Type Your `.conf` for Fun and Profit

Eugene Smolanka

#kievfprog / Mar 18, 2017

Type Your .conf for Profit

Eugene Smolanka

#kievfprog / Mar 18, 2017

```
~ % find /etc -type f | wc -l
931
```

On typical fresh installation of typical Ubuntu Server.

Why Configure

- The best configuration is no configuration at all.
  Rarely affordable.
- A practical system is often prone to changing requirements.
  Better to have some aspects of its behavior configurable.
- Certainly not desirable to tweak code, rebuild, redeploy every time some parameter needs to be changed.
  Especially, if your user ≠ you.

The Problem

Goal: Provide a flexible and powerful yet maintainable way to tune program's behavior without changing the program itself.

- Configuration starts as a few command line flags or a simple key-value file.
  As software evolves the complexity of configuration grows, too.
- As the complexity grows the maintainability gets worse.
  Easier to introduce a costly mistake, harder to introduce configuration changes.

Examples

Nginx

3rd most popular web server on the Internet
with simple C-style configuration.

Easy-peasy

```
server {
    listen      80;
    server_name example.org www.example.org;
    root        /data/www;
    index       index.html index.php;

    location ~* \.(gif|jpg|png)$ {
        expires 30d;
    }

    location ~ \.php$ {
        fastcgi_pass  localhost:9000;
        fastcgi_param SCRIPT_FILENAME
                      $document_root$fastcgi_script_name;
        include       fastcgi_params;
    }
}
```

Trickier one

```
http {
    # ---------------------------------✂----------------------------------
    log_format foobar '$remote_addr - $remote_user [$time_local] '
                      '"$request" $status "$http_referer" '
                      '"$http_user_agent"';

    server {
        # ---------------------------------✂----------------------------------
        map $status $loggable {
            ~^[23]  0;
            default 1;
        }
        access_log /path/to/access.log foobar if=$loggable;
        #                              ^^^^^^    ^^^^^^^^^
    }
}
```

“If” Considered Harmful

```
http {
    # ------------------------------✂----------------------------------
    server {
        # ------------------------------✂----------------------------------
        location / {
            set $true 1;
            if ($true) {
                add_header X-First 1;
            }
            if ($true) {
                add_header X-Second 2; # ← won't fire
            }
            return 204;
        }
    }
}
```

Ansible

Cloud automation and orchestration tool
with YAML-based configuration language.

Example Playbook

```yaml
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running (and enable it at boot)
    service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

Defining Variables

"Variables" = Dynamically scoped let-bindings.

- Inline in a playbook or imported from another playbook:

```
- hosts: webservers
  vars:
    http_port: 80
```

- From a separate YAML file next to "playbook":

```
- hosts: mailservers
  vars_files:
  - /vars/external_vars.yml
```

- As magical a variable, like `hostvars`, `group_names`, `groups`, etc.

Defining Variables

- From “facts” — parameters collected on remote boxes:
  1. From Ansible tool itself.
  2. From INI, JSON, or an executable returning a JSON from “local facts” directory, usually `/etc/ansible/facts.d`
- From the command line:

```
ansible-playbook foo.yml \
  --extra-vars "some_var=1.23.45 other_var=foo" \
  --extra-vars '{"foo":"bar","baz":[1, 42, 3.1415]}' \
  --extra-vars "@some_file.json"
```

Accessing Variables

- Inlined Jinja2 “filter” expressions:

```
- name: touch files with an optional mode
  file: "dest={{item.path}} state=touch
         mode={{item.mode|default(omit)}}"
  with_items:
    - path: /tmp/foo
    - path: /tmp/bar
      mode: "0444"
```

- Accessed from Jinja2 templates:

```
{% if (inventory_hostname in groups.lbservers) %}
-A INPUT -p tcp  --dport {{ listenport }} -j  ACCEPT
{% endif %}
```

- *But how is the scope resolved?*

“Facts”

- Facts can be turned off:

