Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

AUGEAS—a configuration API

David Lutterkort
Red Hat, Inc.
dlutter@redhat.com

Abstract

One of the many things that makes Linux configuration management the minefield we all love is the lack of a local configuration API. The main culprit for this situation, that configuration data is generally stored in text files in a wide variety of formats, is both an important part of the Linux culture and valuable when humans need to make configuration changes manually.

AUGEAS provides a local configuration API that presents configuration data as a tree. The tree is backed directly by the various config files as they exist today; modifications to the tree correspond directly to changes in the underlying files. AUGEAS takes great care to preserve comments and other formatting details across editing operations. The transformation from files into the tree and back is controlled by a description of the file's format, consisting of regular expressions and instructions on how to map matches into the tree. AUGEAS currently can be used through a command line tool, the C API, and from Ruby, Python, and OCaml. It also comes with descriptions for a good number of common Linux config files that can be edited "out-of-the-box."

1 Introduction

Configuration management of Linux[1] systems is a notoriously thorny subject. Problems in this space are numerous, from making large numbers of machines manageable by mere mortals, to the sheer mechanics of changing the configuration files of a single system programmatically. What makes the latter so difficult is the colorful variety of configuration file formats in common use, which has historically prevented any form of system-wide configuration API for Linux systems.

AUGEAS lays the foundation for such an API by focusing on the most basic and mundane task in this area: changing configuration files in a way that abstracts away the formatting details that are irrelevant to programmatic configuration changes. While formatting details may be irrelevant in this context, they are still important, and AUGEAS goes through a lot of trouble to preserve comments, whitespace, etc.

Logic for configuration changes is embedded in many tools, and the same logic is reinvented multiple times, often for no other reason than a difference in implementation language. As an example, `Webmin` [**?**] can edit a wide variety of configuration files; that editing logic, and the hard work for writing and maintaining it, is of no use to other configuration tools like `Puppet`, `Bcfg2`, or `cfengine`, since none of them is written in Perl. The prospect of reimplementing large parts of this logic in Ruby for use by `Puppet` was so unappealing that it became clear that a language-agnostic and tool-independent way to achieve this had to be found.

The lack of a simple local configuration API also prevents building higher-level services around configuration changes. Without it, there is no way to set system-wide (or site-wide) policy for such changes that can be enforced across multiple tools, from simple local GUI tools to remote administration capabilities.

AUGEAS tackles this problem in the simplest possible way and focuses solely on the mechanics of modifying configuration files.

2 Design

There is no shortage of attempts to simplify and unify Linux system configuration. Generally, they fall into one of three categories: *keyhole* approaches targeted at one specific purpose; *greenfield* approaches to solve modifying configuration data once and for all; and *templating*, popular in homegrown config management systems when plain file copying becomes unsatisfactory, and often used as a middle ground for the first two.

[1] Most of this paper applies to any Unix-like system, though we will only talk about Linux here.

A careful look at these three types of approaches was very instructive in setting AUGEAS' design goals, and determining what exactly it should do, and, even more importantly, what it should *not* attempt to do.

2.1 Keyhole Approaches

The most direct and obvious approach to scripting configuration changes is to use simple text editing tools like `sed` and `awk` or the equivalent facilities in the scripting language of choice. Since changing configuration data is usually only part of a larger task, like writing an installer, the resulting scripts are good enough for their purpose, but not general enough to be of use in other situations, even if language barriers are not a concern.

All popular open-source config management systems follow this approach, too, resulting in unnecessary duplication of logic to parse and write configuration files, and a healthy mix of common and unique bugs in that logic. Even seemingly simple file formats hold surprises that make it all too easy to write a parser that will fail to process all legal files for that format correctly. As an example, one simple and popular format uses setting of shell variables in a file sourced into a larger script. Since comments in shell scripts start with a `#` and extend to the end of the line, a parser of such files should strip this pattern from every line it reads before processing it further. Unless, of course, the `#` appears inside a quoted string. But even that will trip up on unquoted uses of `#` that do not start a comment, such as `V=x#y`.

