nix

from a build system to an ansible replacement

build system

GNUMAKE

```
# a typical makefile
edit : main.o command.o display.o insert.o utils.o
        cc -o edit main.o command.o display.o insert.o utils.o

main.o : main.c defs.h
        cc -c main.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
utils.o : utils.c defs.h
        cc -c utils.c
```

build system

GNUMAKE

```
utils.o : utils.c defs.h
        cc -c utils.c

# boils down to
target : prerequisites
        recipe
```

prerequisites are **referenced** in the recipe, thus **should** be considered part of the recipe
target is the **consequence** of the recipe, thus **should** be a function of the recipe

build system

NIX

let’s name the minimal unit of a build as: *derivation*
(disclaimer: codes bellow are just rough approximations of how nix really works)

```
utils.o : utils.c defs.h
        cc -c utils.c
```

```
derivation {
  builder = "cc -c utils.c";
};
```

name the output

```
# evaluate the derivation, and normalize it
{"builder": "cc -c utils.c"}
# then we hash the normalized derivation string
0y876lqq57qyb4×5j0i374zl3gifwqa3nnc4q603gc2jnd42rfr6
```

build system

NIX

to ensure the reproducibility of nix derivations, we also have to track the inputs
a natural design is that every input must be the output of another derivation

```
cc = derivation { ... };
utils.c = derivation { ... };
utils.o = derivation {
  builder = "${cc} -c ${utils.c}";
};
```

```
# which is normalized into
{"builder": "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i -c 0jbrxcbbrdvfpwljhcw7mr64hln8l9ccm ... "}
```

take notice of the implicit input: defs.h, which is likely to be missed by other build systems
nix build derivations in **sandboxes** in which only the referenced pathes can be reached
no more unexpected build failures

build system

NIX

relative paths are inherently ambiguous
thus we replace them with absolute pathes, by prepending a common prefix: **the nix store**

```
{"builder": "/nix/store/0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i -c /nix/store/0jbrxcbbrdv ... "}
```

everytime **any** of the inputs of a derivation is changed, wether it's the compiler or a source file
the derivation itself is changed, thus the inputs are tracked effectively and reliably

build system

NIX

further enhancment to nix derivations

```
cc = derivation { ... };
utils.c = derivation { ... };
utils.o = derivation {
  # hashes are nice for machines, not humans (name is appended to the original hash)
  name    = "utils.o";
  # we can tell the builder where to put our output at build time
  builder = "${cc} -o $out -c ${utils.c}";
  # the platform, on which the derivation could be build
  system  = "x86_64-linux";
  # extra key value pairs that would be passed to the builder as envvars
  foo     = "bar";
};
```

build system

NIX

and expand the definition of derivation output from a single file, to a tree

```
# take a look inside cc the derivation:
.
├── bin
│   └── cc
├── include
│   └── c++
```

but how do we hash a tree of files
we can first combine them into a tarball then hash the tarball
however tar is not deterministic, thus we serialize it into Nix ARchive, a deterministic archive format

build system

NIX

not all files are *built*, some are just lying around, for example, utils.c
we treat them specially and give them the name: *fixed output derivations* (or FOD)
their path is not the hash of the derivation itself, but rather the hash of it's output

```
derivation {
  name       = "utils.c";
  builder    = "none";
  system     = "x86_64-linux";
  outputHash = "0jbrxcbbrdvfpwljhcw7mr64hln8l9ccmpydb680n9wp3mm974sd";
};
```

as FODs have fixed output hashes, they can be built outside of the sandbox

build system

NIX

and, we add a syntax sugar into the nix language: you can reference a local path as a derivation which will be copied into the nix store at evaluation time, and transformed into a FOD

```
cc = derivation { ... };
utils.o = derivation {
  name    = "utils.o";
  src     = ./utils.c;
  builder = "${cc}/bin/cc -o $out -c $src";
  system  = "x86_64-linux";
};
```

build system

NIX

binary caches
given a derivation, we know that it will be *realized* into the same output
whenever, wherever