```
- hosts: whatever
  gather_facts: no
```

- Can be modified in runtime:

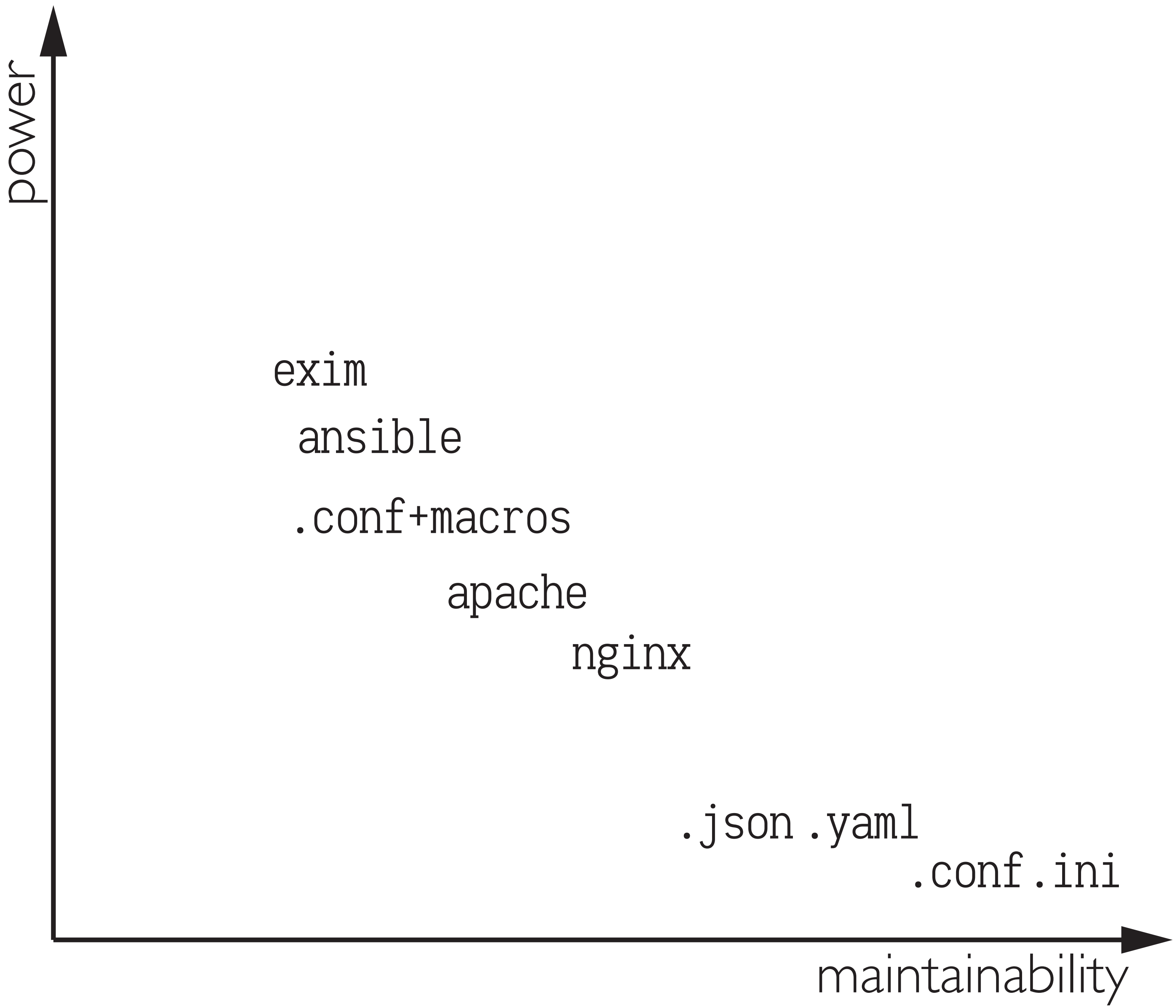
```
- hosts: webservers
  tasks:
    - name: 'create directory for ansible custom facts'
      file: state=directory recurse=yes path=/etc/ansible/facts.d
    - name: 'install custom impi fact'
      copy: src=ipmi.fact dest=/etc/ansible/facts.d
    - name: 're-read facts after adding custom fact'
      setup: filter=ansible_local
```

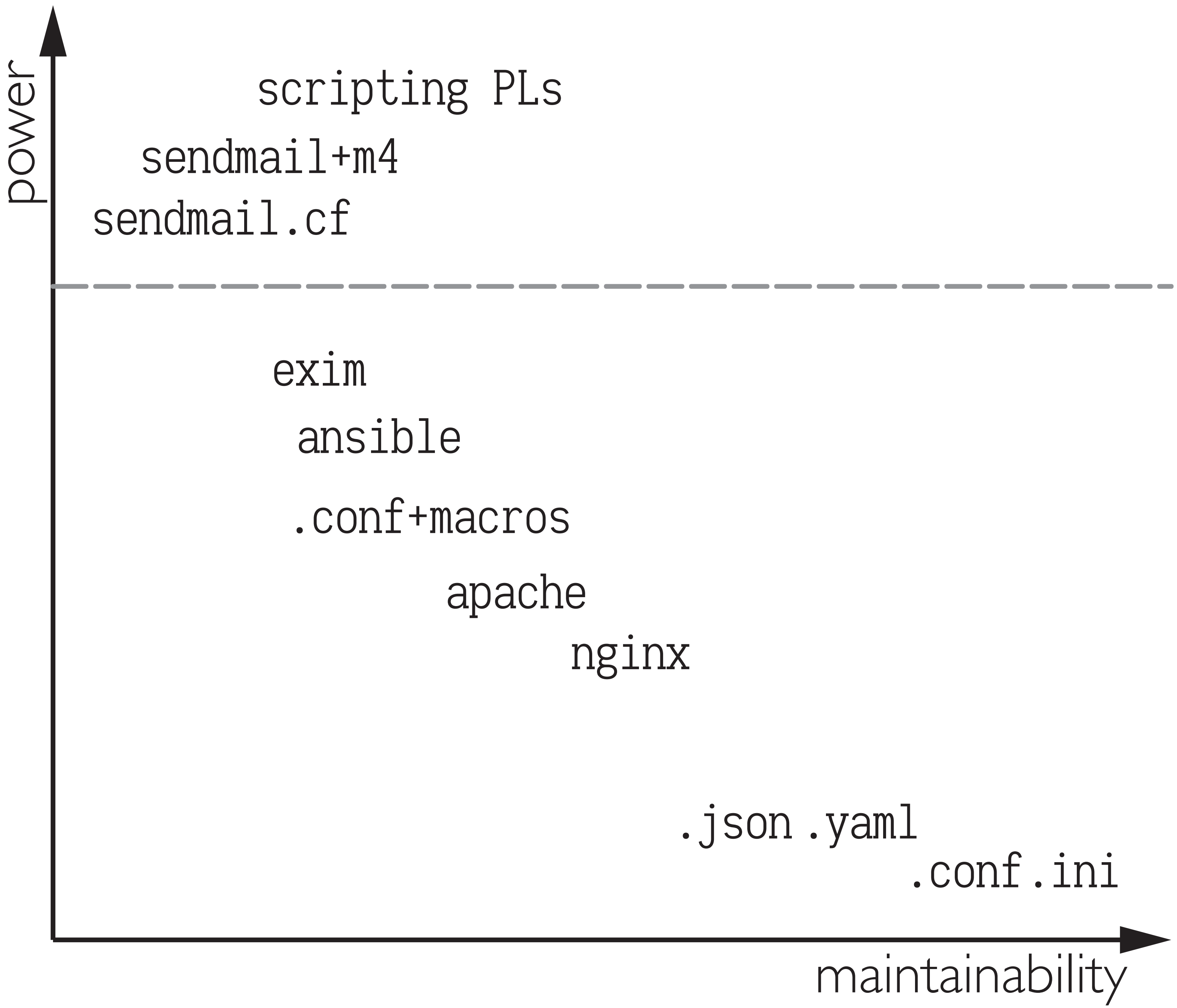
Loops

- Over lists, nested. Or zipping two lists together.
- Over dicts.
- Over file's contents.
- Over “fileglobs”.
- Storing results of each iteration to “register”.
- With 3 second pauses between iterations.
- *Completely different syntax in each case!*

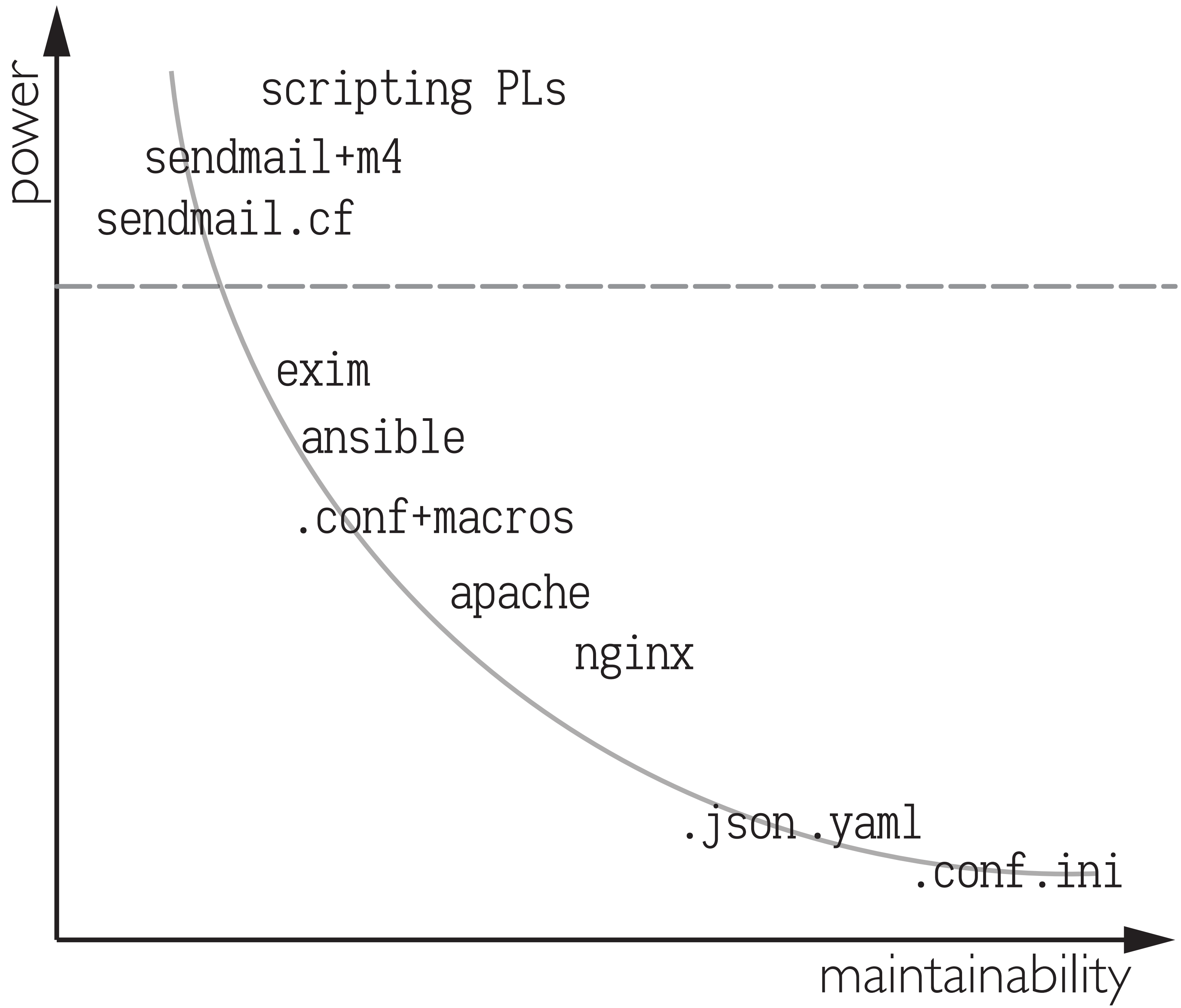

NORTD LABS 2014

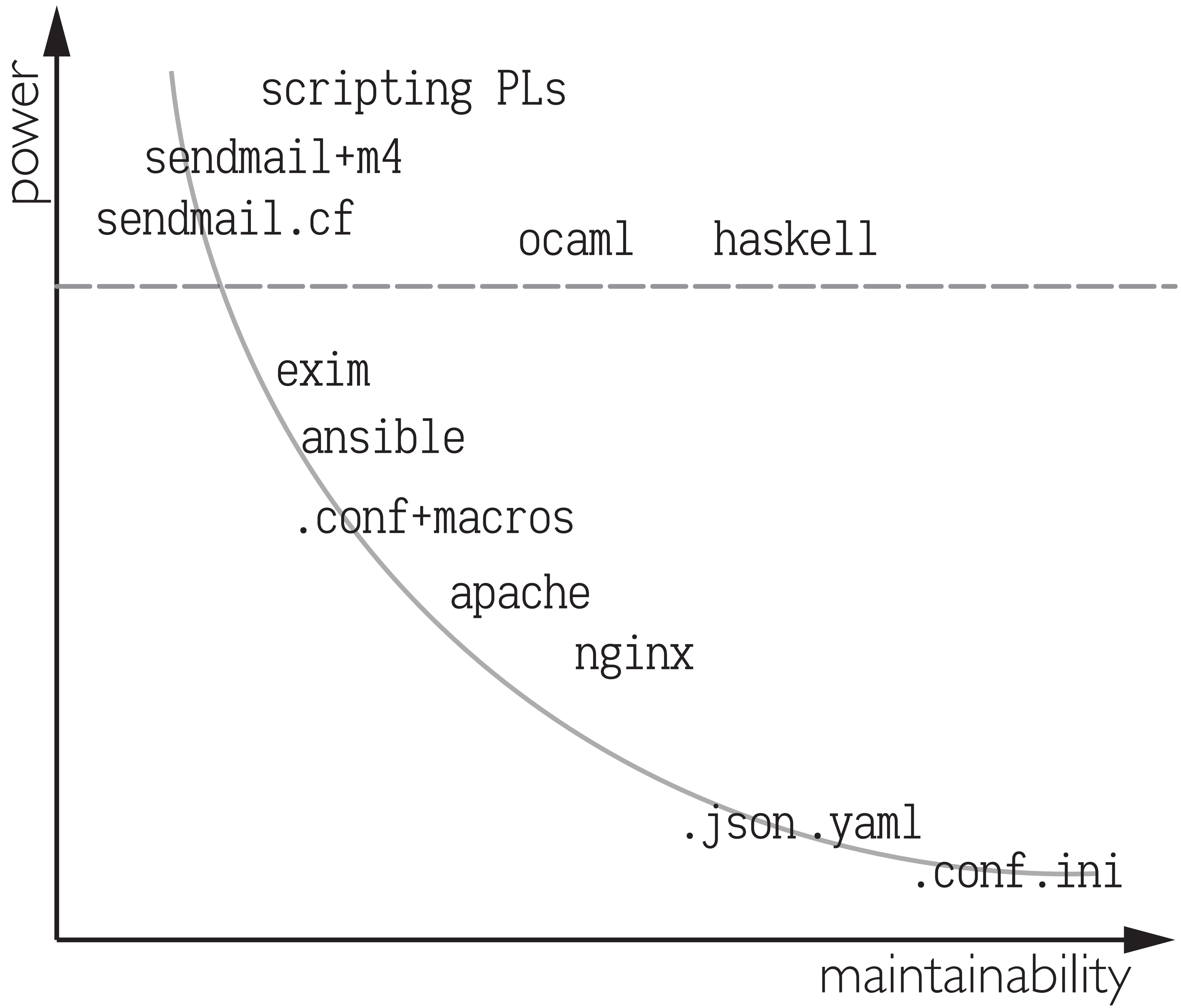

power
.json .yaml
.conf .ini
maintainability

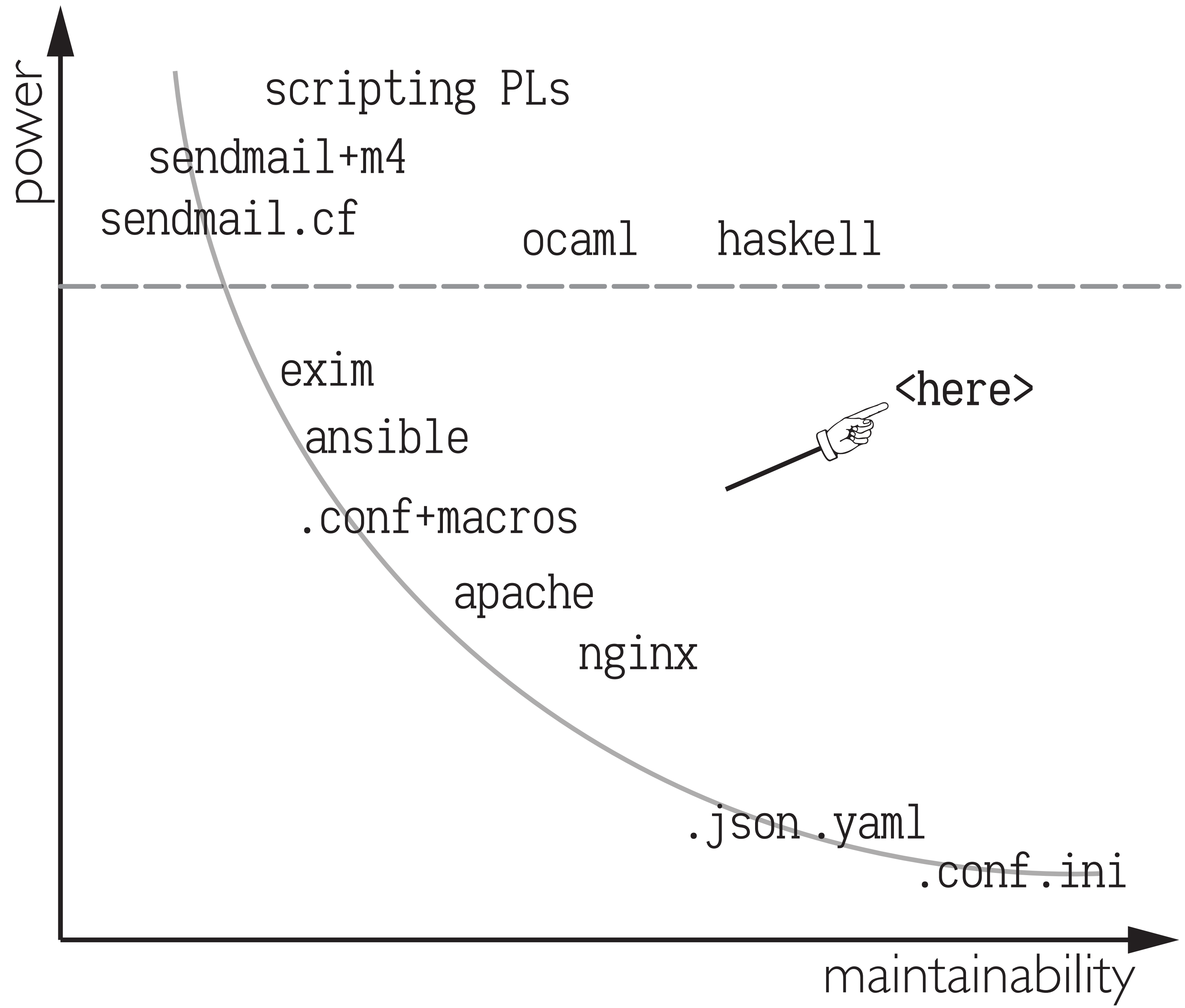

power
exim
ansible
.conf+macros
apache
nginx
.json .yaml
.conf .ini