2.2 Greenfield Approaches

Recognizing that the state of the art of modifying Linux/Unix configuration is less than ideal, various proposals have been put forward to improve it, such as Elektra and Uniconf. Since the variations in config file formats are the biggest roadblock to treating configuration data in a unified manner, they generally start by proposing a new scheme for storing that data. The exact storage scheme varies from project to project, from LDAP to relational databases and files in their *own* favorite format. Such an approach is of course unrealistic, since it requires that the upstream consumers of configuration data modify their applications to use the new API. From the perspective of an upstream maintainer, such changes are without reward while the API is new and unproven. And the only way to prove that an API is worth upstream's effort is to get upstream projects to use it.

A side effect of introducing completely new storage for configuration data is that the configuration files that administrators are used to, and that a multitude of tools knows about, are either no longer used at all, or are no longer the authoritative store for configuration data. This is undesirable, as system administrators have to get used to a whole new way of making local configuration changes, and scripts have to be changed to use the new configuration API.

Greenfield approaches generally also aim much higher than just modifying configuration data. It is tempting to model other aspects of configuration data and build more capabilities into the new unified API, ranging from fine-grained permissioning schemes to hiding differences between Linux distributions and Unix flavors. Each of these addresses a problem that is hard in its own right, often not just because of technical difficulties, but also because modeling it in a way that is suitable for a wide variety of uses is hard.

2.3 Templating

Templating, just as the greenfield approaches, introduces a new "master" store for all configuration data, which makes it impossible to change it in its "native" location, either manually (for example, during an emergency), or with other programs than the template engine.

2.4 Design Goals

With the successes and limitations of these approaches in mind, AUGEAS focuses on the problem at the heart of all of them: editing configuration files programmatically. Above all else, AUGEAS limits its scope to a handful of goals around that task.

As we have seen for greenfield approaches, it is unlikely that the current situation of configuration data scattered across many files in many formats can be addressed by radically breaking with history and custom. At the same time, as shown by templating approaches, the penalty for generating these files from a new authoritative source is high, and rarely ever appropriate. AUGEAS therefore uses the existing config files as its sole store of configuration data and does not require additional data stores.

The multitude of producers and consumers of configuration data makes it imperative that AUGEAS be useful without the support of these producers and consumers. In other words, AUGEAS must be useful without any changes to other code which handles configuration data—in particular, without any support from the primary users of that data like system daemons. Similarly, there are a vast number of tools that modify configuration data, and it should be possible to use these tools side-by-side with AUGEAS. As a consequence, AUGEAS should not rely on such tools preserving any AUGEAS-specific annotations (for example, in comments), while making sure that such annotations added by other tools are preserved across edits with AUGEAS.

We would like AUGEAS to handle as wide a variety of configuration files as possible. Since every format needs some form of intervention by a person, AUGEAS should make it as easy as possible to describe file formats, not only in terms of the notation of the description, but also in the checks that it can perform to spot errors in the description.

How a change to the tree is translated into a change in the underlying file should be intuitive, and should correspond to a reasonable expectation of "minimal" edits. For example, changing the alias of one host in `/etc/hosts` should only lead to a change on the line containing the host entry, and leave the rest of the file untouched.

Finally, configuration changes have to be made in many situations and with tools written in many languages. AUGEAS therefore must be "language neutral" in the sense that it can be used by the widest variety of programming languages. In practice, this means that AUGEAS has to be implemented in C. Furthermore, the public API relies solely on strings as data types, where some strings denote paths in the tree.

3 Using AUGEAS to change files

AUGEAS can be used in a number of ways: a C library API, the `augtool` shell command, and from a number of other programming languages. Currently, bindings are available for Python, Ruby, and OCaml. The following discusses the usage of `augtool`, which closely mirrors the other interfaces.

3.1 The tree and path expressions

In the tree that AUGEAS maintains, each node consists of three pieces of information: a string *label* that is part of the path to the node and all its children, an optional string *value*, and a list of child nodes. Since files are inherently ordered data structures, the AUGEAS tree is also ordered; in particular, the order of siblings matters. Multiple siblings can have the same label—for example, a host entry in `/etc/hosts` can have multiple aliases, each of which is stored in a separate `alias` node.

