Type-driven configuration management with Propellor

Joey Hess
Linux.Conf.Au 2017

Can you read this?

font_size = Point 20

room_size = Approx (Feet 100)

display_size = Feet 5

Best practices for system configuration

- Simple file formats
- Declarative
- Compositional

Configuration management

Ansible config file

- “simple” YAML format
- variables
- conditionals
- loops

```
tasks:
    - command: echo {{ item }}
      with_items: [ 2, 4, 6, 10 ]
      when: item > 5
```

Puppet config file

- simple INI file format
- variables
- hashes

```
ssldir = $vardir/ssl
    {owner = service, mode = 0771}
```

- Separate Puppet Language for conditionals and loops

Configuration management

Ansible config file

- “simple” YAML format
- variables
- conditionals
- loops

Turing complete!

not Turing complete .. or is it?

```
tasks:
    - command: echo {{ item }}
      with_items: [ 2, 4, 6, 10 ]
      when: item > 5
```

Puppet config file

- simple INI file format
- variables
- hashes

```
ssldir = $vardir/ssl
    {owner = service, mode = 0771}
```

- Separate Puppet Language for conditionals and loops

Turing complete config files

- Turing tar-pit

  “everything is possible but
  nothing of interest is easy”

  – Alan Perlis

- Not declarative

Turing complete config files

- Turing tar-pit

  "everything is possible but nothing of interest is easy"

  – Alan Perlis

- Not declarative

- Embedded Domain-Specific Languages
  - Make common cases easy

- Type checker
  - Avoid bad configurations

Propellor config.hs

```
main :: IO ()
main = defaultMain hosts

hosts :: [Host]
hosts = [foo, bar]

foo :: Host
foo = host “foo.example.com” $ props
   & osDebian (Stable “jessie”) X86_64
   & Apt.stdSourcesList
   & Apt.installed [“openssh-server”]
```

Propellor config.hs

```
main :: IO ()
main = defaultMain hosts

hosts :: [Host]
hosts = [foo, bar]

foo :: Host
foo = host “foo.example.com” $ props
   & osDebian (Stable “jessie”) X86_64
   & Apt.stdSourcesList
   & Apt.installed [“openssh-server”]
```

Propellor config.hs

```
main :: IO ()
main = defaultMain hosts

hosts :: [Host]
hosts = [foo, bar]

foo :: Host
foo = host “foo.example.com” $ props
   & osDebian (Stable “jessie”) X86_64
   & Apt.stdSourcesList
   & Apt.installed [“openssh-server”]
```

`hosts` has type list of `Host`

`foo` has type `Host`

Property

The basic building block of Propellor.

osDebian :: DebianSuite -> Architecture -> Property

```
server :: Property
server = osDebian (Stable “jessie”) X86_64

Apt.installed :: [Package] -> Property

sshserver :: Property
sshserver = Apt.installed [“openssh-server”]
```

The basic buildi[illegible]ock

osDebian is a function that takes two parameters, a DebianSuite and an Architecture, and makes a Property.

osDebian :: DebianSuite -> Architecture -> Property

```
server :: Property
server = osDebian (Stable “jessie”) X86_64

Apt.installed :: [Package] -> Property

sshserver :: Property
sshserver = Apt.installed [“openssh-server”]
```

Custom data types

Custom data types used to create Properties

- `Stable “jessie”  :: DebianSuite`
- `X86_64           :: Architecture`
- `“openssh-server” :: Package`
- `User “joeyh”     :: User`
- `Group “adm”      :: Group`
- `SshEdcsa         :: SshKeyType`
- `Port 80          :: Port`

Over 150 such data types used in Propellor

Custom data types

Custom data types used to create Properties

- `Stable “jessie”  :: DebianSuite`
- `X86_64           :: Architecture`
- `“openssh-server” :: Package`
- `User “joeyh”     :: User`
- `Group “adm”      :: Group`
- `SshEdcsa         :: SshKeyType`
- `Port 80          :: Port`

“Did you mean SshEcdsa”?

Over 150 such data types used in Propellor

Why types?

