

Reproductibilité des environnements logiciels avec GNU Guix

Simon Tournier

JRES, 19 mai 2022

Résumé

Ce document s'attache à montrer quelques fonctionnalités de l'outil GNU Guix. Il correspond à un tutoriel court d'une durée de 1 heure, cependant les informations données ici peuvent prendre plus de temps.

GNU Guix est un gestionnaire de paquets transactionnel et déclaratif. Il implémente une discipline de gestion de paquet fonctionnelle. Autrement dit, le processus de construction et d'installation des paquets est vu comme une fonction dans le sens mathématique du terme : cette fonction a des entrées (comme des scripts de construction, un compilateur, des bibliothèques ou dépendances) et renvoie une construction (comme un programme compilé). En tant que fonction pure, son résultat ne dépend que de ses entrées ; par exemple, il ne peut pas faire référence à des logiciels ou des scripts qui n'ont pas été explicitement passés en entrée. Par conséquent, une fonction de construction produit toujours le même résultat quand on lui donne le même ensemble d'entrée. Les constructions sont dites reproductibles.

Basé sur ces fondations, Guix est un gestionnaire d'environnements logiciels : il peut utiliser les paquets qu'il connaît pour créer des environnements sous différentes formes, temporaires ou « permanentes ». Un environnement (collection de paquets) peut être déclaré par un manifeste pour générer un « profil », chaque profil ayant son propre historique permettant de revenir en arrière ou en avant. Un environnement temporaire peut aussi être créé à la volée, et optionnellement isolé (conteneur Linux), ou n'autorisant lecture, écriture ou réseau que spécifiquement. Aussi, ces environnements peuvent être empaquetés dans des images Docker, Singularity ou simplement des archives repositionnables. Pour finir, Guix permet, sur le même principe, la génération de « machines virtuelles » à partir d'une déclaration pouvant contenir un système d'exploitation. Une fois installé, Guix ne nécessite pas de droits spécifiques pour manipuler ses différents environnements.

Dans ce tutoriel, nous nous proposons d'illustrer quelques spécificités de Guix. Ce tutoriel fait echo à la présentation « Administration système reproductible avec GNU Guix » par Julien Lepiller au JRES 2019 (vidéo et article).

(Les termes en police **sans serif** sont des hyperliens.)

Ce tutoriel n'est qu'une introduction à l'outil Guix. Nous avons pris le parti d'une approche aussi didactique que possible. Ce document se veut une invitation et non pas un menu exhaustif.

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.

De cette expérience, la question qui se dégage est la continuité du déploiement de l'environnement computationnel : comment développer sur son ordinateur portable, faire la mise au point sur son ordinateur de bureau et calculer massivement sur la grappe de calcul (*cluster*) mutualisée sans perdre trop de temps sur des problèmes d'administration ?

Notre motivation dans l'utilisation de Guix est d'apporter une réponse aux questions :

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

Pour information :

- Conda est un gestionnaire d'environnement issu de la communauté Python et visant particulièrement le contexte de logiciels scientifiques.
- La commande **module** est un outil largement répandu sur les grappes de calcul qui permet, en jouant sur les variables d'environnement, de charger des outils spécifiques.

Nous précisons plus loin pourquoi nous ne les considérons pas satisfaisants.

1 Introduction

Dans cette section, nous cherchons à souligner le cadre des problèmes et des solutions existantes, ce qui permet, nous l'espérons d'avoir une idée où se positionne l'outil Guix.

1.1 Pourquoi seriez-vous intéressé par Guix ?

Comme nous allons parler de *logiciels*, de *paquets*, *dépendances*, puis de ce que fait un *gestionnaire de paquets* pour créer des environnements computationnels, voici quelques définitions (dans des sens très larges) qui permettent de fixer les idées.

The screenshot shows a presentation slide with a navigation bar at the top containing five items: 'Introduction' (selected), 'Gestion de paquets', 'Création d'images', 'Une histoire de versions', and 'Au final...'. Below the navigation bar is the title 'Pour fixer les idées'. The main content is a list of definitions:

- 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

Below the list are two callout boxes:

- Comment Alice et ses collaborateurs peuvent-ils obtenir le même environnement pour *calculer* avec Python et Numpy ?
- Tuto avec un biais issu d'un environnement plus « scientifique » et moins « ASR » **mais Guix s'adapte à tous les cas d'usage** (ou presque)

The footer of the slide contains 'S. Tournier', 'Reproductibilité des environnements logiciels avec GNU Guix', and '1 / 40'.

Les collaborateurs d'Alice sont Carole, Charlie et Bob. Tous travaillent sur des machines différentes avec des distributions GNU/Linux différentes. Dan n'a pas d'interaction avec Alice et a sous la main uniquement un document (article de recherche ou fichier de configuration).

- Qui n'a pas été dans la situation de Carole qui devait installer plusieurs versions d'un même logiciel ? Et ces versions peuvent entrer en conflit. En général, par défaut, une distribution Linux installe uniquement une seule version d'un logiciel avec toutes ses dépendances. Le problème est l'endroit où résident ces logiciels, par défaut le dossier nommé `/usr/`, et il ne peut pas y avoir deux exécutables, par exemple `python3`, à deux versions différentes, par exemple 3.9 et 3.10. Il faut donc un logiciel externe qui s'occupe de la gestion de différentes versions (et leurs dépendances).

Introduction ●○○○○○ Gestion de paquets ○○○○○○○○○○○○○○ Création d'images ○○○○○○○○○ Une histoire de versions ○○○○○ Au final... ○○○○○

Pourquoi seriez-vous intéressé par Guix ?

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 **met 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 \(lien\)](#)

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 2 / 40

- Qui n'a pas été dans la situation de Charlie la veille d'un évènement important et plus rien ne fonctionnait ? À ma connaissance, cette difficulté de mise à jour est une difficulté des administrateurs systèmes de machine partagée : une fois la mise à jour faite, s'il y a de la casse, pas de possibilités de revenir en arrière.
- Qui n'a pas été dans la situation de Bob « pourquoi cela ne fonctionne pas pour moi, et pourtant tout est pareil » ? Car peut-être que les versions `python` et `numpy` de Bob sont bien les mêmes que celles d'Alice, mais le problème est : est-ce bien le cas pour toutes les dépendances et toutes les dépendances de dépendances ?
- Qui n'a pas été dans la situation de Dan en lisant un article scientifique ou un tutoriel ou une documentation ? Le problème est qu'il est difficile de s'assurer que la liste des logiciels requis (ainsi que toutes les dépendances et leurs dépendances) sont à la bonne version sur deux systèmes différents à deux moments différents.

Pour résumer, les problèmes sont :

1. installer le logiciel ainsi que toutes les dépendances,
2. installer plusieurs versions d'un même logiciel,
3. capturer l'état complet du système.

Rappelons brièvement les termes des solutions :

- un gestionnaire de paquets automatise le processus pour configurer, construire et installer le logiciel ainsi que toutes les dépendances,
- un gestionnaire d'environnements autorise la coexistence de plusieurs versions d'un même logiciel, le plus souvent en jouant sur les variables d'environnement,

Introduction ○○●○○○	Gestion de paquets ○○○○○○○○○○○○○○○○	Création d'images ○○○○○○○○○○	Une histoire de versions ○○○○○	Au final... ○○○○○
------------------------	--	---------------------------------	-----------------------------------	----------------------

Pourquoi seriez-vous intéressé par Guix ?