Because of this structure, AUGEAS' tree is conceptually similar to an XML parse tree. When multiple siblings have the same label, it is of course necessary to distinguish between them. For that, and for simple searches, AUGEAS adapts some of the conventions of XPath [2]. In particular, a label by itself in a path, for example, `alias/`, matches *all* children of a node with label `alias`. The *n*-th child with that label can be picked out with `alias[n]`, and the last such child, with the special notation `alias[last()]`.

Wildcard searches using `*` as a path component are also supported. The path `/p/*/g` matches all grandchildren with label `g` of the node `p`. Searches with `*` are not recursive, and the above pattern does not match a node `p/a/b/g`.

3.2 Tree manipulation

When `augtool` starts, it reads *schema* descriptions out of a set of predefined directories, and parses configuration files according to them. The result of this initialization is the tree that is presented to the user. The user can now query the tree, using `match` to search nodes that match a certain path expression, and `get` to retrieve the value associated with a node.

The tree is modified using `ins` to insert new nodes at a specific position in the tree—for example, to insert another `alias` node after the first such node, and `rm` to delete nodes and whole subtrees. The value associated with a node can be changed with `set`.

Files are not modified while the tree is being changed, both so that files with possibly invalid entries are not produced while multi-step modifications are under way, and to enable more extensive consistency checks when files are finally written. Writing of files is initiated with

the `save` command, which writes new versions of all files whose tree representation has changed; files that have no modifications made to them are not touched.

What files are written, and how the tree is transformed back into files, are again governed by the schemas that `augtool` read on startup. Schemas are written in a domain-specific language, and the primitives of the language ensure that the transformation from file to tree and the reverse transformation from tree to file match and follow certain consistency rules designed to make the round trip from file through tree to modifed file safe and match users' expectations. The mechanisms performing the transformation need to know two pieces of information: which files to transform, and how to transform them. The first is given through a file name filter, described as shell globs specifying which files to include or exclude; the second is done by writing a *lens* that is applied to the contents of each file matching the filter.

4 Lenses and bidirectional programming

AUGEAS needs to transform file contents (strings) into a tree and that tree back into file contents. Rather than having users specify these two mappings separately, and running the risk that they might not be compatible with one another, AUGEAS uses the idea of a *lens* to combine the two mappings in a way that guarantees their compatibility in a very precise sense.

The term *lens* was coined by the `Harmony` project [4], and originates from the "view update" problem: given a concrete view of data (configuration files in AUGEAS' case) and an abstract view of the same data (the tree), construct suitable mappings between the two views that translate changes to the abstract view into intuitively minimal changes of the concrete data. Generally, the mapping from concrete to abstract view leaves out some information, for example, formatting details or comments that are of no use in the abstract view. Conversely, the mapping from abstract to concrete view must restore that data. With that, lenses are not bijective mappings between the concrete and the abstract domains: multiple concrete views, namely all the ones that differ only in unimportant details such as formatting, map to the same abstract view. `Harmony` uses lenses to construct mappings between tree-structured data [3], for example, for synchronization of calendar files that essentially contain the same information, but use different XML-based formats. Similarly, `Boomerang` [1] performs mappings between unstructured text data. AUGEAS uses the same approach for its mapping between text data and trees.

Formally, a lens consists of two functions, *get* and *put*.[2] If C is the set of all concrete data structures (in AUGEAS' case, strings), and A is the set of all abstract data structures (in AUGEAS' case, trees), a lens l consists of

$$l.get : C \to A$$
$$l.put : A \times C \to C$$

The *get* function is used to transform concrete views into abstract ones. The *put* function, which maps abstract views back to concrete views, receives the original concrete view as its second argument and consults that to restore any information left out by the *get* direction. The two directions are tied together by two requirements that express intuitive notions of how lenses should behave when we make a roundtrip from concrete to abstract view and back: for every $c \in C$ and $a \in A$, a lens l must fulfill

$$l.put\ (l.get\ c)\ c = c \qquad \text{(GETPUT)}$$
$$l.get\ (l.put\ a\ c) = a \qquad \text{(PUTGET)}$$