maintainability



power
scripting PLs
sendmail+m4
sendmail.cf
exim
ansible
.conf+macros
apache
nginx
.json .yaml
.conf .ini
maintainability


power
scripting PLs
sendmail+m4
sendmail.cf
exim
ansible
.conf+macros
apache
nginx
.json .yaml
.conf .ini
maintainability


power
scripting PLs
sendmail+m4
sendmail.cf
ocaml
haskell
exim
ansible
.conf+macros
apache
nginx
.json
.yaml
.conf
.ini
maintainability


power
scripting PLs
sendmail+m4
sendmail.cf
ocaml
haskell
exim
<here>
ansible
.conf+macros
apache
nginx
.json
.yaml
.conf
.ini
maintainability

Configuration DSLs

- Complicated (and ad-hoc) semantics.
- Documentation not always good.
  Almost never for in-house software.
- Poor discoverability.
  What parameters are expected over there?
- Poor tooling.
  How do I check this config before I deploy it?
- Prone to leak software implementation details to the DSL.

Can we do better?

Better predictability

Better discoverability

Better refactorability

λ: The Ultimate `.conf`

$\boldsymbol{\lambda}^{\to}$ a.k.a. Simply typed lambda calculus

Hindley-Milner type system for polymorphism and type inference

“Extensible records with scoped labels” by D. Leijen (2005)

Properties

- β- and δ-contraction, using capture-avoiding substitution:

$$\beta\textsf{contr}\ \frac{}{(\lambda x.\ t)s \ \longmapsto\ t[s/x]} \qquad \delta\textsf{contr}\ \frac{}{\textbf{let}\ x = s\ \textbf{in}\ t \ \longmapsto\ t[s/x]}$$

- Type safety:

$$\textsf{preserv.}\ \frac{\Gamma \vdash e : \tau \quad e \ \longmapsto^{*}\ e'}{\Gamma \vdash e' : \tau} \qquad \textsf{progress}\ \frac{\Gamma \vdash e : \tau}{e\ \textsf{val} \land \exists e'.\ e \ \longmapsto\ e'}$$

- Strong normalization:

$$\textsf{norm}\ \frac{e \not\longmapsto}{e\ \textsf{normal}} \qquad \textsf{wn}\ \frac{e \longmapsto e' \quad e' \not\longmapsto}{e \in \mathcal{WN}} \qquad \textsf{sn}\ \frac{\{e' | e \longmapsto e'\} \subseteq \mathcal{SN}}{e \in \mathcal{SN}}$$

Profit

- Lightweight syntax with optional type annotations.