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
```

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 3 / 40

- un conteneur fournit tous les binaires pour être transporté d'une machine à l'autre et donc fonctionne indépendamment de la configuration de la machine hôte.

La double difficulté majeure est :

- d'une part, un contrôle fin dans la production du binaire,
- d'autre part, la reproductibilité, au sens « refaire ailleurs » (autre configuration, autre moment).

Guix, en étant un gestionnaire d'environnements sous *stéroïde*, tente de répondre à cette double difficulté.

Convention. Nous notons `foo@1.2` pour le logiciel `foo` à la version 1.2. Usuellement, une ligne commençant par `$` (dollar) représente une commande *shell*, ainsi que par la suite toutes les lignes commençant par `guix`. La commande `sudo` signifie des droits administrateur.

Introduction 0000●000 Gestion de paquets 0000000000000000 Création d'images 0000000000 Une histoire de versions 000000 Au final... 000000

Pourquoi seriez-vous intéressé par Guix ?

Concrètement

Soit

- ▶ vous vous reconnaissez dans un scénario,
- ▶ vous n'êtes pas satisfait par une des solutions,
- ▶ vous êtes curieux d'un nouveau outil.

Dans les 3 cas, ce tutoriel illustre :

- ▶ Comment faire en pratique avec GNU Guix.
- ▶ Quel est le problème à traiter ? Et pourquoi c'est compliqué.

Guix est un gestionnaire d'environnements sous *stéroïde*

Ce tutoriel est complémentaire de la présentation de Julien Lepiller en 2019 (vidéo et article)

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 4 / 40

Introduction 0000●000 Gestion de paquets 0000000000000000 Création d'images 0000000000 Une histoire de versions 000000 Au final... 000000

Pourquoi seriez-vous intéressé par Guix ?

Stéroïde signifie. . .

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)

Ce tuto court est un coup de projecteur sur :

- ▶ la gestion de paquets *fonctionnelle*
- ▶ création de *machines virtuelles*

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

(Facile à essayer...)

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 5 / 40

1.2 Installation de GNU Guix

Il est important de noter en tout premier lieu le **manuel** (traduit en français) qui fournit le point d'entrée pour l'installation et la configuration. La version de référence est la **version anglaise**.

1.2.1 Installation recommandée

Nous recommandons d'installer Guix sur une distribution Linux, voir la section **Installation binaire** du manuel. En tant que `root` (avec les droits administrateur), il faut exécuter les commandes :

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

Attention. Nous vous recommandons de parcourir le script *shell* `guix-install.sh` et de ne pas l'exécuter aveuglément. Ce script fait principalement 2 choses :

1. télécharge, installe et configure le démon `guix-daemon`,
2. crée les différents dossiers nécessaires au bon fonctionnement.

Cette installation n'interfère en rien avec la distribution hôte. Pour « désinstaller » Guix, il suffit de supprimer les dossiers `/gnu`, `/var/guix`, `$HOME/{.cache,.config}/guix`, `/root/.config/guix` et les mécanismes de complétion de *shell* (p. ex. `/etc/bash_completion.d`).

Remarques. Pour une utilisation plus agréable, nous recommandons d'autoriser les substituts binaires (voir la section **Substituts**) lors de l'installation via le script. Nous recommandons aussi de configurer *Name Service Switch* (`nscd`) sur la distribution hôte (voir la section du manuel pour une explication du pourquoi).

Pour des constructions locales, il est aussi possible d'utiliser un mécanisme de déchargement (*offload*) sur une machine tierce. Nous renvoyons à la section du manuel **Utiliser le dispositif de déchargement**.

Pour finir, nous recommandons d'installer le paquet `glibc-locales`, dans le *profil* `root` ainsi que dans le *profil* utilisateurs (voir la section **Régionalisation**).

```
sudo guix install glibc-locales
guix install glibc-locales
```

et d'exporter la variable d'environnement

```
export GUIX_LOCPATH=$HOME/.guix-profile/lib/locale
```

ce qui évite des messages de *warning*.

1.2.2 Autres méthodes d'installation

Guix System, Guix dans une VM. Il nous apparaît plus aisé de commencer par installer Guix sur une distribution Linux et d'utiliser l'outil comme gestionnaire de paquets sous *stéroïde*. Mais les plus téméraires d'entre vous peuvent vouloir installer Guix comme système d'exploitation complet ou dans une machine virtuelle. Dans ce cas, le manuel fournit une section dédiée. Nous recommandons d'utiliser l'installateur graphique.

Par ailleurs, il est à noter que Guix n'a pas un modèle stable versus expérimental mais des révisions considérées comme *release*, intensivement testées mais ne recevant pas de correctifs. Actuellement, c'est l'étiquette **v1.3.0** sortie au mois de mai 2021. Il est possible de télécharger des images soit de la version dite « **standard** » (v1.3.0), soit des images générées à partir des derniers développements. Chacun a ses avantages et inconvénients :

avantage « standard » : bien testé,

inconvénient « standard » : potentiel manque de substituts binaires,

avantage « derniers développements » : inclus les correctifs depuis le dernier étiquetage,

inconvénient « derniers développements » : potentiellement instable ; ça peut fonctionner parfaitement comme cela peut être cassé.

Installation via le paquet Debian. Nous n'avons pas intensivement testé cette méthode d'installation, c'est pourquoi nous la ne recommanderions pas. Mais elle est peut-être d'intérêt pour certains d'entre vous. La version Debian Bullseye fournit le paquet Guix :

```
$ sudo apt install guix
```

Installation sur une grappe de calcul. Les grappes de calcul (*cluster*) ont souvent des configurations très spécifiques, nous renvoyons à cette **explications** pour les points essentiels. Sinon, nous recommandons de prendre contact avec la communauté via p. ex. guix-science@gnu.org.

1.2.3 Révision de Guix pour ce tutoriel

Ce tutoriel devrait être pouvoir être joué quelque soit la révision. Cependant, il est possible que les exemples fournis évoluent ; comme les versions des paquets Python ou autres. La commande, en tant qu'utilisateur sans privilège particulier,

```
$ guix pull --commit=eb34ff1
```

permet de se placer dans l'exact même révision que celle utilisée lors de l'écriture de ce document.

Remarques. Les fermes de substituts officielles n'étant pas des ressources illimitées, il est possible que les substituts ne soient plus disponibles dans quelques mois ou années après mai 2022. Le projet Guix s'efforce autant que possible d'assurer la conservation des substituts binaires qu'il produit et nous espérons que cette remarque soit caduque.

Dans le cas d'absence de substituts binaires, cela signifie que la commande précédente (`pull`) risque de prendre plus de temps, mais surtout que les exemples utilisant les piles logicielles risquent de compiler depuis les sources toutes les dépendances (binaires) manquantes de la dite pile. D'un côté, c'est pour cela que Guix est un outil puissant pour la reproductibilité.

D'un autre côté, si les substituts ne sont plus disponibles, alors il sera difficile de suivre ce tutoriel avec votre ordinateur portable et nous conseillons donc d'utiliser un déchargement (*offload*). Ou de saisir l'esprit général et d'utiliser vos propres exemples avec substituts.

2 Gestion de paquets

À partir de cette section, nous considérons une installation fonctionnelle de l'outil Guix sur une distribution Linux récente. Toutes les commandes seront exécutées dans un terminal en tant qu'utilisateur régulier (sans droit d'administration particulier). La révision de Guix utilisée est `eb34ff1`.

Nous recommandons la lecture de la section « Pour démarrer » du manuel.

2.1 Utilisation de `guix package`

Guix est avant tout un gestionnaire de paquets, c.-à-d. que Guix automatise le processus pour configurer, construire et installer le logiciel ainsi que toutes les dépendances. Les fonctionnalités de gestion de paquets sont données par la commande `guix package`.

The screenshot shows the Guix manual page for "Utilisation de guix package". At the top, there is a navigation bar with five items: "Introduction", "Gestion de paquets" (which is the current page), "Création d'images", "Une histoire de versions", and "Au final...". Below the navigation bar, the title "Commandes basiques : Résumé" is displayed. The main content area contains a list of basic commands:

```
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.
```

Below this list, it states: "alias de guix package, p. ex. guix package --install".

There is a red header "Transactionnel" followed by a list of transactional commands:

```
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.
```

At the bottom of the screenshot, there is a footer with the text "S. Tournier", "Reproductibilité des environnements logiciels avec GNU Guix", and "8 / 40".

Les commandes précédentes sont des alias de la commande `guix package`.

1. `guix search` ↔ `guix package --search=`
2. `guix show` ↔ `guix package --show=`
3. `guix install` ↔ `guix package --install=`
4. `guix remove` ↔ `guix package --remove=`

Pour de plus amples informations sur chacune des commandes, se référer au manuel (section `Invoquer guix package`). En quelques mots :

1. `guix search` : Recherche dans tous les paquets disponibles les termes `dynamically-typed programming language` dans les noms, synopsis et descriptions des paquets.

2. `guix show` : Affiche les informations relatives au paquet : nom, version, dépendances, synopsis, description, etc. Par convention si aucune version n'est donnée, il choisit la dernière.
3. `guix install` : Installe le paquet `python`, par convention si aucune version n'est donnée, il choisit la dernière.
4. Installe les paquets `python-ipython` `python-numpy`.
5. `guix remove` : Supprime l'installation du paquet `python-ipython`.
6. Installe les deux autres paquets `python-matplotlib` `python-scipy`.

Il est important de noter que les opérations d'installation / suppression sont transactionnelles. D'une part, une action (transaction) peut comporter plusieurs opérations. D'autre part, cette action (transaction) transforme un état dans un autre et ces états sont sauvegardés (voir *génération* et la Gestion de *profil*, section 2.2); en d'autres termes il est toujours possible de défaire une action (transaction).



- ▶ 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

Toutes ces commandes ressemblent très fortement à toutes celles d'autres gestionnaires de paquets, comme `apt` (Debian/Ubuntu) ou `yum` (RedHat). À la différence qu'il ne faut pas de droits d'administration particuliers pour installer des paquets.

Nous allons maintenant illustrer trois fonctionnalités qui font de Guix le couteau suisse des gestionnaires d'environnement. Les *profils* permettent de garder sous contrôle l'installation d'outils (paquets) potentiellement en conflit. La *gestion déclarative* permet de configurer les *profils* dans un style de programmation fonctionnelle. Pour finir, nous verrons la création à la volée d'un environnement computationnel et optionnellement isolé.

2.2 Gestion de *profil*

Un notion essentielle dans l'utilisation de Guix est la notion de *profil* : un répertoire contenant les paquets installés.



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

Il n'est pas nécessaire d'entrer dans ces détails pour utiliser Guix mais ce tutoriel nous apparaît opportun pour décortiquer.

Le profil par défaut est `$HOME/.guix-profile` et nous pouvons afficher les variables d'environnements de ce profil avec la commande

```
$ guix package --search-paths -p $HOME/.guix-profile
export PATH="/home/alice/.guix-profile/bin"
export GUIX_PYTHONPATH="/home/alice/.guix-profile/lib/python3.9/site-packages"
export GI_TYPELIB_PATH="/home/alice/.guix-profile/lib/girepository-1.0"
export XDG_DATA_DIRS="/home/alice/.guix-profile/share"
```

et ce profil est un lien symbolique

```
$ file $HOME/.guix-profile
/home/alice/.guix-profile: symbolic link to
                               /var/guix/profiles/per-user/alice/guix-profile
```

et vers quoi pointe le lien ?

```
$ readlink -f $HOME/.guix-profile
/gnu/store/jy24qk0vdgqvysnryldqn52b3fg03z10-profile
```

où `jy24qk...` ressemble à un condensat (*hash*) – il condense toutes les entrées (dépendances) pour la construction dudit profil. Le type du lien pointé est un dossier

```
$ file $(readlink -f ~/.guix-profile)
/gnu/store/jy24qk0vdgqvysnryldqn52b3fg03z10-profile: directory
```

et ce dossier contient

```
$ tree -L 1 $(readlink -f ~/.guix-profile)
/gnu/store/jy24qk0vdgqvysnryldqn52b3fg03z10-profile
|-- bin
|-- etc
|-- include
|-- lib
|-- manifest
'-- share
```

The screenshot shows a presentation slide with a dark blue header and a light blue body. The header contains navigation tabs: 'Introduction', 'Gestion de paquets', 'Création d'images', 'Une histoire de versions', and 'Au final...'. The main title is 'Mais qu'est-ce un profil?'. Below the title, the text reads: 'Filesystem Hierarchy Standard (FHS) = norme de la hiérarchie des systèmes de fichiers (définit l'arborescence et contenu des répertoires systèmes pour les systèmes Unix)'. A light green box contains a list of FHS directories and their purposes: 'usr' (Binaires exécutables), 'etc' (Fichiers de configuration), 'include' (Entêtes des bibliothèques partagées), 'lib' (Bibliothèques partagées), and 'share' (Documentation entre autres). Below this box is the command 'ls \$HOME/.guix-profile'. A light pink box at the bottom states 'Un profil est un répertoire contenant les paquets installés'. The footer of the slide includes 'S. Tournier', 'Reproductibilité des environnements logiciels avec GNU Guix', and '11 / 40'.

Conclusion

- Un *profil* contient les variables d'environnements ajustées.
- Un *profil* est un lien symbolique.
- Un *profil* pointe vers un dossier du dépôt (*store*).
- Un *profil* a la structure hiérarchique de FHS (comme */usr/*).

Sans entrer dans les détails, le dépôt (*store*) est monté à */gnu/store* et il est déduplicé, c.-à-d. qu'il pourrait être vu comme une forêt de liens symboliques (voir Fig. 1). L'exemple Fig. 1 utilise des paquets qui sont classiques dans le contexte bioinformatique. Cependant, on peut se poser la question des nombres 42 et 43. Examinons ceci.

```
$ ls -l /var/guix/profiles/per-user/alice/guix-profile*
/var/guix/profiles/per-user/alice/guix-profile -> guix-profile-5-link
```

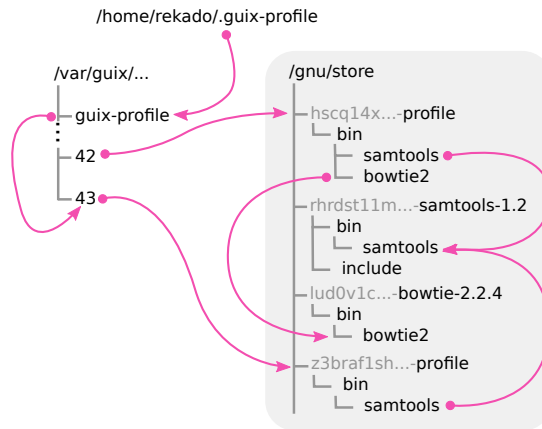


FIGURE 1 – Forêt de liens symboliques

```

/var/guix/profiles/per-user/alice/guix-profile-0-link -> /gnu/store/0fv...-profile
/var/guix/profiles/per-user/alice/guix-profile-1-link -> /gnu/store/vsr...-profile
/var/guix/profiles/per-user/alice/guix-profile-2-link -> /gnu/store/k3c...-profile
/var/guix/profiles/per-user/alice/guix-profile-3-link -> /gnu/store/dks...-profile
/var/guix/profiles/per-user/alice/guix-profile-4-link -> /gnu/store/33d...-profile
/var/guix/profiles/per-user/alice/guix-profile-5-link -> /gnu/store/jy2...-profile
    