Put into words, GETPUT expresses that transforming a string c into a tree, and then transforming the unmodified tree back into a string, should yield the string c we started with, and ensures that the *put* direction restores any information not captured in the tree faithfully when the tree is not modified. The PUTGET law states that transforming any tree back into a string using an arbitrary concrete string c as the second argument to *put* and transforming the result back into a tree must yield exactly the tree we started with, limiting how *put* uses its second argument c when transforming a tree: it can only use it for those parts of a string that are abstracted away by the *get* direction.

The lens laws are weaker than requiring that *get* and *put* be inverses of one another. That would require that both be bijective, and keep lenses from doing what makes them so useful in practice: abstracting away unimportant information like comments or how many spaces are used to separate two values. Since all lenses contain a *get* and *put* direction that are compatible in the sense laid

[2] Strictly speaking, there is a third function, *create*, involved to create new concrete data from abstract data alone; since we are not proving anything about lenses here, there's no need to distinguish between *put* and *create*.

down by the lens laws, building complex lenses from simpler ones is called *bidirectional programming*, since every lens expresses how to get from input *c* to output *a*, and at the same time, how to get from an output *a* back to an input *c*.

Lenses are built from simple builtin lenses, called *lens primitives*, by combining them with a few builtin *lens combinators*. This mechanism of building complex lenses from simpler ones forms the backbone of AUGEAS' schema descriptions. The two lens laws, GETPUT and PUTGET, restrict how lenses can be combined; in AUGEAS, these restrictions are enforced by the typechecker described below.

Typically, a complex lens that describes the processing of a whole file is broken up in smaller lenses, each of which processes a small portion of the file, for example the aliases of a host in `/etc/hosts`. To support this mode of working, where complex lenses are gradually built from simpler ones, AUGEAS has a builtin unit test facility which makes it possible to verify that a "small" lens applied to a text fragment or a partial tree produces the desired result.

4.1 Matching

Behind the scenes, when a lens is applied either to a string (in the *get* direction) or to a tree (in the *put* direction), it has to be *matched* to the current input for a variety of reasons, the most basic being to check whether a lens applies to the input at all.

In the *get* direction, this poses little difficulty and matching of a lens to a string boils down to matching a string to a regular expression. Regular expressions are restricted to the ones familiar from formal language theory, not the ones popular in various languages such as Perl, as those introduce extensions that leave the realm of regular languages. Some of the computations that the typechecker has to perform can be easily done with regular languages but become uncomputable when a broader class of languages, such as context-free languages, are considered. In practical terms, AUGEAS uses the notation of extended POSIX regular expressions, but does not support backreferences.[3]

[3]The implementation currently also lacks support for a few other features, such as named character classes, but unlike backreferences, those are supportable in principle.

Matching a tree in the *put* direction to a lens is more complicated than string matches for the *get* direction. To avoid implementing a mechanism that matches trees against a full tree schema, AUGEAS defines tree matching solely in terms of matching the labels at one level of the tree against a regular expression. For example, a lens that produces any number of nodes labelled `a` followed by a node labelled `b`, matches any tree that has such a sequence of nodes at its root level, regardless of the structure of the trees underneath each `a` and `b` node. This simplification makes it possible to reduce tree matching to matching regular expressions against strings. A tree with two nodes labelled `a` followed by a node labelled `b` and a node labelled `c` at its root level is converted into the string `a/a/b/c/` for purposes of matching, and the lens mentioned above is converted to the regular expression `(a/)*b/`. Clearly, this lens does not match the tree `a/a/b/c/`.

4.2 Lens primitives

AUGEAS has a handful of lens primitives; strictly speaking, the builtins are functions that, given a regular expression, indicated as *re*, or a string, indicated by *str*, or both, produce lenses.

Tree nodes are constructed by the *subtree* lens combinator discussed in the next section. The lens primitives lay the groundwork for the *subtree* lens: they mark which parts of the input to use as a tree label, which to store as the node's value, and which to omit from the tree.

