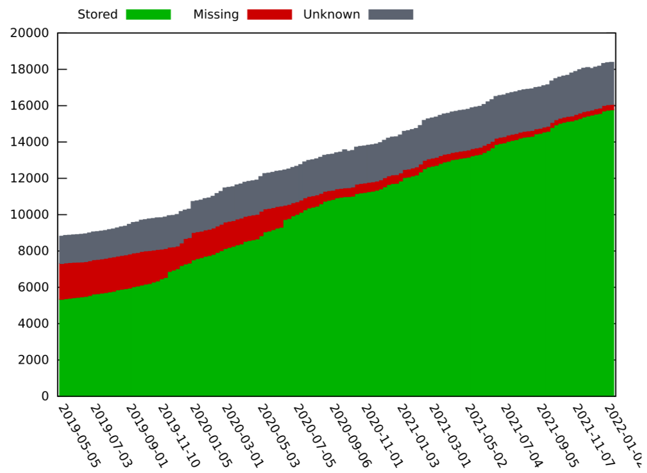
- sauvegarder (automatiquement) les sources dans SWH
- retrouver (automatiquement) les sources depuis SWH

Mais pas tous les types de sources !

- Des outils disponibles dans Guix pour les sources :
  - dépôt Git `git-fetch`
  - archives *tarballs* (compressées) `url-fetch`
- ...et manquants pour tout le reste (aide bienvenue :-))

Quelques stats : préservation de Guix

https://ngyro.com/pog-reports/latest/



Sauvegarder dans Software Heritage

- Git = 98.3%
- *tarballs* ≈ 70%

Oui, mais...

- Comment identifier mon code ?

  Identifiant intrinsèque vs extrinsèque (lien)

  tag (v1.2.3) vs hash (`eb34ff1`), quel hash ? etc.
- Comment s'assurer que tout le graphe est bien préservé ?
- Que se passe-t-il s'il manque les sources d'un seul nœud ?
- On n'a pas parlé du *boostrap* (les racines du graphes).
- etc.

Au final

. . .Guix est un *continuum* (sur n'importe quelle distribution Linux)

un **gestionnaire de paquets** déclaratif	`guix package`	(`-m` *manifest*)
temporairement étendu à la volée	`guix shell`	(`--container`)
maîtrisant exactement l'*état*	`guix time-machine`	(`-C` *channels*)
qui produit des ***packs* distribuables**	`guix pack`	(`-f docker`)
qui génèrent des ***machines virtuelles* isolées**	`guix sytem vm`	
et aussi une bibliothèque Scheme	`guix repl`	(*extensions*)
(la distribution Linux elle-même	`config.scm`	(Guix System))

Guix permet un contrôle fin du graphe de configuration sous-jacent

`guix time-machine -C channels.scm --` *commande options* `une-config.scm`

`une-config.scm` est **reproductible** d'une machine à l'autre et dans le temps

Scenarii

- Alice utilise `python@3.9` et `numpy@1.20.3`

```
$ guix install python python-numpy
```

- Carole **collabore** avec Alice... mais utilise d'autres outils pour un autre projet

```
$ guix time-machine -C version-alice.scm \
       -- install -m outils-alice.scm -p projet/alice
```

- Charlie **mets à jour** son système et **tout est cassé**

```
$ guix pull --roll-back
```

- Bob utilise les **mêmes versions** qu'Alice mais n'a **pas le même résultat**

```
error: You found a bug
```

- Dan essaie de **rejouer plus tard** le scénario d'Alice...
  (voir ligne commande Carole)
  ...ça dépend de la date du scénario ;-)

On s'habitue vite...

- Fonctionne uniquement avec Linux.
- Environnement isolé implique une forte transparence,
  c.-à-d., difficile avec des parties propriétaires.
- Certaines commandes peuvent apparaître lentes (`pull`, `search`, etc.),
  ou retourner des erreurs obscures.
- Les premiers pas requièrent un peu de patience,
  et d'accepter que ce n'est pas *comme d'habitude*.

La communauté est très accueillante et toujours disponible pour aider

En production

Grid'5000		828-nodes	(12,000+ cores, 31 clusters)	(France)
GliCID (CCIPL)	Nantes	392-nodes	(7500+ cores)	(France)
PlaFrIM Inria	Bordeaux	120-nodes	(3000+ cores)	(France)
GriCAD	Grenoble	72-nodes	(1000+ cores)	(France)
Max Delbrück Center	Berlin	250-nodes	+ workstations	(Allemagne)
UMC	Utrecht	68-nodes	(1000+ cores)	(Pays-Bas)
UTHSC Pangenome		11-nodes	(264 cores)	(USA)

(le vôtre ?)

https://hpc.guix.info

JRES – Marseille, 2022 (PDF) (Vidéo)

comment refaire plus tard et là-bas ce qui a été fait aujourd'hui et ici ?

traçabilité et transparence

être capable d'étudier bogue-à-bogue

Guix devrait s'occuper de tout

```
guix time-machine -C channels.scm -- cmd -m manifest.scm
```

si on spécifie

« comment construire » `channels.scm`

« quoi construire » `manifest.scm`

Des questions ?

`guix-science@gnu.org`

`https://hpc.guix.info/events/2022/café-guix/`

Cette présentation voir là

`https://zimoun.gitlab.io/fcc-inrae/`

(Software Heritage id swh:1:rev:f722a7ce1a5a98c8b5c1619ba826e034b5090db6)