  Built-in number, string, list, record, and variant types.
- Safe abstractions.
  Lambdas; no name capture; no macros; lexical scope.
- Static and strong typing.
  Consistency checks before deploying; no typing surprises.
- Totality and purity.
  Always terminate; no exceptions; no launching missiles.
- Strong normalization.
  Indirections and abstractions simplified mechanically.

Abstract Syntax

$$
\begin{array}{llll}
\text{Terms } t & ::= & c & \text{(constants)} \\
& \mid & x & \text{(variables)} \\
& \mid & \textbf{let}\ x\ =\ t\ \textbf{in}\ t & \text{(let bindings)} \\
& \mid & \lambda x.\ t & \text{(}\lambda\text{ abstraction)} \\
& \mid & t\ t & \text{(fun. application)} \\
\\
\text{Constants } c & ::= & \{\ell = \_ \mid \_\} & \text{(record extend)} \\
& \mid & \_.\ell & \text{(record select)} \\
& \mid & \_ - \ell & \text{(record restrict)} \\
& \mid & \ell\ \_ & \text{(variant inject)} \\
& \mid & \textbf{case}\ \_\ \textbf{of}\ \{\ell\ x \to \_, \ldots\} & \text{(variant decompose)} \\
& \mid & \ldots &
\end{array}
$$

Abstract Syntax

$$
\begin{array}{llll}
\text{Constants } c & ::= & \ldots & \\
& \mid & [\,] \;\mid\; \_ :: \_ & \text{(list constructors)} \\
& \mid & \diamond \;\mid\; \text{head} \;\mid\; \text{tail} \;\mid\; \text{fold} & \text{(list combinators)} \\
& \mid & \mathtt{\textbf{string}} \;\mid\; \mathtt{\textbf{bool}} \;\mid\; \mathtt{\textbf{int}} & \\
& \mid & \mathtt{\textbf{nat}} \;\mid\; \mathtt{\textbf{float}} \;\mid\; \ldots & \text{(primitive literals)} \\
& \mid & \_ \mathbin{++} \_ & \text{(string concatenation)} \\
& \mid & \neg \;\mid\; \vee \;\mid\; \wedge & \text{(bool operators)} \\
& \mid & \textbf{if}\ \_\ \textbf{then}\ \_\ \textbf{else}\ \_ & \text{(branch)} \\
& \mid & + \;\mid\; - \;\mid\; * & \text{(int operators)} \\
& \mid & +_{\mathbb{N}} \;\mid\; *_{\mathbb{N}} \;\mid\; \text{iterate} & \text{(nat combinators)}
\end{array}
$$

Abstract Syntax

$$
\begin{array}{llll}
\text{Kinds } \kappa & ::= & \star & \text{(star kind)} \\
& \mid & \mathsf{row} & \text{(row kind)} \\