- *key re* matches the regular expression *re* in the *get* direction and tells the *subtree* lens to use that as the label of the tree node it is constructing. In the *put* direction, it outputs the label of the current tree node.
- *label str* does not consume any input in the *get* direction, nor does it produce output in the *put* direction. It simply tells the *subtree* lens to use the *str* as the label of a tree node.
- *seq str* is similar to *label*; in the *get* direction, it sets the label of the enclosing *subtree* to a number. When that subtree is used in an iteration, the numbers are consecutive, starting from 1. The *str* argument is used to distinguish between separate sequences. In the *put* direction, the *seq* lens expects a tree node labelled with any positive number. There

is also a *counter str* lens whose sole effect it is to reset the counter with the given name back to 1 in the *get* direction.

- *store re* matches the regular expression *re* in the *get* direction and tells the *subtree* lens to use that as the value of the tree node it is constructing. In the *put* direction, it outputs the value of the current tree node.
- *del re str* matches the regular expression *re* in the *get* direction and suppresses any matches from inclusion in the tree. In the *put* direction, it restores the match in the output, if the current tree node corresponds to preexisting input, or outputs the default *str* if the current tree node was added to the tree and does not have any counterpart in the original input.

4.3 Lens combinators

Besides the *subtree* lens, there are a few more lens combinators that make it possible to build complicated lenses from the five lens primitives listed above. In the following, l, l_1, and l_2 always refer to arbitrary lenses:

- The *subtree* lens, written as $[l]$, applies l in the *get* direction to the input and constructs a new tree node based on the results of $l.get$. It uses whatever l marked as label and value for the new tree node; if l contains other *subtree* lenses, the trees constructed by them become the children of the new tree node.
- Lens concatenation, written as $l_1 \cdot l_2$, applies first l_1 and then l_2. In the *get* direction, the tree produced by l_1 is concatenated with that produced by l_2; similarly, in the *put* direction, the current tree is split and the first part is passed to the *put* direction of l_1, and the second part to the *put* direction of l_2.
- Lens union, $l_1|l_2$, chooses one of l_1 or l_2 and applies it. Which one is chosen depends on which one matches the current text in the *get* direction, or the current tree in the *put* direction.
- Lens iteration, l^* and l^+, applies l as long as it matches the current text in the *get* direction and the current tree in the *put* direction.

When AUGEAS processes a file with a lens l, it expects that the lens for that file processes the file in its entirety: that means that $l.get$ has to match the whole contents of the file. If it only matches partially, AUGEAS flags that as an error and refuses to produce the tree for that file. Similarly, when AUGEAS writes the tree back to a file, it expects that the entire subtree for that file, for example, everything under `/files/etc/hosts`, gets written out to file. It is an error if any nodes in that subtree are not written, or if required nodes (such as the canonical name for an entry in `/etc/hosts`) are missing from the tree.

5 Writing schemas

The description of how files are to be mapped to the tree, and the tree back into files are defined in AUGEAS' domain-specific language. The language is a functional language, following the syntactic conventions of ML. Figure 1 shows the definitions needed to process `/etc/hosts`.

Schema descriptions are divided into modules, one per file. A module can contain `autoload` directives and names defined with `let`. AUGEAS' language is strongly typed, and statically typechecked; this ensures that as many checks as possible are performed without ever transforming a single file. The available types are `string`, `regexp`, `lens`, `filter`, and `transform`—the last two are only needed for describing which lens is applied to what file.

String literals are enclosed in double-quotes, and can use the escape sequences familiar from C. Regular expression literals are enclosed in forward slashes and use extended POSIX syntax.

The most important part of the listing in Figure 1 is line 16, which defines the lens used to transform a whole `/etc/hosts` file. Strictly speaking, lenses are only ever applied to strings; finding files and reading and writing their contents is done by transforms. The transform `xfm` combines the lens `lns` and a filter that includes the one file `/etc/hosts`. Transforms are used by `augtool` when it starts up to find all the files it needs to load; to this end, it looks for all transforms in modules on its search path that are marked for autoload, as the transform `xfm` is on line 2 in the example.

