Configuration Pinocchio
The Lies Plainly Seen and the Quest to be a Real Discipline

Andre Masella

May 15, 2015
Overview

- Where are we?
- SRSLY?
- How do we get out?
- Are we there yet?
When did configs get complicated?

- In The Before Times, usually software was installed once and left alone.
- Cloud and cluster computing means running the same things many ways.
- Application have spread out beyond the binary (e.g., into the database).
- Testing is no longer possible by starting the binary on a workstation.
Why are configs complicated?

- Complexity is semantic and independent of the format.
- There is a conflict between terse (easy to write) and explicit (easy to read).
- Macro languages add additional behaviour.
- There are strange embedded programming languages.
What is a Configuration?

- Configurations are usually generated by an *ad hoc* process where constants in the code are externalised.
- This including a configuration file, command-line arguments, built-time arguments, environment variables, and information in a database.
- The format doesn’t really matter. You can transform any of those to a “config file”.

Andre Masella
Configuration Pinocchio
Terseness is achieved two ways: macros and default propagation.

If the macro language is separate from the binary, the expansion can be observed. Often, it is built-in.

Default propagation works in many ways:
- elided parameters in parts of the configuration
- values that override (in the config, environment, or on the command line)
Survey of Configurations

- Examined common servers’ configurations: Apache, NGINX, Samba, Asterisk, Make, BIND, and CUPS.
- Looked at default propagation mechanisms and found:
  - local stanza, global stanza, binary default (implicit)
  - global stanza, local stanza, binary default (implicit)
  - local stanza, template stanzas, binary default (explicit)
  - hybrid (Apache)
- macro languages and embedded programming languages
Survey of Configurations – Highlights

- Apache can change most of the configuration based on the query (e.g., different security depending on the browser).
- Apache and NGINX’s rewriting rules are Turing complete! Roland Illig has implemented a Towers of Hanoi solver.
- BIND has different default propagation schemes depending on the parameter.
- BIND also has a rewrite system, though not Turing complete.
- Make has contextually-determined lazy or eager evaluation.
- Asterisk does GoTo via string bashing.
Weird Machines

- If Apache’s URL rewriting can be Turing complete, then every incoming URL is a program executed by the `mod_rewrite` virtual machine.
- URLs are a strange byte-code for a *weird machine*.
- Weird machines are targets for exploitation.
- Many configurations define weird machines for the queries in the binary.
- Since they are Turing complete, they can’t be checked for correctness.
- If they do things like Asterisk, where we compute jump targets from user data, that’s scary.
Composability

► Our configurations lack composability, which is what cloud configurations demand.
► There should be a configuration that configures a server for running on metal and in Docker.
► We already have composition for some servers. LDAP queries as embedded in configurations.
► String bashing needs to go away.
Solution: Divide and Conqueror

- Separate the macro language and default propagation from the binary.
- Make EPLs less weird and either using existing languages or more byte-code-like interfaces.
- Make a configuration language that can handle the composition easily.
- Creating a unified configuration format is not worth doing.
Not-So-Weird Machines

- Replace weird machines with existing scripting languages (e.g., Guile, Lua, FORTH, GameMonkey, TCL, JavaScript).
- There are many obscure programming languages that few people know; when you create a new programming language, you can be guaranteed that no one will know it. – K. Schaffrick
- If you really need a machine, make it virtual, not weird.
Byte Code and Virtual Machine

- Replacing weird machines with byte-code will be easier to:
  - implement (easy to parse, easy to build a simple VM)
  - optimise (convert to LLVM/JVM/CLR)
  - verify and secure
  - debug (crash and dump the machine state)
  - specify

- People can also build good tools on top of it.
Configuration Languages

A language for configurations needs:

- have a sensible default propagation policy
- composability in the face of multiple binaries and formats
- be semantically meaningful
- the features of normal languages (e.g., types, debugging, libraries)
Why not use traditional languages?

- They aren’t very good at it!
- Imperative languages make determining data flow our problem, but we don’t care.
- Functional ones require us to know lots about the format of our data, which we don’t know and will change.
- They are concerned about I/O and a whole bunch of other things that are unhelpful for configurations.
# Existing Configuration Languages

<table>
<thead>
<tr>
<th></th>
<th>Coil</th>
<th>Flabbergast†</th>
<th>HOCON</th>
<th>Jsonnet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Functional</td>
<td>Functional</td>
<td>Imperative*</td>
<td>Functional</td>
</tr>
<tr>
<td>Side-effect Free</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Prototype</td>
<td>Prototype</td>
<td>Prototype</td>
<td>Prototype</td>
</tr>
<tr>
<td>Typing Strength</td>
<td>Weak</td>
<td>Strong</td>
<td>Weak</td>
<td>Strong</td>
</tr>
<tr>
<td>Typing Enforcement</td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Schema Validation</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Turing Complete</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Scoping</td>
<td>Lexical</td>
<td>Dynamic</td>
<td>Lexical</td>
<td>Lexical</td>
</tr>
<tr>
<td>Default Propagation</td>
<td>Inheritance</td>
<td>Scope, inheritance</td>
<td>Inheritance</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Output Format</td>
<td>Python objects</td>
<td>Text, Custom</td>
<td>Java, Python, or Ruby objects</td>
<td>JSON</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>NixOS</th>
<th>Pan</th>
<th>Pystachio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Functional</td>
<td>Imperative</td>
<td>Imperative</td>
</tr>
<tr>
<td>Side-effect Free</td>
<td>Yes</td>
<td>No</td>
<td>Hybrid*</td>
</tr>
<tr>
<td>Inheritance</td>
<td>None</td>
<td>Class-based</td>
<td>Class-based</td>
</tr>
<tr>
<td>Typing Strength</td>
<td>Strong</td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Typing Enforcement</td>
<td>Dynamic</td>
<td>Hybrid*</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Schema Validation</td>
<td>None</td>
<td>Assignment</td>
<td>Request</td>
</tr>
<tr>
<td>Turing Complete</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Scoping</td>
<td>Lexical</td>
<td>Lexical</td>
<td>Hybrid*</td>
</tr>
<tr>
<td>Default Propagation</td>
<td>Operator</td>
<td>Inheritance</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Output Format</td>
<td>Java objects</td>
<td>JSON, XML</td>
<td>Python objects</td>
</tr>
</tbody>
</table>

* Depends on context. † Mine.
Existing Configuration Languages

- Most are functional.
- Most use prototype inheritance.
- More dynamic scoping than expected.
- Some schema validation.
- All are very immature.
Conclusion

- Stand back and decide what your config files *really* are.
- Stop the weird machines.
- Delegate the configuration manipulation to a configuration language.
Thanks

- Kyle W. Schaffrick, Google, Inc.
- Dr. Gráinne Sheerin, Google, Inc.
- Dr. Dan G. Brown, University of Waterloo
- James L. Schofield, Couch Labs, Inc.