- Proof

Composing Property

```
requires :: Property -> Property -> Property
before   :: Property -> Property -> Property
fallback :: Property -> Property -> Property
onChange :: Property -> Property -> Property

securefoo :: Property
securefoo =
   Apt.installed [“foo”]
      `requires` File.containsLines “/etc/foo” [“secure=1”]
      `onChange` Service.restarted “foo”
```

Composing Property

```
requires :: Property -> Property -> Property
before   :: Property -> Property -> Property
fallback :: Property -> Property -> Property
onChange :: Property -> Property -> Property

securefoo :: Property
securefoo =
   Apt.installed [“foo”]
      `requires` File.containsLines “/etc/foo” [“secure=1”]
      `onChange` Service.restarted “foo”
```

x `requires` y
Is just another way to write
requires x y

Compositional Desert

- Things can be composed, but everything looks the same

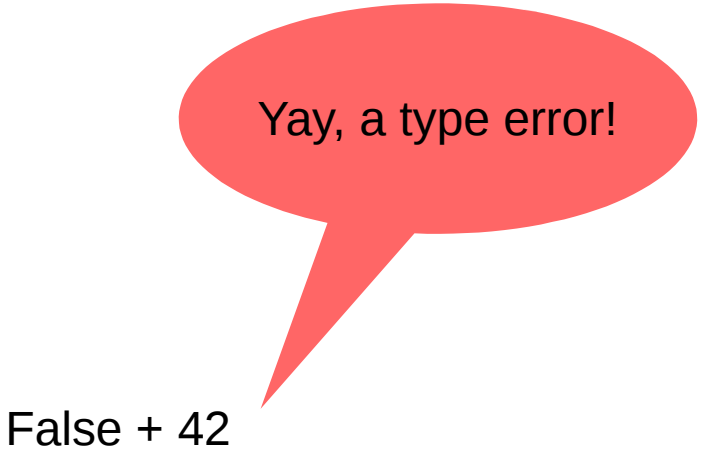
```
Property -> Property -> Property
Property -> Property -> Property
```

- These types are not preventing problems.

- “Container orchestration” etc

Avoiding the compositional desert

- Refine the `Property` type
- Make bad combinations be type errors.



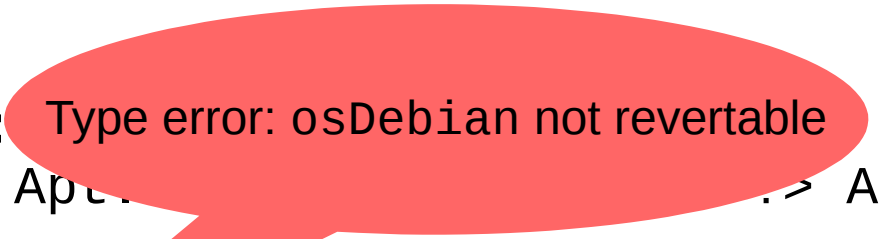
RevertableProperty

```
bar :: RevertableProperty
bar = Apt.installed [“bar”] <!> Apt.removed [“bar”]

foo :: Host
foo = host “foo.example.com” $ props
   & osDebian (Stable “jessie”) X86_64
   & Apt.stdSourcesList
   & bar
```

RevertableProperty

```
bar :: RevertableProperty
bar = Apt.installed [“bar”] <!> Apt.removed [“bar”]

foo :: Host
foo = host “foo.example.com” $ props
   & osDebian (Stable “jessie”) X86_64
   & Apt.stdSourcesList
   ! bar
```

RevertableProperty




```
bar :: 
bar = Apt                          .> Apt.removed [“bar”]

foo :: Ho
foo = host “foo.example.com” $ props
    ! osDebian (Stable “jessie”) X86_64
    & Apt.stdSourcesList
    ! bar
```

Composing RevertableProperty

```
webserverDocked :: RevertableProperty
webserverDocked = Docker.docked webserver

webContent :: Property
webContent = File.hasContent index [“hello, world”]
  Where index = “/var/www/index.html”

webserver :: RevertableProperty
webserver = webserverDocked `requires` webContent
```

Type error:
webContent is not revertable

Composing RevertableProperty

```
webserverDocked :: RevertableProperty
webserverDocked = Docker.docked webserver
    <!> Docker.undocked webserver

webContent :: RevertableProperty
webContent = File.hasContent index [“hello, world”]
      <!> File.notPresent index
  where index = “/var/www/index.html”

webserver :: RevertableProperty
webserver = webserverDocked `requires` webContent
```

When reverted, `requires` runs backwards!