The `/etc/hosts` file is line-oriented, with lines further subdivided in fields separated by whitespace. The fields are the IP address, the canonical name, and an

```
 1: module Hosts =
 2:   autoload xfm

 4:  let sep_tab = del /[ \t]+/ "\t"
 5:  let sep_spc = del /[ \t]+/ " "
 6:  let eol = del "\n" "\n"

 8:  let comment = [ del /#.*\n/ "# " ]
 9:  let word = /[^# \n\t]+/
10:  let host = [ seq "host" .
11:                [ label "ipaddr" . store  word ] . sep_tab .
12:                [ label "canonical" . store word ] .
13:                [ label "alias" . sep_spc . store word ]*
14:              . eol ]

16:  let lns = ( comment | host ) *

18:  let xfm = transform lns (incl "/etc/hosts")
```

Figure 1: The definition of the lenses used for mapping `/etc/hosts` into the tree and back.

```
127.0.0.1       localhost
192.168.0.2     server
# A comment
192.168.0.3     ns
```

(a) Restoring comments by position

```
127.0.0.1       localhost
192.168.0.1     router
# A comment
192.168.0.2     server
192.168.0.3     ns
```

(b) Initial `/etc/hosts`

```
127.0.0.1       localhost
# A comment
192.168.0.2     server
192.168.0.3     ns
```

(c) Restoring comments by key

Figure 2: Two possibilities of restoring comments in a changed file. After removing the tree node for `192.168.0.1` from the tree for the initial file *(middle)*, the tree can either be transformed so that the comment is restored at the same position *(left)*, or so that the comment is restored by its key *(right)*.

arbitrary number of aliases for a host. Lines starting with # are comments. Accordingly, the lens `lns` on line 16 processes any combination of matches for the `comment` and `host` lens.

The `comment` lens on line 8 deletes any line matching the regular expression `/#.*\n/`, i.e. anything from a starting # to the end of the line. Since AUGEAS requires that a file in its entirety is matched, there is no need to anchor regular expressions at the start and end of lines with `^` or `$`. The `del` primitive is enclosed in a subtree construct `[...]`; that causes the tree to contain a node with `NULL` label and value for every comment in the file. The reason for doing this has to do with how the *put* direction of *del* restores text: conceptually, the *get* direction of lenses produces a skeleton of the parsed text consisting of all the text deleted by the *del* lens with "holes" to fill in the parts stored in the tree. The *put* direction traverses the tree and fills the holes. The skeletons are associated with the parent node in the tree. If the comments were not their own tree node, AUGEAS would treat the whole `/etc/hosts` file as consisting of some comments with a fixed number of host entries between the comments. As an example, consider the initial file in Figure 2. After the initial file shown in the middle is read into the tree and the tree node corresponding to `192.168.0.1` is deleted, there are two ways in which the comment can be preserved when the tree is saved back to file: either by putting the comment after the second host entry (by position) as shown in Figure 4.3 or as coming after the entry for `127.0.0.1` but before the entry for `192.168.0.2` (by key). The former behavior results from *not* enclosing the *del* for `comment` in a subtree, the latter is the behavior of the lens in Figure 1.

The `host` lens on lines 10–14 in Figure 1 is straight-

forward in comparison: it stores host entries in separate subtrees, labelled with the number of the host entry. Each such subtree consists of a node labelled `ipaddr`, followed by a node labelled `canonical`, followed by zero or more nodes labelled `alias`. The value for each of the nodes is taken from splitting the line along spaces. The only difference between the `sep_tab` and `sep_spc` lenses that consume the whitespace between tokens on a line is how they behave when the tree is modified so that a brand new host entry is written to the file: `sep_tab` produces a tab character in that case, whereas `sep_spc` produces a space character.