& \mid & \kappa_1 \to \kappa_2 & \text{(arrow kind)} \\
\\
\text{Type Schemes } \sigma & ::= & \forall \alpha^{\kappa}.\ \sigma & \text{(polytypes)} \\
& \mid & \tau^{\star} & \text{(monotypes)}
\end{array}
$$

Abstract Syntax

Types $\tau^\kappa,\ \rho^{\mathsf{row}}$	$::=$	$\mathtt{B}$	$: \star$	(base types)
	$\mid$	α^κ	$: \kappa$	(type variables)
	$\mid$	$\tau^\star \to \tau^\star$	$: \star \to \star \to \star$	(functions)
	$\mid$	$[\tau^\star]$	$: \star \to \star$	(lists)
	$\mid$	$\varnothing$	$: \mathsf{row}$	(empty row)
	$\mid$	$(\!\mid \ell : \tau^\star \mid \rho^{\mathsf{row}} \mid\!)$	$: \star \to \mathsf{row} \to \mathsf{row}$	(row extend)
	$\mid$	$\{\rho^{\mathsf{row}}\}$	$: \mathsf{row} \to \star$	(records)
	$\mid$	$\langle\rho^{\mathsf{row}}\rangle$	$: \mathsf{row} \to \star$	(variants)

$$
\text{Base types } \mathtt{B} \ ::= \ \mathtt{String} \mid \mathtt{Bool} \mid \mathtt{Int} \mid \mathtt{Float} \mid \mathtt{Nat}
$$

Motivating Example

Motivating Example