RevertableProperty: Good and Bad

```
webserverDocked :: RevertableProperty
webserverDocked = Docker.docked webserver
    <!> Docker.undocked webserver

webContent :: RevertableProperty
webContent = File.hasContent index [“hello, world”]
      <!> File.notPresent index
  where index = “/var/www/index.html”
```

Containers

- Docker
- Systemd machine
- Chroot
- FreeBSD jail

Containers

```
webserver :: Systemd.Container
webserver = systemd.debContainer “webserver” $ props
    & osDebian Testing X86_64
    & Apt.installedRunning “apache2”
    & Systemd.bind “/var/www”

foo :: Host
foo = host “foo.example.com” $ props
    & Systemd.nspawned webserver
```

Containers

```
webserver :: Systemd.Container
webserver = systemd.debContainer “webserver” $ props
    & osDebian Testing X86_64
    & Apt.installedRunning “apache2”
    & Systemd.bind “/var/www”

foo :: Host
foo = host “foo.example.com” $ props
    & Systemd.nspawned webserver
```

Composition of several `Propertys` into `Systemd.Container`

DNS configuration

```
hosts = [web, dns]

web = host “web.example.com” $ props
    & ipv4 “42.42.1.1”
    & alias “www.example.com”
    & Systemd.nspawned webserver

dns = host “dns.example.com” $ props
    & ipv4 “64.64.1.1”
    & Dns.primary hosts “example.com” (mkSOA “ns1.example.com” 0) []
```

Info

```
& ipv4 “42.1.1.1”
& alias “www.example.com”
& osDebian (Stable “jessie”) X86_64
& Systemd.bind “/var/www”
& Ssh.hostPubKey SshEd25519 “…”
```

Seen many of these before.

Each is a Property that configures a Host with `Info`

Info

```
& ipv4 “42.1.1.1”
& alias “www.example.com”
& osDebian (Stable “jessie”) X86_64
& Systemd.bind “/var/www”
& Ssh.hostPubKey SshEd25519 “…”
```

What is the type of Info?

Seen many of these before.

Each is a Property that configures a Host with `Info`

IsInfo

A value of `Info` combines multiple types in the `IsInfo` class.

```
toInfo :: IsInfo v => v -> Info
addInfo :: IsInfo v => Info -> v -> Info
fromInfo :: IsInfo v => Info -> v
```

Almost dynamic programming, but still preserves type safety.

```
toInfo (dnsAlias “www.example.com”)
    `addInfo` (Debian (Stable “jessie”) X86_64)
```

`Info` Propagation Problem

- Property adds `Info`
- Hosts have `Info`
- Containers can have `Info` too
- Need to make sure the Info is propagated when composing these types.

```
Property HasInfo
Property NoInfo
```

Supported Operating Systems

- Originally: Linux (Debian specifically; maybe Ubuntu etc)

Type Level Supported Operating Systems

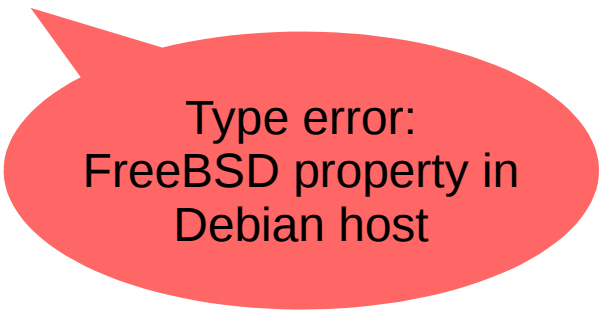
- `Property DebianLike`
- `Property FreeBSD`
- `Property UnixLike`
- `Property (DebianLike + FreeBSD)`
- `Property (HasInfo + DebianLike)`