> never build software more than once

we can serve our nix store as a binary cache
to reuse store pathes across machines

there’s no need to build the same chromium variant twice

build system

NIX

- derivation is the smallest unit of compilation in nix, just like a rule in makefile
- derivation is the combination of what and how to build the output
- the inputs of a derivation must be derivations
- the absolute path of the output is determined by hashing the derivation’s normalized form, before building
- derivations are built in sandboxes with no network access and only access to required pathes
- a special type of derivation: FOD exists, to bring in sources as derivations
- build results can be shared as binary caches, eliminating the need for duplicated builds

build system

NIX

nix is a build system, but most of the existing projects are already using other build systems
let me introduce: generic builders, which bridges the gap between nix and them

```
builder = ''
# bring deps into PATH
for dep in $deps; do
  export PATH=$dep/bin:$PATH
done
# extract source into PWD (which is an isolated tmp)
tar -xf $src
./configure --prefix=$out
make
make install
'';
hello = derivation {
  name    = "hello";
  src     = ./hello.tar.gz;
  builder = builder;
  system  = "x86_64-linux";
  deps    = [ gcc gnumake ... ]
};
```

build system

NIX

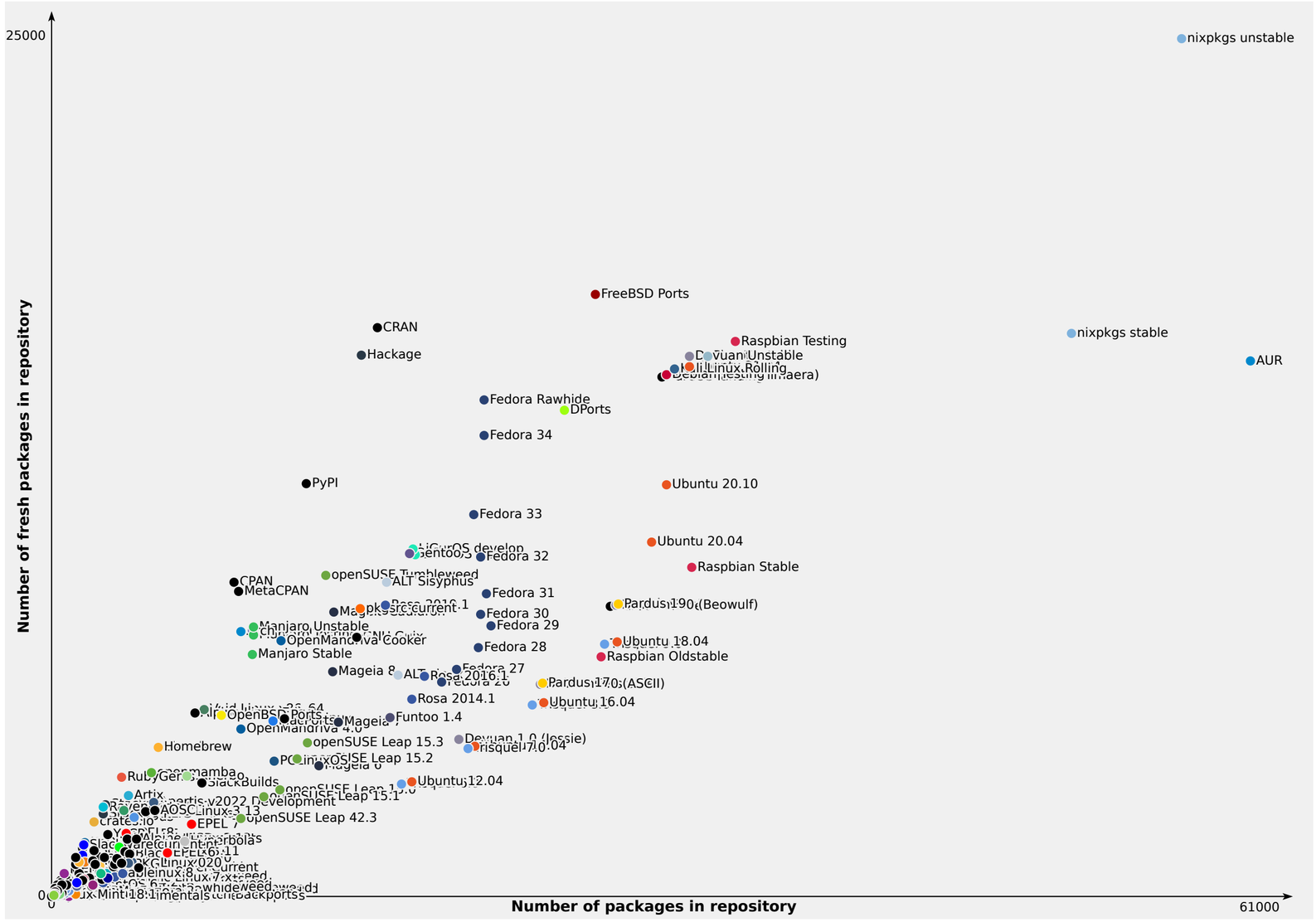
rust, golang, c++, python...
there are so many languages and frameworks
can we unite the efforts in building softwares
thus we have **nixpkgs**