The example in Figure 1 also illustrates two different ways to transform array-like constructs into the tree: the whole `/etc/hosts` file can be viewed as an array of lines of host entries (ignoring comments for the moment), and the aliases for each host are an array of space-separated tokens. For the former, we used the `seq` lens to produce a tree node for each host entry, wherease for the latter, we simply produce a new node with label `alias` for each alias. The reason for this is again connected to how formatting is preserved when entries are deleted from the tree or added to it. The former construct, using `seq` restores spacing by key, the number of the host entry in this case, whereas the latter restores it by position. When a new host entry is inserted into the tree under a new key, e.g. `10000`, all existing entries keep their spacing, since the skeletons for each entry are restored using that key. On the other hand, when a new alias for a host is inserted into the tree as the first alias for the host, the spacing is restored by position, so that the space between the (new) first and second alias is the same as the space that was in the initial file between the (old) first and second alias. Generally, it is preferrable to map arrays into the tree by using the same fixed label repeatedly, as is done for aliases here, since it makes the tree easier to manipulate, but often, considerations of what it means to preserve formatting in an "intuitive" way require that constructs using `seq` be used.

5.1 Lens development

When developing a lens for a new file format, the needed lens is gradually built up from simpler lenses and tested against appropriate text or tree fragments. For example, with the definitions from Figure 1, we can add a test

```
test lns get
  "127.0.0.1 localhost.localdomain localhost" = ?
```

to that same file. Running the modified file through `augparse` prints the tree resulting from applying the *get* direction of `lns` to the given string; `augparse` is a companion to `augtool` geared towards lens development.

There are two different kinds of tests: `test LNS get STR = RESULT` applies the *get* direction of `LNS` to `STR` and compares the resulting tree to `RESULT`, or prints it if `RESULT` is `?`. Conversely, `test LNS put STR after COMMANDS = RESULT` first applies the *get* direction of `LNS` to `STR`; it then changes the resulting tree using the `COMMANDS`, which can change the tree similar to `augtool`'s `set`, `rm`, and `ins` commands, and transforms the modified tree back to a string using the *put* direction of `LNS`. The test succeeds if this string equals the given `RESULT` string.

5.2 The typechecker

When configuration files are modified programmatically, ensuring that the changed configuration files are still valid is a major concern. AUGEAS contains a typechecker, very closely modelled on `Boomerang`'s [1] typechecker, that helps guard against common problems that could lead to invalid configuration files. Typechecking is performed statically—in other words, based solely on the schema description, to help weed out problems before any file is ever transformed according to that schema.

Typechecking happens in two phases: the first phase performs fairly standard checks that arguments to functions and operators have the type required by those functions and operators, for example, to ensure that the *subtree* operator `[...]` is only applied to lenses, and not to strings or regular expressions.

The second phase checks lenses for certain problems as they are constructed from simpler lenses. The details of those checks are based on the theoretical foundation laid by `Boomerang` [1] and make heavy use of computations on the regular languages matched by those lenses. In essence, the checks ensure that the lens laws GETPUT and PUTGET hold for any lens that is constructed.

Explaining these checks in detail would triple this paper in size; instead, let us just look at one of them to provide a taste of what the typechecker does. For two lenses l_1 and l_2, call the regular expressions they match in the *get*

direction r_1 and r_2. The *get* direction of the concatenation $l = l_1 \cdot l_2$ of these two lenses matches the concatenation $r = r_1 r_2$ of their underlying regular expressions. When $l.get$ is applied to a string u matching r, it needs to split it into two strings $u = u_1 u_2$ and then pass u_1 to $l_1.get$ and u_2 to $l_2.get$. The two lenses l_1 and l_2 may do completely different things with these strings, for example, l_1 may be a *del* lens and l_2 a *store* lens. It is therefore imperative that there be no ambiguity in how u is split into two strings; otherwise, the way u is processed by l, and therefore the resulting tree, would depend on arcane implementation details of the split operation, and may change unexpectedly as code is changed.

The typechecker therefore checks every time a concatentation of two lenses is formed to ensure that the regular languages matched by them are *unambiguously concatenable*; in other words, that each string u matching r can be split in exactly one way in one part matching r_1 and one part matching r_2.

Similar checks are performed for iteration of lenses to ensure that a string matching an iterated lens l^* can be split in exactly one way into n pieces matching l. For the union $l_1|l_2$ of lenses, checks are performed to ensure that whether l_1 or l_2 is chosen is guaranteed to be unique.