Type Level Supported Operating Systems

- `Property DebianLike`
- `Property FreeBSD`
- `Property UnixLike`
- `Property (DebianLike + FreeBSD)`
- `Property (HasInfo + DebianLike)`

Type Level Supported Operating Systems

```
foo :: Host
foo = host “foo.example.com” $ props
    & osDebian (Stable “jessie”) X86_64
    & Pkg.updated
```

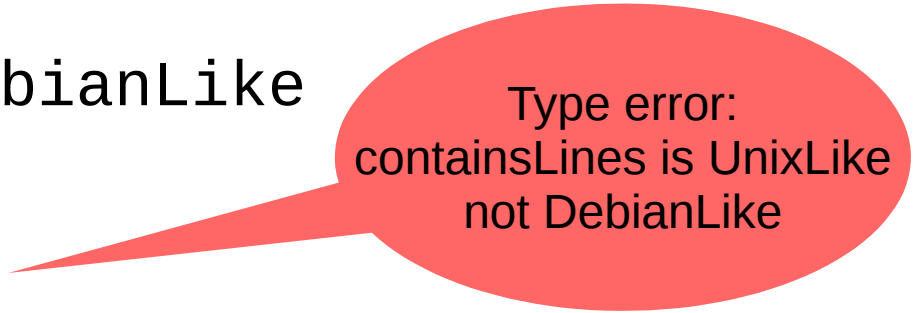


Composing Supported Operating Systems

```
Apt.upgraded :: Property DebianLike
Pkg.upgraded :: Property FreeBSD

upgraded :: Property (DebianLike + FreeBSD)
upgraded = Apt.upgraded `pickOS` Pkg.upgraded
```

Small Problem..

```
fsckfix :: Property DebianLike
fsckfix =
    File.containsLines
        “/etc/default/rcS” [“FSCKFIX=yes”]
```



Solution

```
fsckfix :: Property DebianLike
fsckfix = tightenTargets $
    File.containsLines
        “/etc/default/rcS” [“FSCKFIX=yes”]
```

`tightenTargets` makes the supported OS list more specific.

```
tightenTargets :: ?? -> ??
```