> The Nix Packages collection (Nixpkgs) is a set of thousands of packages for the Nix package manager, released under a permissive MIT/X11 license. Packages are available for several platforms, and can be used with the Nix package manager on most GNU/Linux distributions as well as NixOS.

nixpkgs provides not only the compiler toolchains as a basis for other packages, but also infrastructures for packaging applications written in various languages and frameworks
and

build system

NIX



build system

NIX

a real life example of building a golang package

```
{ fetchgit, buildGoModule, fetchFromGitHub, lib }:
buildGoModule rec {
  pname = "k0sctl";
  version = "v0.8.4";
  src = fetchgit {
    url = "https://github.com/k0sproject/k0sctl";
    rev = "v0.8.4";
    sha256 = "08k0aa73kb4hs4zl8a2nmasag0czmppb2r0s1afj287c2a4ynw73";
  };
  vendorSha256 = "sha256-8GFZxjkLeTGWxJ3uzaPZaeeJzmmPN9Ao3z8a3JooP0s=";
  subPackages = [ "." ];
  meta = with lib; {
    homepage = "https://github.com/k0sproject/k0sctl";
    license = licenses.asl20;
  };
}
```

build system

NIX

and I hear gentoo users screaming: where are my use flags

```
{ fetchurl, fetchpatch, lib, stdenv, pkg-config, libgcrypt, libassuan, libksba
, libgpgerror, libiconv, npth, gettext, texinfo, buildPackages
, guiSupport ? true, pinentry ? null }:
stdenv.mkDerivation rec {
  pname = "gnupg";
  version = "2.2.27";
  src = fetchurl {
    url = "mirror://gnupg/gnupg/${pname}-${version}.tar.bz2";
    sha256 = "1693s2rp9sjwvdslj94n03wnb6rxysjy0dli0q1698af044h1ril";
  };
  nativeBuildInputs = [ pkg-config texinfo ];
  buildInputs = [
    libgcrypt libassuan libksba libiconv npth gettext
    readline libusb1 gnutls adns openldap zlib bzip2 sqlite
  ];
  pinentryBinaryPath = pinentry.binaryPath or "bin/pinentry";
  configureFlags = [
    "--with-libgpg-error-prefix=${libgpgerror.dev}"
  ] ++ optional guiSupport "--with-pinentry-pgm=${pinentry}/${pinentryBinaryPath}";
}
```

package manager

NIX

being able to build packages in a reproducible, declarative and reliable manner is great
but it makes little sense if we cannot **install** them
which, is the role of a package manager

package manager

NIX

the build time dependencies are explicitly specified
and the runtime dependencies are **automatically** recognized
by serializing the store path into Nix ARchive
then search for references to other store pathes within it

```
$ nix-store -q --references /nix/store/x64nf80gzcq6v5j1isnnkvwd73n9v4sy-nano-5.7
/nix/store/sbbifs2ykc05inws26203h0xwcadnf0l-glibc-2.32-46
/nix/store/9m4hy7cy70w6v2rqjmhvd7ympqkj6yxk-ncurses-6.2
/nix/store/x64nf80gzcq6v5j1isnnkvwd73n9v4sy-nano-5.7
```