```

Par conséquent, l'historique est en quelque sorte conservé et Guix donne la possibilité de changer vers quoi les liens pointent. Le profil lui-même est donc immuable.

Introduction ○○○○○○○	Gestion de paquets ○○○○●○○○○○○○○	Création d'images ○○○○○○○○○	Une histoire de versions ○○○○○	Au final... ○○○○○
-------------------------	-------------------------------------	--------------------------------	-----------------------------------	----------------------

Gestion de profil

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)

Une première fonctionnalité est la conservation des actions (installation ou suppression) sur un profil. Chaque action complète va créer une *génération* (état) – c’est en ce sens que Guix est transactionnel. Il est donc possible de revenir en arrière (*roll-back*) ou de basculer vers une autre génération (*switch-generation*). Par exemple,

```
$ guix package --list-generations
Generation 1   avril 21 2022 20:22:28
+ python      3.9.9   out    /gnu/store/sz7...-python-3.9.9

Generation 2   avril 21 2022 20:38:30
+ python-numpy 1.20.3 out    /gnu/store/s89...-python-numpy-1.20.3
+ python-ipython 7.27.0 out    /gnu/store/rj0...-python-ipython-7.27.0

Generation 3   avril 21 2022 20:41:38
- python-ipython 7.27.0 out    /gnu/store/rj0...-python-ipython-7.27.0

Generation 4   avril 21 2022 21:05:14
+ python-scipy 1.7.3   out    /gnu/store/x3x...-python-scipy-1.7.3
+ python-matplotlib 3.5.1 out    /gnu/store/inl...-python-matplotlib-3.5.1

Generation 5   avril 21 2022 21:19:34 (current)
+ tree        2.0.2   out    /gnu/store/b9z...-tree-2.0.2
```

où le symbole + représente un ajout de paquet, le symbole - une suppression pour chaque génération et `guix package --list-installed` fournit la liste complète des paquets de l’état courant.

Il est possible de supprimer une (ou plusieurs) génération avec l’option `--delete-generations`. Nous renvoyons à la section du manuel sur le ramasse-miette qui permet une gestion de ce que contient le dépôt (*store*).

Pour finir, autant de profils que l’on souhaite peuvent être créés. D’une part, chaque profil est indépendant et d’autre part, ils peuvent être composés. En d’autres termes

```
guix install python --profile=interpreteur
guix install python-numpy --profile=bibliotheque
eval $(guix package --search-paths=prefix -p interpreteur -p bibliotheque)
```

et l’environnement résultant contiendra `python` et `python-numpy`. D’autre part, l’ajout des variables d’environnement (`search-paths`) est contrôlable avec la valeur de l’argument `prefix`, `suffix` ou `exact` ; les variables d’environnement peuvent être, respectivement, placées avant ou après les valeurs actuelles, ou remplacer les valeurs actuelles.

Nous recommandons Les profils en pratique du livre de cuisine.

Introduction ○○○○○○○	Gestion de paquets ○○○○○●○○○○○○○○	Création d'images ○○○○○○○○○	Une histoire de versions ○○○○○	Au final... ○○○○○
Gestion de <i>profil</i>				
<i>Profil</i> , 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
```

2.3 Gestion déclarative

Il devient vite fastidieux de saisir à l'invite du *shell* la liste des paquets et il apparaît naturel de vouloir sauvegarder cette liste dans un fichier. Par exemple, il n'est pas rare de versionner cette liste. Guix permet ceci, et un peu plus, au travers d'une gestion déclarative.

The slide shows a navigation bar with five items: 'Introduction', 'Gestion de paquets', 'Création d'images', 'Une histoire de versions', and 'Au final...'. The 'Gestion de paquets' item is highlighted. Below the bar, the title 'Gestion déclarative' is displayed. The main content area contains the text 'déclaratif = fichier de configuration' and a green box with the text 'Un fichier some-python.scm peut contenir cette déclaration :'. Inside this box is a Scheme code snippet:

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

. Below the code, the command `guix package --manifest=some-python.scm` is shown, followed by 'équivalent à' and the command `guix install python python-matplotlib python-numpy python-scipy`. The footer of the slide includes 'S. Tournier', 'Reproductibilité des environnements logiciels avec GNU Guix', and '14 / 40'.

Le fichier de configuration est usuellement appelé `manifest`. Il contient la liste des paquets comme saisi à l'invite du *shell*. Cette liste est une spécification du manifeste à instancier dans le profil.

The slide shows a navigation bar with five items: 'Introduction', 'Gestion de paquets', 'Création d'images', 'Une histoire de versions', and 'Au final...'. The 'Gestion de paquets' item is highlighted. Below the bar, the title 'Gestion déclarative : remarques' is displayed. The main content area contains the text 'Version? Nous le verrons dans la suite' and 'Langage? Domain-Specific Language (DSL) basé sur Scheme (« langage fonctionnel Lisp »)'. Below this, there are two bullet points: '► (Oui (quand (= Lisp parenthèses) (baroque)))' and '► Mais continuum :'. Under the second bullet point, there is a numbered list: '1 configuration (manifest)', '2 définition des paquets (ou services)', '3 extension', and '4 le cœur est écrit aussi en Scheme'. A pink box contains the text 'Guix est adaptable à ses besoins'. Below this, the text 'Déclaratif vs Impératif (et non pas Donnée inerte vs Programme)' and 'Programmation déclarative = programmation fonctionnelle ou descriptive (L^AT_EX) ou logique (Prolog)' is shown. The footer of the slide includes 'S. Tournier', 'Reproductibilité des environnements logiciels avec GNU Guix', and '15 / 40'.

Tout d'abord, il est important de noter que *déclaratif* ne signifie pas donnée inerte, comme l'on pourrait retrouver avec un format comme CSV ou YAML. Le terme *déclaratif* (« on décrit le *quoi*, c'est-à-dire le problème ») s'oppose à *impératif* (« on décrit le *comment*, c'est-à-dire la structure de contrôle correspondant à la solution »).

Avec le Tableau 1, nous espérons illustrer ce *continuum* dans Guix.

Language	files	blank	comment	code
Scheme	1105	47761	63485	804643
`-- packages	538	26787	37972	642152
`-- guix	272	9699	13346	75577
`-- services	53	3232	2764	26723
C++	50	3425	2198	10635
make	4	213	292	3087
Bourne Shell	29	850	1213	2891

TABLE 1 – `cloc --exclude-ext=patch --exclude-dir=po` .

Les paquets, les services et Guix lui-même sont écrits dans le même langage Scheme (pour être précis GNU Guile). Ce langage est dérivé du langage fonctionnel Lisp et il facilite la définition de langages dédiés (DSL, *domain-specific language*).