```
[ { host     = "db.example.com",
    port     = 5433,
    user     = "alice",
    password = Plain "some_secret",
    dbname   = "foobar"
  }
]

-- : [ { host     : String
--     , port     : Int
--     , user     : String
--     , password : <Plain : String | r>
--     , dbname   : String
--     | s
--     } ]
```

Motivating Example

```
alice_db_connection = {
    host = "db.example.com",
    port = 5433,
    user = "alice",
    password = Plain "some_secret",
    dbname = "foobar"
  }

bob_db_connection = {
    host = "db.example.com",
    port = 5433,
    user = "bob",
    password = Plain "other_secret",
    dbname = "foobar"
  }

[ alice_db_connection, bob_db_connection ]
```

Motivating Example

```
default_pgsql_connection = {
    host = "db.example.com",
    port = 5433,
    dbname = "foobar"
  }
alice_db_connection = {
    user = "alice",
    password = Plain "some_secret"
  | default_pgsql_connection
  }
bob_db_connection = {
    user = "bob",
    password = Plain "other_secret",
    host = "db2.example.com"  -- ← record restrict + record extend
  | default_pgsql_connection
  }
[ alice_db_connection, bob_db_connection ]
```

Motivating Example

```
default_pgsql_connection = {
    host = "db.example.com",
    port = 5433,
    dbname = "foobar",
    password = Ask
  }
alice_db_connection = {
    user = "alice",
  | default_pgsql_connection
  }
bob_db_connection = {
    user = "bob",
    host = "db2.example.com",
    missing_parameter = "WAT"     -- cannot unify bob and alice!
  | default_pgsql_connection
  }
[ alice_db_connection, bob_db_connection ]
```

Modules

```
-- DB/PostgreSql.conf:
module DB.PostgreSql
default_pgsql_connection = {
    host = "db.example.com",
    port = 5433,
    dbname = "foobar",
    password = Ask
  }

-- Main.conf:
import DB.PostgreSql

alice_db_connection =
  { user = "alice" | default_pgsql_connection }
bob_db_connection =
  { user = "bob"   | default_pgsql_connection }

[ alice_db_connection, bob_db_connection ]
```

Desugared Modules

```
DB.PostgreSql = {
  default_pgsql_connection = {
      host = "db.example.com",
      port = 5433,
      dbname = "foobar",
      password = Ask
    }
  }

alice_db_connection =
  { user = "alice" | DB.PostgreSql.default_pgsql_connection }

bob_db_connection =
  { user = "bob"   | DB.PostgreSql.default_pgsql_connection }

[ alice_db_connection, bob_db_connection ]
```

Type Declarations and Annotations

```
-- DB/PostgreSql.conf:
type Conn =
  { host     : String
  , port     : Int
  , user     : String
  , password : <Plain : String, Ask : ()>
  , dbname   : String
  }

-- Main.conf
import DB.PostgreSql  (Conn, default_pgsql_connection)

alice_db_connection : Conn
  = { user = "alice" | default_pgsql_connection }

bob_db_connection : Conn
  = { user = "alice" | default_pgsql_connection }

[ alice_db_connection, bob_db_connection ] : [ DB.PostgreSql.Conn ]
```

Type Holes

```
import DB.PostgreSql (Conn, default_pgsql_connection)

alice_db_connection : Conn
  = { user = "alice"
    , password = ?password        -- ← type hole
    | default_pgsql_connection
    }

[ alice_db_connection ] : [ DB.PostgreSql.Conn ]

-- Main.conf:5:17:
--   Type hole “password” has type:
--     < Plain : String
--     , Ask   : ()
--     >
```

Type Declarations

```
type Length = Double

type Point = {x : Double, y : Double}

type Direction = <Left : (), Center : (), Right : ()>

type Maybe a = <Nothing : (), Just : a>

-- No inductive types
type Nat n = <Zero : (), Succ : Nat n>
--                              ^^^
```

Functions

```
double = fun n → n * 2
-- double : Int → Int

id = fun a → a
-- id : forall a. a → a

maybe : forall a b. (a → b) → b → <Nothing : (), Just : a> → b
  = fun f x m →
      case m of
        Nothing () → x,
        Just y     → f y

map : forall a b. (a → b) → [a] → [b]
  = f xs → fold (fun a bs → (f a :: bs)) [] xs
```

Recursion: primitive

```
-- No general recursion
factorial : Int -> Int
  = fun n -> if n > 1 then n * factorial (n - 1) else 1
--                             ^^^^^^^^^

-- Built-in recursor for Nats
iterate : forall a. (Nat -> a -> a) -> a -> Nat -> a

factorial : Nat -> Nat
  = iterate (fun a b -> a * b) 1

odd : Nat -> Bool
  = fun n -> if n == 0 then False else not (odd (n-1))
--                                          ^^^

odd = fun n -> iterate (fun _ isOdd -> not isOdd) False n
```

FP in Configs

- Nix: “Purely functional package manager”
- Fugue's Ludwig: DSL for cloud infrastructure configuration
- Gabriel Gonzalez's Dhall: minimalistic System Fω-based configuration language
- Jane Street uses OCaml for configs
- XMonad, Yi, …: Haskell for configs

Q&A

Thanks!