The lens laws impose restrictions both on the *get* and the *put* direction of lenses, and both are enforced by the typechecker. The concatenation of two lenses is not only restricted by the requirement that any input string in the *get* direction can be split unambiguously, but also by the requirement that any tree in the *put* direction can be split unambiguously. Splitting (for concatenation and iteration) and choice (for union) of trees is performed solely on the labels of the immediate children of the node under consideration. This has the advantage that it is easy to implement, and can be easily reduced to checks similar to those for the *get* direction, but has the disadvantage that as far as the typechecker and the *put* direction of lenses are concerned, all tree nodes labeled `foo` are identical, no matter whether they are a leaf or whether they are the root of complicated subtrees. This simple approach to matching trees has not yet led to any significant problems in practice, but it is conceivable that a more sophisticated approach to trees and tree schemas is needed at some point in the not-too-distant future.

6 Future Work

The current implementation of AUGEAS is useful, but by no means complete. Improvements can be made in almost every area: first and foremost is the task of expanding the set of configuration files that AUGEAS can process "out-of-the-box."

Several limitations of the current implementation would be particularly interesting to remove. First, the public API lacks support for recursive matching. While path expressions can use the $\star$ wildcard operator, that operator does only match one level in the tree. An operator that matches multiple levels at once would be very useful, similar to the $\star\star$ extension to filename globbing in some programs, or the `//` operator in XPath expressions. Other changes to the public API, such as efficient iteration over large parts of the tree, are desirable.

The language currently misses the concept of permutations; for example, in some files entries take options, similar to shell commands. If individual options are processed by lenses $l_1, l_2, \ldots, l_n$, there is no convenient way in the language to construct a lens that matches a permutation of these n lenses; permutations either need to be written out manually or approximated with a construct like $(l_1|l_2|\ldots|l_n)^*$. Because of the combinatorial complexity involved, a straightforward implementation of permutations will be of limited use in practice. Instead, adding an operator like RelaxNG's `interleave` to the language seems more promising, but even that will require some special care to keep the runtime of the typechecker bearable.

AUGEAS can only handle file formats that can be described as a regular language. In particular, file formats that have constructs that can be nested arbitrarily deep can not be processed by AUGEAS. That is a fairly severe limitation in practice, as it precludes processing of the very popular `httpd.conf`: the Apache configuration allows some constructs, most notably `IfModule`, that can be nested to an arbitrary depths. While it is not terribly hard to expand the implementation to process such non-regular file formats, enhancing the typechecker to handle them is hard. The most promising approach is to expand the class of file formats that AUGEAS accepts only very slightly, e.g. by allowing *balanced languages*, but not all context-free languages, and basing typechecking such file formats off suitable regular approximations of the file format.

Another area of possible improvements are services built on top of AUGEAS: `system-config-boot`, one of the graphical configuration tools shipped with Fedora, contains some experimental code that separates the user interface from the logic changing `/etc/grub.conf` through DBus. The UI sends messages to a DBus activated service that checks the users credentials with PolicyKit and, if the users is authorized to make the change, uses Augeas to edit `/etc/grub.conf`. The backend does not need any specific knowledge about the file being edited, and it would be fairly easy to expand this code to add permissioning to configuration changes that distinguishes between different nodes in the Augeas tree, even if those nodes ultimately come from the same file.

In a similar vein, it would be interesting to investigate a remote-able configuration API built on top of Augeas, where a special daemon allows remote counterparts to modify a system's configuration.

Acknowledgments

We wish to thank Benjamin Pierce and Nathon Foster and their collaborators for their work on `Harmony` and `Boomerang`. Anders Moeller provided invaluable input, not the least through his `dk.brics.automaton` Java package, for the finite automata library used by the type checker.

References

[1] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, January 2008.

[2] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`.

[3] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007. Preliminary version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004; extended abstract presented at *Principles of Programming Languages (POPL)*, 2005.

[4] Benjamin Pierce *et al.* Harmony. `http://alliance.seas.upenn.edu/~harmony`.