The screenshot shows a presentation slide with a navigation bar at the top containing five items: 'Introduction', 'Gestion de paquets', 'Création d'images', 'Une histoire de versions', and 'Au final...'. The 'Gestion de paquets' item is highlighted. Below the navigation bar, the slide title is 'Gestion déclarative : exemple de transformation (machine de Goldberg ^{:-)} (lien)'. The main content is a code block in Scheme DSL:

```
(define python "python")

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

Below the code block, the text reads: 'Guix DSL, *variables*, Scheme et chaîne de caractères.'

The footer of the slide contains: 'S. Tournier', 'Reproductibilité des environnements logiciels avec GNU Guix', and '16 / 40'.

Par exemple, nous illustrons la capacité de manipuler *programmiquement* les paquets. Avec le mots-clé `append`, deux listes sont concaténées. Le mot-clé `map` applique une fonction aux éléments d'une liste et retourne une liste. Le mot-clé `lambda` définit une fonction (anonyme).

Ainsi, il est possible de déclarer des transformations à appliquer aux paquets.

Transformation de paquets. Les transformations de paquets sont des options qui rendent possible la définition de variantes de paquets – par exemple, des paquets construits en utilisant une chaîne d'outils différente que celle par défaut.

Introduction ○○○○○○ **Gestion de paquets** ○○○○○○●○○○○○ Création d'images ○○○○○○○○ Une histoire de versions ○○○○○ Au final... ○○○○○○

Gestion déclarative

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

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 17 / 40

Introduction ○○○○○○ **Gestion de paquets** ○○○○○○●○○○○○ Création d'images ○○○○○○○○ Une histoire de versions ○○○○○ Au final... ○○○○○○

Gestion déclarative

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
```

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 18 / 40

Les transformations sont appliquées récursivement, donc elles s'appliquent aussi aux dépendances de dépendances. Par exemple, la ligne de commande pour recompiler le paquet `python` et toutes ses dépendances avec le compilateur GCC à la version 7 à la place de la version 10 (par défaut) est

```
guix install python --with-c-toolchain=python=gcc-toolchain@7
```

et ceci se traduit naturellement dans un fichier de configuration *manifest*.

```
(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"))))
```

Ici, nous utilisons le module définissant les transformations et nous définissons la transformation `transform` via la procédure `options->transformation`; autrement dit `transform` est une fonction prenant en argument un paquet et retournant un nouveau paquet dans lequel la chaîne de compilation a été remplacée. Pour finir, `compose` permet de composer deux fonctions et nous aurions pu écrire à la place

```
(lambda (pkg) (transform (specification->package pkg)))
```

Le manuel fournit un index de programmation qui aide à se familiariser avec le langage dédié.

2.4 Environnement isolé à la volée

À ce stade, nous sommes capables de créer des profils. Par exemple, la commande `guix package --list-profiles` liste tous les profils de l'utilisateur. Tant qu'il y a un élément qui pointe dans un profil, le ramasse-miettes ne peut pas *glaner* les paquets non utilisés. Dans certains cas, nous voulons temporairement tester un outil sans se préoccuper de la gestion du profil. C'est la fonctionnalité de `guix shell`.

Introduction
oooooooo

Gestion de paquets
oooooooooooooooo●o

Création d'images
oooooooooooo

Une histoire de versions
ooooo

Au final...
ooooo

Environnement isolé à la volée

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`

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 21 / 40

La commande `guix shell` permet de créer ou étendre temporairement un profil. Par exemple, dans la section précédente, le programme `cloc` est utilisé pour compter les lignes de code. La commande `guix shell cloc` a permis de lancer un *shell* temporaire contenant le programme.

Les options `pure` et `container` permettent de vérifier que toutes les dépendances de l'application sur laquelle on travaille sont bien capturées. En particulier, l'option `container` permet de lancer un *shell* temporaire totalement isolé.

Nota bene : La commande `guix shell` a été introduite récemment pour remplacer `guix environment`. Cette dernière est encore conservée pour assurer une rétro-compatibilité. Toutefois, nous recommandons d'utiliser `guix shell`, en particulier si vous n'avez jamais utilisé Guix auparavant.

3 Création d'images

Dans la section précédente, nous avons vu comment créer des environnements « isolés ». Il semble naturel de vouloir les partager ou les distribuer.

3.1 Création d'un *pack*

La technologie usuelle pour transporter du matériel est celle d'un **conteneur**. Cependant, il ne faut **pas perdre de vue** la **transparence** et la **reproductibilité**.

The screenshot shows a presentation slide with a navigation bar at the top containing five items: 'Introduction', 'Gestion de paquets', 'Création d'images', 'Une histoire de versions', and 'Au final...'. The 'Création d'images' item is highlighted with a blue bar. Below the navigation bar, the slide title is 'Création d'un pack' and the main heading is 'Comment capturer un environnement ? Conteneur'. The slide content includes the text 'Conteneur = smoothie :-)', two bullet points asking about Dockerfile construction and binary inclusion, a code block for a Dockerfile, and a question about regenerating images over time. The footer of the slide reads 'S. Tournier', 'Reproductibilité des environnements logiciels avec GNU Guix', and '22 / 40'.

Un conteneur est comme un « smoothie » dans le sens où nous pouvons immédiatement dire si nous l'aimons ou non, mais il est difficile, si ce n'est impossible, de donner précisément la liste exhaustive des ingrédients. Par conséquent, comment, à deux moments différents, reproduire exactement le même conteneur sur deux machines différentes ?

Un *pack* est un *lot de logiciels* enregistrés sous leur forme binaire dans un format spécifique. Le format d'archivage **tar** est probablement le plus connu, historiquement. Aujourd'hui, **Docker** ou **Singularity** sont très largement utilisés comme format de conteneur.

Le but d'un conteneur est de capturer l'intégralité de la pile logicielle pour fonctionner indépendamment du système hôte. Guix fournit une commande (`guix size`) pour examiner la taille des éléments à inclure. Par exemple, le paquet `python-numpy` ne représente que moins de 8% des 301,5Mo totaux nécessaires à son bon fonctionnement. Nous voyons que le paquet `python-numpy` a aussi besoin du paquet `python` ainsi que de la bibliothèque d'algèbre linéaire `openblas`, entre autres.

Introduction
○○○○○○○

Gestion de paquets
○○○○○○○○○○○○○○○○○○○○

Création d'images
○○●○○○○○○○

Une histoire de versions
○○○○○

Au final...
○○○○○○○

Création d'un *pack*

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.

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 25 / 40

Prenons un exemple : Construire et déployer cette présentation. Les deux commandes suivantes sur une machine ayant Guix permettent de récupérer les sources (L^AT_EX) puis de construire la présentation au format PDF.

```
guix shell git nss-certs \
  -- git clone https://gitlab.com/zimoun/jres22-tuto-guix
```

```
guix shell -m manifest.scm \
  -- rubber --pdf presentation.tex
```

Alice souhaite pouvoir générer cette présentation sur une infrastructure ne possédant pas Guix. À la place, l'infrastructure utilise des conteneurs Docker. Par conséquent, Alice empaquette au format Docker

```
guix pack \
  --format=docker \
  -C none \
  -S /bin=bin -S /lib=lib -S /share=share -S /etc=etc \
  -m manifest.scm \
  --save-provenance
```

qui produit l'élément dans le dépôt (*store*) `/gnu/store/n92...-rubber-...-docker-pack.tar`. Nous renvoyons à la section du manuel pour une description exhaustive des options de la commande `guix pack`. Notons que `--symlink / -S` ajoute les liens symboliques spécifiés dans le *pack*; par exemple, `-S /bin=bin` crée un lien symbolique `/bin` qui pointe vers le sous-répertoire `bin` du profil.

Ce *pack* une fois créé peut être chargé en local comme image Docker pour, par exemple, ajouter une étiquette (*tag*) et pousser cette image sur un dépôt (*registry*)

```
$ docker load < $(guix pack ...)
$ docker images
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
rubber-texlive-base-texlive-fonts-ec     latest  6f08ec6e1f4c  52 years ago  1.03GB
$ docker tag 6f08ec6e1f4c zimoun/jres
$ docker push zimoun/jres
```

où les points ... sont par exemple les options précédentes.

Il faut noter la date qui correspond au 1^{er} janvier 1970, c.-à-d. qui donne les conditions pour reproduire l'image bit-à-bit. D'autre part, il faut aussi noter que Guix vérifie si l'élément est déjà présent dans le dépôt (*store*).