by searching for references recursively, we can form a **closure** of the package to install
that is, itself and all it’s direct and transitive runtime dependencies

package manager

NIX

when we want to *install* a derivation
there are generally two possibilities

1. the closure is already in the store or available from a binary cache
2. it has to be built

the first case is trivial, for the second case, we build it, then we infer the closure
now that it's already in the nix store, we only have to create a **view** of the nix store to make it accessible

```
activation = "export PATH=${hello}/bin:${nano}/bin";
```

a new concept: **activation script**
activation script is an idempotent shell script the bring *whatever* in the nix store into life

for the example above, we may as well symlink all the packages to install, into /usr/bin
mimicking the behavior of traditional package managers

package manager

NIX

advantages over traditional package manager

- no SAT solver
- install multiple versions or variants of the same package together
- no assumptions about the global state of the system
- no replace file in-place
- no worry about kernel panic midway a upgrade

package manager

NIX

> winget can only install packages for now, but not remove them

garbage collection to the rescue
when we install a derivation, we register it as a gcroot
when we remove it, we deregister it
periodically, we walk through all reachable store pathes from the gcroots
then we can safely remove the unreachable pathes

operating system

FHS

an operating system is a collection of packages, and their configurations

```
.
├── boot
├── etc
└── usr
    ├── bin
    ├── include
    ├── lib
    └── share
```

operating system

NIXOS

why there must be an an artificial separation between packages and configurations

```
.
├── boot
└── nix
    ├── store
    │   ├── zzbgphm1v31zfi7×3hvjdwlxj3kl9vg-activate
    │   └── zzvq5qwlm2xikawfqxb0q8gl2bw391a9-foo
    │       ├── bin
    │       ├── share
    │       ├── include
    │       └── lib
    └── var/nix/profiles/system/activate -> /nix/store/zzbgphm1v31zfi7×3hvjdwlxj3kl9vg-activate
```

operating system

NIXOS

```
{ config, pkgs, ... }: {
  boot.loader.systemd-boot.enable = true;
  boot.kernel.sysctl = {
    "kernel.panic" = 10;
    "kernel.sysrq" = 1;
  };
  boot.kernelParams = [
    "quiet"
    "mitigations=off"
  ];
  fileSystems."/".device.label = "nixos";
  environment.systemPackages = with pkgs;[ git gnupg rustc ];
  services.sshd.enable = true;
}
```

declaratively config your operating system, from the bootloader, to every running service

configuration management

ANSIBLE

expand the configuration of a single host, to a fleet of hosts

```yaml
- name: configure db servers
  hosts: databases
  remote_user: root
  tasks:
  - name: ensure that postgresql is started
    ansible.builtin.service:
      name: postgresql
      state: started
```

NIXOS

```nix
# postgresql.nix
{ config, pkgs, lib, ... }: {
  services.postgresql.enable = true;
}
# for every host with database role
{ config, pkgs, lib, ... }: {
  imports = [ ./postgresql.nix ];
}
```

configuration management

NIXOS

the power of nixos roots in the nix language

- validate configuration fields before deployment
- reference values across modules of configuration
- reuse configuration efficiently
- unified language for any system component

configuration management

NIXOS

```
{ pkgs, config, ... }:
let
  socketPath = "${config.services.traefik.dataDir}/podman.sock";
in
{
  virtualisation.oci-containers.backend = "podman";
  virtualisation.oci-containers.containers = { ... };
  systemd.services.podman-traefik.serviceConfig.ExecStart = ''
    ${pkgs.socat}/bin/socat \
    UNIX-LISTEN:${socketPath} \
    UNIX-CONNECT:/run/podman/podman.sock
  '';
  services.traefik = {
    enable = true;
    staticConfigOptions = {
      providers.docker = {
        endpoint = "unix://${socketPath}";
      };
    };
  };
}
```

for the curios

- hail hydra !

  nixos.org
- still want to dive deeper

  nix pills
- how to get something done

  nix.dev
- telegram group

  @nixos_zhcn