Type level port conflict detection

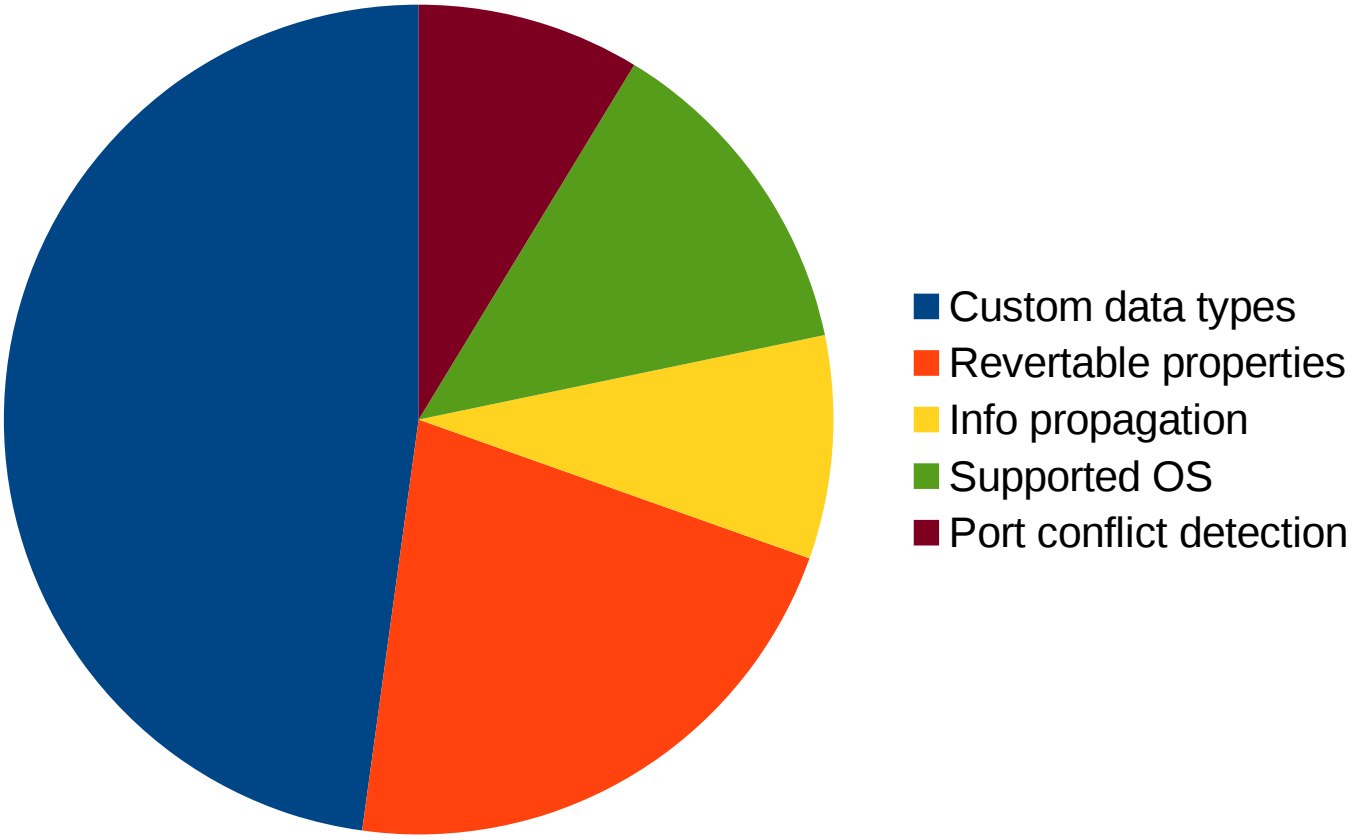
webserver :: Property (UsingPort 80 + UsingPort 443 + DebianLike)

torBridge :: Property (UsingPort 443 + DebianLike)

Problem: Configuration is now being done at type level?

Problem: Virtual hosts

Problems Avoided Using Types

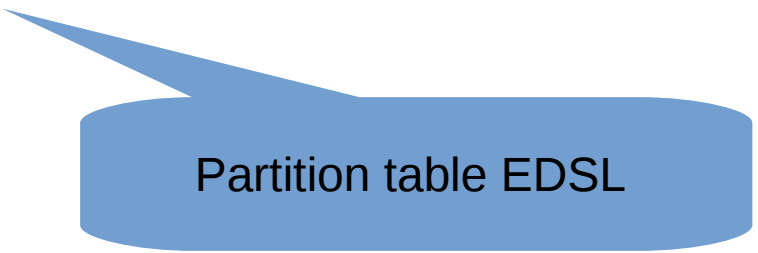


Bonus: `Host` functions

```
web :: Word8 -> Host
web n = host hn $ props
    & ipv4 (“42.42.1.” ++ show n)
    & alias “www.example.com”
    & Systemd.nspawned webserver
  where hn = “www” ++ show n ++ “.example.com”

hosts = [dns] ++ map web [1..10]
```

Bonus: Disk image creation

```
foo :: Host
foo = host “foo.example.com” $ props
    & imageBuilt “/srv/diskimages/bar-disk.img” (hostChroot bar) MSDOS (grubBooted PC)
        [ partition EXT2 `mountedAt` "/boot" `setFlag` BootFlag
        , partition EXT4 `mountedAt` “/” `addFreeSpace` MegaBytes 5000
        , swapPartition (MegaBytes 256)
        ]

bar :: Host
bar = host “bar.example.com” $ props
    & osDebian Unstable X86_64
    & Apt.installed [“linux-image-amd64”]
    & hasPassword (User “root”)
```



Questions?

https://propellor.branchable.com/