Maintenant, il suffit pour Carole de télécharger depuis le dépôt (*registry*) le conteneur Docker. Par exemple, ce conteneur pourrait être utilisé pour de l'intégration continue à travers GitLab CI (voir l'exemple dédié pour construire cette présentation).



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

Pour aller plus loin : Peut-on extraire le manifeste Guix d'un *pack* au format Docker ?

Reproductibilité d'un conteneur Docker généré par Guix

Internet étant fait de serveurs qui vont et viennent, les URL risquent de ne plus pointer vers le contenu. Par conséquent, nous donnons ici des liens vers des archives (meilleure pérennité) pour le contenu de ce tutoriel :

- Source sur Software Heritage
- Déploiement sur Web Archive

en plus, évidemment, des JRES.

3.2 Création d'une machine virtuelle

Dans la section précédente, nous avons vu comment créer des *pack* contenant un déploiement de paquets. Cependant, nous souhaiterions aussi pouvoir déployer des services.

Introduction
○○○○○○

Gestion de paquets
○○○○○○○○○○○○○○○○○○

Création d'images
○○○○●○○○○

Une histoire de versions
○○○○○

Au final...
○○○○○

Création d'une machine virtuelle

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

`guix system` permet une configuration déclarative d'un système

- ▶ `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)

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 27 / 40

Sur le même principe que la commande `guix package`, il est possible de chercher les services disponibles avec `guix system search`. De plus, la commande `guix system` permet de générer une image suivant différents formats. Par exemple, ceci permet d'obtenir une image *bootable* qui donne une distribution complète – appelée Guix System.

La commande qui nous intéresse particulièrement est `guix system vm`. Elle permet de générer une machine virtuelle (VM) qui partage son dépôt avec son hôte (n'importe quelle distribution Linux avec Guix installé). L'avantage principal est une isolation complète qui, par exemple, permet de fournir des services. Le déploiement de services avec Guix mériterait un tutoriel en soi, nous donnons ici un point d'entrée.

Comme pour la gestion de paquets, la configuration de la machine virtuelle se fait dans un langage déclaratif. Le point d'entrée est le mot-clé `operating-system` (décrit dans le manuel) qui spécifie les options. En particulier, un champ liste les paquets à provisionner et un autre champ liste les services avec éventuellement leurs caractéristiques.

Par exemple, la commande

```
guix system vm src/example/vm-image.scm
```

produit le script `/gnu/store/0n5...-run-vm.sh` qui lance QEMU avec les éléments du dépôt (*store*) adéquats, c.-à-d.

Introduction 000000 Gestion de paquets 0000000000000000 Création d'images 0000000000 Une histoire de versions 00000 Au final... 000000

Création d'une machine virtuelle

Configuration déclarative d'une machine virtuelle

```
(use-modules (gnu) (guix) (srfi srfi-1))
(use-service-modules desktop mcron networking xorg)
(use-package-modules certs fonts xorg)

(operating-system
 (host-name "gnu")
 (keyboard-layout (keyboard-layout "us" "altgr-intl")))

 (users (cons (user-account
              (name "guest")
              (password "") ;no password
              (group "users"))
             %base-user-accounts))
```

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 28 / 40

Introduction 000000 Gestion de paquets 0000000000000000 Création d'images 0000000000 Une histoire de versions 00000 Au final... 000000

Création d'une machine virtuelle

Configuration déclarative d'une machine virtuelle (2)

```
(use-modules (gnu) (guix) (srfi srfi-1))
(use-service-modules desktop mcron networking xorg)
(use-package-modules certs fonts xorg)

(operating-system
 [...]
 (packages (append (list font-bitstream-vera nss-certs)
                  %base-packages))
 (services
  (append (list (service xfce-desktop-service-type)
                (simple-service 'cron-jobs mcron-service-type
                               (list auto-update-resolution-crutch))
                (service dhcp-client-service-type))))))
```

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 29 / 40

```
$ /gnu/store/0n5...-run-vm.sh -nic user
```

démarré la machine virtuelle avec, ici, le réseau. Ces machines virtuelles peuvent être administrées avec le paquet `virt-manager`.

Introduction ○○○○○○○	Gestion de paquets ○○○○○○○○○○○○○○○○○○	Création d'images ○○○○○○○○●	Une histoire de versions ○○○○○	Au final... ○○○○○
Création d'une machine virtuelle				
Machines virtuelles avec Guix				

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

4 Une histoire de versions

Le but de cette section est double, d'une part montrer par l'exemple la capacité de Guix de s'affranchir de l'état externe et d'autre part en quoi Guix est dit *fonctionnel*.

Version ? Graphe ?

Une question récurrente des nouveaux utilisateurs de Guix est : comment est-ce que je spécifie la version de l'outil à utiliser ? Une réponse à cette question appelle une autre question : qu'est-ce que l'on appelle version ? Est-ce la version du code source ? Est-ce la version du binaire ? Mais alors, pour le même code source transformé par deux chaînes de compilation différentes, appelle-t-on les deux binaires résultants la même *version* ?

The screenshot shows a presentation slide with a navigation bar at the top containing five items: 'Introduction', 'Gestion de paquets', 'Création d'images', 'Une histoire de versions' (which is highlighted with a black dot), and 'Au final...'. Below the navigation bar is a blue header with the text 'Quelle est ma version de Guix ?'. The main content is a terminal window with a light green background showing the following commands and output:

```
$ 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
```

Below the terminal window is a pink box containing the text: 'Un état fixe toute la collection des paquets et de Guix lui-même'. Underneath this box is a note: '(Un état peut contenir plusieurs canaux (*channel* = dépôt Git), avec des URL, branches ou commits divers et variés)'. At the bottom of the slide, there is a footer with 'S. Tournier', 'Reproductibilité des environnements logiciels avec GNU Guix', and '31 / 40'.

Guix capture un état : la liste de tous les paquets et les commandes Guix elles-mêmes. Les commandes Guix sont données par le canal principal et ce canal fournit un ensemble de paquets (et de services). Il est tout à fait possible d'étendre la collection de paquets en fournissant d'autres canaux pour avoir des paquets variants. Par exemple, l'initiative **GuixHPC** (qui correspond à Guix dans un contexte scientifique) essaie de maintenir une .

Pour spécifier un canal supplémentaire, l'utilisateur (sans droit particulier) doit modifier le fichier `$HOME/.config/guix/channels.scm`, p. ex.

```
(cons (channel
      (name 'guix-hpc)
      (url "https://gitlab.inria.fr/guix-hpc/guix-hpc"))
      %default-channels)
```


Définir un paquet

The slide shows a navigation bar at the top with five items: 'Introduction', 'Gestion de paquets', 'Création d'images', 'Une histoire de versions', and 'Au final...'. The current slide is 'Définir un paquet'. Below the title is a code block defining a Python package with its dependencies.

```
(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))))
```

Below the code block, there are two bullet points:

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

At the bottom of the slide, there is a note: (Quel commencement du graphe? Problème du *bootstrap* dont nous ne parlerons pas ici)

The footer of the slide contains: S. Tournier, Reproductibilité des environnements logiciels avec GNU Guix, 33 / 40

Un paquet correspond à un nœud du graphe. D'une part, il contient des *meta* informations comme le nom, la version du code source, un synopsis et description, la licence. D'autre part, il contient la recette pour être construit par Guix : où trouver les sources, quelle chaîne de compilation par défaut, quelles dépendances, et quelles modifications spécifiques (`arguments`) pour indiquer les options de compilation ou les changements mineurs dans la chaîne de compilation.

Pour faciliter la création de nouveaux paquets, la commande `guix import` permet de générer une recette Guix à partir des dépôts dédiés à un écosystème (PyPI, CRAN, Bioconductor, etc.). Il faut noter que l'*utilisabilité* de la recette résultante est très variable car elle dépend fortement de la qualité ou disponibilité des métadonnées du dépôt source.

De notre point de vue, empaqueter pour Guix est généralement plus facile que pour les autres gestionnaires. Cependant, la construction totalement isolée de Guix oblige à une certaine gymnastique. Au final, nous tenons à souligner que le travail d'empaquetage quelque soit le gestionnaire sous-jacent est un tâche laborieuse.

Changer d'état

La commande `guix pull` permet de changer d'état en mettant à jour la collection de paquets des différents canaux et Guix lui-même. Par exemple, la commande `guix pull -news` permet de savoir quelles sont les différences entre notre état précédent et notre état courant.

La commande `guix upgrade` permet de mettre à jour les paquets d'un profil. Cette commande respecte les transformations, s'il y a, dans la mise à jour.

En ayant la capacité d'utiliser d'autres canaux, de créer des profils et de mettre à jour la collection des paquets, un utilisateur sans privilège devient autonome dans la gestion de ses outils. Bien entendu, sur une machine mutualisée, l'administrateur peut créer des profils exposés aux utilisateurs pour leurs fournir des suites d'outils par défaut.

The screenshot shows a presentation slide with a navigation bar at the top containing five items: 'Introduction', 'Gestion de paquets', 'Création d'images', 'Une histoire de versions', and 'Au final...'. The 'Gestion de paquets' item is highlighted with a blue bar. Below the navigation bar, the slide title is 'Changer d'état' and the main heading is 'Concrètement, en pratique?'. The content includes a pink box stating 'une version = un graphe', followed by the text 'Mise à jour de Guix' and the command 'guix pull' with a note '(création d'une nouvelle *génération* interne)'. A blue arrow points to the text 'Il est possible de spécifier un *état* :', followed by a green box containing two terminal snippets: 'alice@laptop\$ guix describe --format=channels > alice-laptop.scm' and 'carole@desktop\$ guix pull --channels=alice-laptop.scm'. Another blue arrow points to the text 'Il est possible de se placer **temporairement** dans un état spécifique pour exécuter une commande (comme créer un profil)', followed by a green box with the command 'guix time-machine --commit=c61df1792c -- install python -p python/autres'. At the bottom, a footer bar contains 'S. Tournier', 'Reproductibilité des environnements logiciels avec GNU Guix', and '34 / 40'.

Comme il est possible de connaître l'historique d'installation et de suppression des paquets d'un profil avec la commande `guix package -l`, la commande `guix pull --list-generations` fournit l'historique des états de l'utilisateur. Notons que les commandes Guix elle-mêmes vivent dans un profil particulier (nommé `$HOME/.config/guix/current`).

Ainsi, sur une machine multi-utilisateurs, cette liste des états peut différer entre les utilisateurs. La commande `guix describe` permet de capturer l'état courant pour soit le *versionner* avec un projet, soit le partager. Par exemple, Alice sauvegarde son état courant dans le fichier `alice-laptop.scm` et transmet à Carole ce fichier. Ensuite, Carole sur une autre machine peut se placer dans l'exact même état qu'Alice.

Lorsque l'on travaille sur plusieurs projets, il peut devenir fastidieux de changer d'état *globalement* et il est souhaitable d'avoir la capacité de se placer

temporairement dans un état donné pour réaliser une opération (installation, création de profil, etc.). Ceci est possible avec la commande `guix time-machine`.

Introduction
○○○○○○○
Gestion de paquets
○○○○○○○○○○○○○○○○○○
Création d'images
○○○○○○○○○
Une histoire de versions
○○○○●
Au final...
○○○○○

Changer d'état
Reproductibilité en arrière, en avant : `guix time-machine`

Pour être reproductible dans le temps, il faut :

- ▶ Une préservation de **tous** les codes source ($\approx 75\%$ 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)

S. Tournier
Reproductibilité des environnements logiciels avec GNU Guix
35 / 40

Carole et Bob, les collaborateurs d’Alice, qui aussi travaillent sur d’autres projets avec d’autres collaborateurs peuvent convenir d’un état et s’y placer. Ils auront le même environnement logiciel. Grâce aux mécanismes d’inférieurs, il est possible de tendre à une reproductibilité dans le temps.

Pour assurer cette reproductibilité, le seul point sur lequel nous pouvons directement agir est la sauvegarde de tous les codes sources. Le projet Guix participe à la mise en place de ponts avec des systèmes d’archivage comme Software Heritage. Chaque état correspond à une collection de paquets et donc à des codes sources et des recettes Guix de construction. Le chantier est en cours pour préserver « quoi » et « comment » construire.

5 Au final...

Au final...

Introduction ○○○○○○○	Gestion de paquets ○○○○○○○○○○○○○○○○○○	Création d'images ○○○○○○○○○○	Une histoire de versions ○○○○○	Au final... ●○○○○○
-------------------------	--	---------------------------------	-----------------------------------	-----------------------

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

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

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 36 / 40

Introduction ○○○○○○○	Gestion de paquets ○○○○○○○○○○○○○○○○○○	Création d'images ○○○○○○○○○○	Une histoire de versions ○○○○○	Au final... ●○○○○○
-------------------------	--	---------------------------------	-----------------------------------	-----------------------

Au final...
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 **met 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**

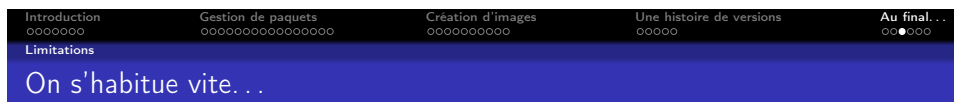
```
error: You found a bug
```

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

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 37 / 40

Limitations

En tant que contributeur régulier, j'ai une vision biaisée qui peut-être sur-estime les défauts à traiter et sous-estime toutes les fonctionnalités qui résolvent un large spectre de problèmes. Pour sûr, après plus de deux années d'utilisation intensive de l'ordinateur portable à la machine de calcul en passant pour la station de travail, je ne reviendrai pas à des outils sans un paradigme déclaratif et fonctionnel.



- ▶ 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

1. Le problème est la transparence de la chaîne de *bootstrap* : comment construire à « partir de rien ». Techniquement, en quelques mots, le nœud du problème est le port de la bibliothèque C `glibc` pour un autre noyau. Autrement dit, quelle est la taille de la graine binaire que l'on s'autorise comme racine du graphe des dépendances ?
2. Il est possible d'utiliser des pilotes spécifiques pour certains matériels *hardware* mais c'est du travail. D'un autre côté, dans un contexte de reproductibilité scientifique, ne doit-on pas chercher au maximum la transparence de toute la chaîne de traitement ?

L'autre implication est l'utilisation de logiciels requérant par exemple un jeton de licence – comme le compilateur Fortran d'Intel `ifort`. L'environnement de construction étant totalement isolé, l'accès au jeton est impossible lors de la compilation.
3. Tout est régulièrement amélioré, dans la mesure du possible avec les moyens disponibles.
4. On y vient, on n'en repart pas.

Ressources



Introduction ○○○○○○ Gestion de paquets ○○○○○○○○○○○○○○ Création d'images ○○○○○○○○ Une histoire de versions ○○○○○○ Au final... ○○○●○○

Ressources

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> 

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 39 / 40

Introduction ○○○○○○ Gestion de paquets ○○○○○○○○○○○○○○ Création d'images ○○○○○○○○ Une histoire de versions ○○○○○○ Au final... ○○○●○○

Ressources

Remerciements

- ▶ Ludovic Courtès
- ▶ Ricardo Wurmus (Guix comme bibliothèque Scheme : Guix Workflow Language)
- ▶ Mathieu Othacehe (Cuirass [\(lien\)](#) = CI qui assure la disponibilité des *substituts* binaires)
- ▶ tous les contributeurs (de la question au *patch* en passant par la rapport de *bug*)

« *Présidents* » de la session JRES :

- ▶ Yann Dupont
- ▶ Emmanuel Halbwachs

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 40 / 40

A Bonus

A.1 Définition fonctionnelle d'un paquet

Pour Guix, la construction d'un paquet est vue comme une fonction pure :

- la construction retournée est la même pour les mêmes arguments,
- l'évaluation n'a pas d'effets de bord.

Par exemple, si la compilation sauvegarde la date, alors la construction retournée n'est pas identiquement la même pour les mêmes arguments. Par ailleurs, l'évaluation a un effet de bord car fournir une date est une action qui a besoin d'éléments extérieurs qui ne dépendent pas uniquement ni exclusivement du corps de la fonction.

Bonus
●○○○○○
Définition fonctionnelle d'un paquet

Définition fonctionnelle (*derivation*) (A valeur d'illustration pour fixer les idées)

```
def pkg(source, build_sytem, arguments, inputs):
    src = fetch(source)
    import build_system as bs
    build = bs.builder(arguments)
    envBuild = build.add(inputs)
    binary = envBuild(src)
    return binary

python3 = pkg(Python, configure_make, deps=[bzip2, expat, ...])
```

- ▶ Les sources `src` sont obtenues à partir de la définition du paquet via `source`
- ▶ La recette `build` est déduite de la définition du paquet via `build-system` et `arguments` (`./configure && make && make install`)
- ▶ L'environnement `envBuild` est **totallement isolé** et ne contient que le nécessaire défini par `build-system` étendus de dépendances `inputs` supplémentaires

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 40 / 40

A.2 Autour du graphe des dépendances

Suivant la transformation de paquets que nous appliquons, il peut arriver que le *monde* soit recompilé. Ici, nous illustrons la signification d'un contrôle fin du graphe des dépendances.

Nous commençons par construire la paquet `python` avec le chaîne de compilation `gcc@7` via un fichier déclaratif *manifeste*. En étant attentifs, nous remarquons que le paquet `ghc-emojis` sera reconstruit.

Bonus
●○○○○
Autour du graphe des dépendances
Transformations : pourquoi tout peut-il être recompilé ?

```
$ guix build -m example/some-python-with-gcc7.scm --derivations --dry-run
...
/gnu/store/vi2j8aakp7jqxq8m97xvqk0d8q8i27s2-ghc-emojis-0.1.2.drv
...
```

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**

Pourquoi doit-on recompiler une bibliothèque Haskell sur les caractères emoji ?

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 40 / 40

La question est pourquoi une pile logicielle Python à caractère scientifique dépend-elle d'une bibliothèque Haskell de caractère emojis ?

Rappelons que Guix permet d'inspecter le graphe des dépendances. Par exemple, le paquet `python` a 137 dépendances (nœuds), majoritairement indirectes.

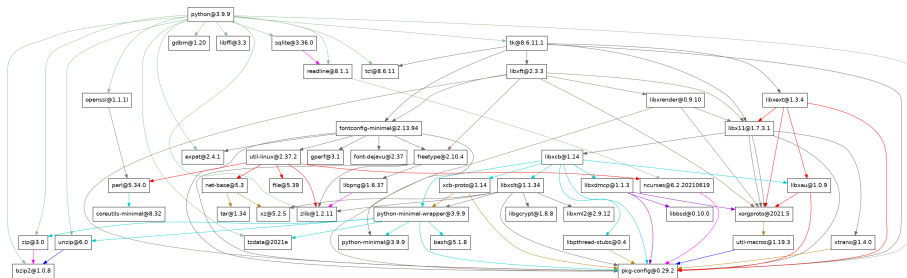
Nous cherchons donc quel est le chemin entre la bibliothèque scientifique Python Scipy et la bibliothèque Haskell Emojis. Incroyable, non ? Pour construire Scipy, une bibliothèque Python traitant principalement de calcul scientifique comme des solveurs de système linéaire, il faut donc un compilateur Haskell comme GHC.

Nous cherchons ensuite quel est le chemin entre la bibliothèque Haskell Emojis et l'interpréteur Python. Incroyable, à nouveau, car il y en a un. Le compilateur Haskell a besoin de Python.

Finalement, recompiler le paquet `python` va automatiquement recompiler `ghc`. Puis, comme le *build system* du paquet `ghc-emojis` n'est pas identiquement le même, ce paquet va être reconstruit. Et ainsi de suite. . .

Bonus
 Autour du graphe des dépendances
Graph 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

Bonus
 Autour du graphe des dépendances
Inspection du graphe des dépendances

python-scipy The Scipy library provides efficient numerical routines
 ghc-emojis Conversion between emoji characters and their names

Quelle dépendance entre une bibliothèque scientifique et une bibliothèque de pictogrammes ?

```
$ guix graph \
  --path python-scipy ghc-emojis
python-scipy@1.7.3
python-pydata-sphinx-theme@0.6.3
python-jupyter-sphinx@0.3.2
python-nbconvert@6.0.7
pandoc@2.14.0.3
ghc-emojis@0.1.2

$ guix graph \
  -t bag-emerged \
  --path ghc-emojis python
ghc-emojis@0.1.2
ghc@8.10.7
python@3.9.9
```

Guix permet un contrôle fin de l'arbre des dépendances et son inspection

A.3 Pour fixer les idées

Bonus
○○○○●○
Pour fixer les idées

Que signifie exécuter un programme ?

```
python3 -c 'print(1+2)'
```

- ▶ python3 est ici un exécutable binaire (lisible par une machine)
- ▶ Le code source de *Python* est accessible (lisible par une personne)
- ▶ python3 dépend à l'exécution de bibliothèques (dépendances)

Comment python3 est-il obtenu ?

Analogie : yahourt = *Lait* + skyr (Skyr : Yaourt islandais)

python3 = *Python* + compilateur (+ make + etc.)

Conclusion

- ▶ Les sources (*Python*) ne suffisent pas pour la Reproductibilité,
- ▶ il faut aussi connaître deux environnements :
 - ▶ de construction (compilateur, make, etc.),
 - ▶ d'exécution (dépendances)

S. Tournier Reproductibilité des environnements logiciels avec GNU Guix 40 / 40

Pour trouver les dépendances nécessaires à l'exécution, il suffit de lancer `ldd $(which python3)`. À titre d'illustration, ces bibliothèques sont nécessaires : `linux-vdso.so.1 libpthread.so.0 libutil.so.1 libm.so.6 ...`

Par conséquent, on peut se demander :

1. Comment le binaire `python3` a-t-il été produit ? Quel compilateur ? Quelles dépendances ?
2. Comment les dépendances ont-elles été produites ?

Bonus
○○○○○●

Pour fixer les idées

Qu'est-ce qu'un gestionnaire de paquets ?

gestionnaire des paquets = gestion des dépendances

fournit dans une mesure l'environnement de construction et/ou d'exécution.

Est-ce que `foo = bar` ?

- ▶ `foo <- Source@1.2 + env-construction@A`
- ▶ `bar <- Source@1.2 + env-construction@B`

Comment s'assurer que deux environnements sont identiques ?

Conclusion

Trop de combinaisons pour avoir le même environnement :

- ▶ sur plusieurs machines,
- ▶ à différents points dans le temps.